

**Radar Toolbox**  
Reference



**MATLAB® & SIMULINK®**

R2022a



# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

## *Radar Toolbox Reference*

© COPYRIGHT 2021–2022 by The MathWorks, Inc.

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

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

## **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://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](http://www.mathworks.com/patents) for more information.

## **Revision History**

March 2021	Online only	New for Version 1.0 (R2021a)
September 2021	Online only	Revised for Version 1.1 (R2021b)
March 2022	Online only	Revised for Version 1.2 (R2022a)

<b>1</b>	<b>Functions</b>
<b>2</b>	<b>Blocks</b>
<b>3</b>	<b>Apps</b>
<b>4</b>	<b>Objects</b>



# Functions

---

## radareqpow

Peak power estimate from radar equation

### Syntax

```
Pt = radareqpow(lambda,tgtrng,SNR,tau)
Pt = radareqpow(lambda,tgtrng,SNR,tau,Name,Value)
```

### Description

`Pt = radareqpow(lambda,tgtrng,SNR,tau)` estimates the peak transmit power, `Pt`, required for a radar operating at a wavelength of `lambda` meters to achieve the specified signal-to-noise ratio, `SNR`, in decibels for a target at a range of `tgtrng` meters. `tau` is the pulse width. The target has a nonfluctuating radar cross section (RCS) of 1 square meter.

`Pt = radareqpow(lambda,tgtrng,SNR,tau,Name,Value)` estimates the required peak transmit power with additional options specified by one or more `Name,Value` pair arguments.

### Examples

#### Compute Required Transmit Power

Estimate the required peak transmit power required to achieve a minimum SNR of 6 dB for a target at a range of 50 km. The target has a nonfluctuating RCS of 1 m<sup>2</sup>. The radar operating frequency is 1 GHz. The pulse duration is 1  $\mu$ s.

```
fc = 1.0e9;
lambda = physconst('LightSpeed')/fc;
tgtrng = 50e3;
tau = 1e-6;
SNR = 6;
Pt = radareqpow(lambda,tgtrng,SNR,tau)

Pt = 2.1996e+05
```

#### Compute Required Transmit Power at Specified System Temperature

Estimate the required peak transmit power required to achieve a minimum SNR of 10 dB for a target with an RCS of 0.5 m<sup>2</sup> at a range of 50 km. The radar operating frequency is 10 GHz. The pulse duration is 1  $\mu$ s. Assume a transmit and receive gain of 30 dB and an overall loss factor of 3 dB. The system temperature is 300 K.

```
fc = 10.0e9;
lambda = physconst('LightSpeed')/fc;
Pt = radareqpow(lambda,50e3,10,1e-6,'RCS',0.5, ...
    'Gain',30,'Ts',300,'Loss',3)

Pt = 2.2809e+06
```

## Compute Required Transmit Power for Bistatic Radar

Estimate the required peak transmit power for a bistatic radar to achieve a minimum SNR of 6 dB for a target with an RCS of 1 m<sup>2</sup>. The target is 50 km from the transmitter and 75 km from the receiver. The radar operating frequency is 10 GHz and the pulse duration is 10 μs. The transmitter and receiver gains are 40 dB and 20 dB, respectively.

```
fc = 10.0e9;
lambda = physconst('LightSpeed')/fc;
SNR = 6;
tau = 10e-6;
TxRng = 50e3;
RvRng = 75e3;
TxRvRng = [TxRng RvRng];
TxGain = 40;
RvGain = 20;
Gain = [TxGain RvGain];
Pt = radareqpow(lambda, TxRvRng, SNR, tau, 'Gain', Gain)

Pt = 4.9492e+04
```

## Input Arguments

### lambda — Wavelength of radar operating frequency

positive scalar

Wavelength of radar operating frequency, specified as a positive scalar. The wavelength is the ratio of the wave propagation speed to frequency. Units are in meters. For electromagnetic waves, the speed of propagation is the speed of light. Denoting the speed of light by  $c$  and the frequency (in hertz) of the wave by  $f$ , the equation for wavelength is:

$$\lambda = \frac{c}{f}$$

Data Types: double

### tgtrng — Target range

positive scalar | two-element row vector of positive values | length- $J$  column vector of positive values |  $J$ -by-2 matrix of positive values

Target ranges for a monostatic or bistatic radar.

- Monostatic radar - the transmitter and receiver are co-located. `tgtrng` is a real-valued positive scalar or length- $J$  real-valued positive column vector.  $J$  is the number of targets.
- Bistatic radar - the transmitter and receiver are separated. `tgtrng` is a 1-by-2 row vector with real-valued positive elements or a  $J$ -by-2 matrix with real-valued positive elements.  $J$  is the number of targets. Each row of `tgtrng` has the form `[TxRng RxRng]`, where `TxRng` is the range from the transmitter to the target and `RxRng` is the range from the receiver to the target.

Units are in meters.

Data Types: double

**SNR — Input signal-to-noise ratio at receiver**scalar | length- $J$  real-valued vector

Input signal-to-noise ratio (SNR) at the receiver, specified as a scalar or length- $J$  real-valued vector.  $J$  is the number of targets. Units are in dB.

Data Types: double

**tau — Single pulse duration**

positive scalar

Single pulse duration, specified as a positive scalar. Units are in seconds.

Data Types: double

**Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'RCS', 3.0

**RCS — Radar cross section**1 (default) | positive scalar | length- $J$  vector of positive values

Radar cross section specified as a positive scalar or length- $J$  vector of positive values.  $J$  is the number of targets. The target RCS is nonfluctuating (Swerling case 0). Units are in square meters.

Data Types: double

**Ts — System noise temperature**

290 (default) | positive scalar

System noise temperature, specified as a positive scalar. The system noise temperature is the product of the system temperature and the noise figure. Units are in Kelvin.

Data Types: double

**Gain — Transmitter and receiver gains**

20 (default) | scalar | real-valued 1-by-2 row vector

Transmitter and receiver gains, specified as a scalar or real-valued 1-by-2 row vector. When the transmitter and receiver are co-located (monostatic radar), `Gain` is a real-valued scalar. Then, the transmit and receive gains are equal. When the transmitter and receiver are not co-located (bistatic radar), `Gain` is a 1-by-2 row vector with real-valued elements. If `Gain` is a two-element row vector it has the form `[TxGain RxGain]` representing the transmit antenna and receive antenna gains.

Example: [15, 10]

Data Types: double

**Loss — System losses**0 (default) | scalar | length- $J$  real-valued vector

System losses, specified as a scalar. Units are in dB.



Example: 1

Data Types: double

### **AtmosphericLoss — Atmospheric absorption loss**

0 (default) | scalar | two-element row vector of real values | length- $J$  column vector of real values |  $J$ -by-2 matrix of real values

Atmospheric absorption losses for the transmit and receive paths.

- When the absorption is a scalar or length- $J$  column vector, the loss specifies the atmospheric absorption loss for a one-way path.
- When the absorption is a 1-by-2 row vector or  $J$ -by-2 column vector, the first column specifies the atmospheric absorption loss for the transmit path and the second column contains the atmospheric absorption loss for the receive path

Example: [10,20]

Data Types: double

### **PropagationFactor — Propagation factor**

0 (default) | scalar | two-element row vector of real values | length- $J$  column vector of real values |  $J$ -by-2 matrix of real values

Propagation factor for the transmit and receive paths.

- When the propagation factor is a scalar or length- $J$  column vector, the propagation factor is specified for a one-way path.
- When the propagation factor is a 1-by-2 row vector or  $J$ -by-2 column vector, the first column specifies the propagation factor for the transmit path and the second column contains the propagation factor for the receive path

Units are in dB.

Example: [10,20]

Data Types: double

### **CustomFactor — Custom factor**

0 (default) | scalar | length- $J$  column vector of real values

Custom loss factors specified as a scalar or length- $J$  column vector of real values.  $J$  is the number of targets. These factors contribute to the reduction of the received signal energy and can include range-dependent STC, eclipsing, and beam-dwell factors. Units are in dB.

Example: [10,20]

Data Types: double

## **Output Arguments**

### **Pt — Transmitter peak power**

positive scalar

Transmitter peak power, returned as positive scalar. Units are in watts.

## More About

### Point Target Radar Range Equation

The point target radar range equation estimates the power at the input to the receiver for a target of a given radar cross section at a specified range. The model is deterministic and assumes isotropic radiators. The equation for the power at the input to the receiver is

$$P_r = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 R_t^2 R_r^2 L}$$

where the terms in the equation are:

- $P_t$  — Peak transmit power in watts
- $G_t$  — Transmit antenna gain
- $G_r$  — Receive antenna gain. If the radar is monostatic, the transmit and receive antenna gains are identical.
- $\lambda$  — Radar wavelength in meters
- $\sigma$  — Target's nonfluctuating radar cross section in square meters
- $L$  — General loss factor in decibels that accounts for both system and propagation loss
- $R_t$  — Range from the transmitter to the target
- $R_r$  — Range from the receiver to the target. If the radar is monostatic, the transmitter and receiver ranges are identical.

Terms expressed in decibels, such as the loss and gain factors, enter the equation in the form  $10^{x/10}$  where  $x$  denotes the variable. For example, the default loss factor of 0 dB results in a loss term of  $10^{0/10}=1$ .

### Receiver Output Noise Power

The equation for the power at the input to the receiver represents the *signal* term in the signal-to-noise ratio. To model the noise term, assume the thermal noise in the receiver has a white noise power spectral density (PSD) given by:

$$P(f) = kT$$

where  $k$  is the Boltzmann constant and  $T$  is the effective noise temperature. The receiver acts as a filter to shape the white noise PSD. Assume that the magnitude squared receiver frequency response approximates a rectangular filter with bandwidth equal to the reciprocal of the pulse duration,  $1/\tau$ . The total noise power at the output of the receiver is:

$$N = \frac{kTF_n}{\tau}$$

where  $F_n$  is the receiver *noise factor*.

The product of the effective noise temperature and the receiver noise factor is referred to as the *system temperature*. This value is denoted by  $T_s$ , so that  $T_s = TF_n$ .

### Receiver Output SNR

Define the output SNR. The receiver output SNR is:

$$\frac{P_r}{N} = \frac{P_t \tau G_t G_r \lambda^2 \sigma}{(4\pi)^3 k T_s R_t^2 R_r^2 L}$$

You can derive this expression using the following equations:

- Received signal power in “Point Target Radar Range Equation” on page 1-11
- Output noise power in “Receiver Output Noise Power” on page 1-11

### Theoretical Maximum Detectable Range

Compute the maximum detectable range of a target.

For monostatic radars, the range from the target to the transmitter and receiver is identical. Denoting this range by  $R$ , you can express this relationship as  $R^4 = R_t^2 R_r^2$ .

Solving for  $R$

$$R = \left( \frac{NP_t \tau G_t G_r \lambda^2 \sigma}{P_r (4\pi)^3 k T_s L} \right)^{1/4}$$

For bistatic radars, the theoretical maximum detectable range is the geometric mean of the ranges from the target to the transmitter and receiver:

$$\sqrt{R_t R_r} = \left( \frac{NP_t \tau G_t G_r \lambda^2 \sigma}{P_r (4\pi)^3 k T_s L} \right)^{1/4}$$

## References

- [1] Richards, M. A. *Fundamentals of Radar Signal Processing*. New York: McGraw-Hill, 2005.
- [2] Skolnik, M. *Introduction to Radar Systems*. New York: McGraw-Hill, 1980.
- [3] Willis, N. J. *Bistatic Radar*. Raleigh, NC: SciTech Publishing, 2005.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

### See Also

phased.Transmitter | phased.ReceiverPreamp | noisepow | radareqrng | radareqsnr | systemp

Introduced in R2021a

## radareqrng

Maximum theoretical range estimate

### Syntax

```
maxrng = radareqrng(lambda,SNR,Pt,tau)
maxrng = radareqrng(lambda,SNR,Pt,tau,Name,Value)
```

### Description

`maxrng = radareqrng(lambda,SNR,Pt,tau)` estimates the theoretical maximum detectable range `maxrng` for a radar operating with a wavelength of `lambda` meters with a pulse duration of `Tau` seconds. The signal-to-noise ratio is `SNR` decibels, and the peak transmit power is `Pt` watts.

`maxrng = radareqrng(lambda,SNR,Pt,tau,Name,Value)` estimates the theoretical maximum detectable range with additional options specified by one or more `Name,Value` pair arguments.

### Examples

#### Estimate Maximum Detectable Range

Estimate the theoretical maximum detectable range for a monostatic radar operating at 10 GHz using a pulse duration of 10  $\mu$ s. Assume the output SNR of the receiver is 6 dB.

```
lambda = physconst('LightSpeed')/10e9;
SNR = 6;
tau = 10e-6;
Pt = 1e6;
maxrng = radareqrng(lambda,SNR,Pt,tau)

maxrng = 4.1057e+04
```

#### Estimate Maximum Detectable Range With Target RCS

Estimate the theoretical maximum detectable range for a monostatic radar operating at 10 GHz using a pulse duration of 10  $\mu$ s. The target RCS is 0.1 m<sup>2</sup>. Assume the output SNR of the receiver is 6 dB. The transmitter-receiver gain is 40 dB. Assume a loss factor of 3 dB.

```
lambda = physconst('LightSpeed')/10e9;
SNR = 6;
tau = 10e-6;
Pt = 1e6;
RCS = 0.1;
Gain = 40;
Loss = 3;
maxrng2 = radareqrng(lambda,SNR,Pt,tau,'Gain',Gain, ...
    'RCS',RCS,'Loss',Loss)
```

maxrng2 = 1.9426e+05

## Input Arguments

### **lambda** — Wavelength of radar operating frequency

positive scalar

Wavelength of radar operating frequency, specified as a positive scalar. The wavelength is the ratio of the wave propagation speed to frequency. Units are in meters. For electromagnetic waves, the speed of propagation is the speed of light. Denoting the speed of light by  $c$  and the frequency (in hertz) of the wave by  $f$ , the equation for wavelength is:

$$\lambda = \frac{c}{f}$$

Data Types: double

### **SNR** — Input signal-to-noise ratio at receiver

scalar | length- $J$  real-valued vector

Input signal-to-noise ratio (SNR) at the receiver, specified as a scalar or length- $J$  real-valued vector.  $J$  is the number of targets. Units are in dB.

Data Types: double

### **Pt** — Transmitted peak power

positive scalar

Transmitter peak power, specified as a positive scalar. Units are in watts.

Data Types: double

### **tau** — Single pulse duration

positive scalar

Single pulse duration, specified as a positive scalar. Units are in seconds.

Data Types: double

## Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: SNR, 10

### **RCS** — Radar cross section

1 (default) | positive scalar | length- $J$  vector of positive values

Radar cross section specified as a positive scalar or length- $J$  vector of positive values.  $J$  is the number of targets. The target RCS is nonfluctuating (Swerling case 0). Units are in square meters.

Data Types: double

**Ts — System noise temperature**

290 (default) | positive scalar

System noise temperature, specified as a positive scalar. The system noise temperature is the product of the system temperature and the noise figure. Units are in Kelvin.

Data Types: double

**Gain — Transmitter and receiver gains**

20 (default) | scalar | real-valued 1-by-2 row vector

Transmitter and receiver gains, specified as a scalar or real-valued 1-by-2 row vector. When the transmitter and receiver are co-located (monostatic radar), `Gain` is a real-valued scalar. Then, the transmit and receive gains are equal. When the transmitter and receiver are not co-located (bistatic radar), `Gain` is a 1-by-2 row vector with real-valued elements. If `Gain` is a two-element row vector it has the form `[TxGain RxGain]` representing the transmit antenna and receive antenna gains.

Example: `[15,10]`

Data Types: double

**Loss — System losses**0 (default) | scalar | length-*J* real-valued vector

System losses, specified as a scalar. Units are in dB.

Example: 1

Data Types: double

**CustomFactor — Custom factor**0 (default) | scalar | length-*J* column vector of real values

Custom loss factors specified as a scalar or length-*J* column vector of real values. *J* is the number of targets. These factors contribute to the reduction of the received signal energy and can include range-dependent STC, eclipsing, and beam-dwell factors. Units are in dB.

Example: `[10,20]`

Data Types: double

**unitstr — Units of the estimated maximum theoretical range**

'm' (default) | 'km' | 'mi' | 'nmi'

Units of the estimated maximum theoretical range, specified as one of:

- 'm' meters
- 'km' kilometers
- 'mi' miles
- 'nmi' nautical miles (U.S.)

**Output Arguments****maxrng — Estimated theoretical maximum detectable range**

positive scalar

The estimated theoretical maximum detectable range, returned as a positive scalar. The units of `maxrng` are specified by `unitstr`. For bistatic radars, `maxrng` is the geometric mean of the range from the transmitter to the target and the receiver to the target.

## More About

### Point Target Radar Range Equation

The point target radar range equation estimates the power at the input to the receiver for a target of a given radar cross section at a specified range. The model is deterministic and assumes isotropic radiators. The equation for the power at the input to the receiver is

$$P_r = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 R_t^2 R_r^2 L}$$

where the terms in the equation are:

- $P_t$  — Peak transmit power in watts
- $G_t$  — Transmit antenna gain
- $G_r$  — Receive antenna gain. If the radar is monostatic, the transmit and receive antenna gains are identical.
- $\lambda$  — Radar wavelength in meters
- $\sigma$  — Target's nonfluctuating radar cross section in square meters
- $L$  — General loss factor in decibels that accounts for both system and propagation loss
- $R_t$  — Range from the transmitter to the target
- $R_r$  — Range from the receiver to the target. If the radar is monostatic, the transmitter and receiver ranges are identical.

Terms expressed in decibels, such as the loss and gain factors, enter the equation in the form  $10^{x/10}$  where  $x$  denotes the variable. For example, the default loss factor of 0 dB results in a loss term of  $10^{0/10}=1$ .

### Receiver Output Noise Power

The equation for the power at the input to the receiver represents the *signal* term in the signal-to-noise ratio. To model the noise term, assume the thermal noise in the receiver has a white noise power spectral density (PSD) given by:

$$P(f) = kT$$

where  $k$  is the Boltzmann constant and  $T$  is the effective noise temperature. The receiver acts as a filter to shape the white noise PSD. Assume that the magnitude squared receiver frequency response approximates a rectangular filter with bandwidth equal to the reciprocal of the pulse duration,  $1/\tau$ . The total noise power at the output of the receiver is:

$$N = \frac{kTF_n}{\tau}$$

where  $F_n$  is the receiver *noise factor*.

The product of the effective noise temperature and the receiver noise factor is referred to as the *system temperature*. This value is denoted by  $T_s$ , so that  $T_s = TF_n$ .

### Receiver Output SNR

Define the output SNR. The receiver output SNR is:

$$\frac{P_r}{N} = \frac{P_t \tau G_t G_r \lambda^2 \sigma}{(4\pi)^3 k T_s R_t^2 R_r^2 L}$$

You can derive this expression using the following equations:

- Received signal power in “Point Target Radar Range Equation” on page 1-11
- Output noise power in “Receiver Output Noise Power” on page 1-11

### Theoretical Maximum Detectable Range

Compute the maximum detectable range of a target.

For monostatic radars, the range from the target to the transmitter and receiver is identical. Denoting this range by  $R$ , you can express this relationship as  $R^4 = R_t^2 R_r^2$ .

Solving for  $R$

$$R = \left( \frac{NP_t \tau G_t G_r \lambda^2 \sigma}{P_r (4\pi)^3 k T_s L} \right)^{1/4}$$

For bistatic radars, the theoretical maximum detectable range is the geometric mean of the ranges from the target to the transmitter and receiver:

$$\sqrt{R_t R_r} = \left( \frac{NP_t \tau G_t G_r \lambda^2 \sigma}{P_r (4\pi)^3 k T_s L} \right)^{1/4}$$

## References

- [1] Richards, M. A. *Fundamentals of Radar Signal Processing*. New York: McGraw-Hill, 2005.
- [2] Skolnik, M. *Introduction to Radar Systems*. New York: McGraw-Hill, 1980.
- [3] Willis, N. J. *Bistatic Radar*. Raleigh, NC: SciTech Publishing, 2005.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.



## **See Also**

phased.Transmitter | phased.ReceiverPreamp | noisepow | radareqpow | radareqsnr | systemp

## **Topics**

“Modeling Target Position Errors due to Refraction”

**Introduced in R2021a**

## radareqsnr

SNR estimate from radar equation

### Syntax

```
SNR = radareqsnr(lambda,tgtrng,Pt,tau)
SNR = radareqsnr(lambda,tgtrng,Pt,tau,Name,Value)
```

### Description

`SNR = radareqsnr(lambda,tgtrng,Pt,tau)` estimates the output signal-to-noise ratio, SNR, at the receiver based on the wavelength `lambda`, the range `tgtrng`, the peak transmit power `Pt`, and the pulse width `tau`.

`SNR = radareqsnr(lambda,tgtrng,Pt,tau,Name,Value)` estimates the output SNR at the receiver with additional options specified by one or more `Name,Value` pair arguments.

### Examples

#### Compute SNR Using Radar Equation

Estimate the output SNR for a target with an RCS of 1 m<sup>2</sup> at a range of 50 km. The system is a monostatic radar operating at 1 GHz with a peak transmit power of 1 MW and pulse width of 0.2  $\mu$ s. The transmitter and receiver gain is 20 dB. The system temperature has the default value of 290 K.

```
fc = 1.0e9;
lambda = physconst('LightSpeed')/fc;
tgtrng = 50e3;
Pt = 1e6;
tau = 0.2e-6;
snr = radareqsnr(lambda,tgtrng,Pt,tau)

snr = 5.5868
```

#### Compute SNR with Specified System Temperature

Estimate the output SNR for a target with an RCS of 0.5 m<sup>2</sup> at 100 km. The system is a monostatic radar operating at 10 GHz with a peak transmit power of 1 MW and pulse width of 1  $\mu$ s. The transmitter and receiver gain is 40 dB. The system temperature is 300 K and the loss factor is 3 dB.

```
fc = 10.0;
T = 300.0;
lambda = physconst('LightSpeed')/10e9;
snr = radareqsnr(lambda,100e3,1e6,1e-6,'RCS',0.5, ...
    'Gain',40,'Ts',T,'Loss',3)

snr = 14.3778
```

## Compute SNR for Bistatic Radar

Estimate the output SNR for a target with an RCS of 1 m<sup>2</sup>. The radar is bistatic. The target is located 50 km from the transmitter and 75 km from the receiver. The radar operating frequency is 10.0 GHz. The transmitter has a peak transmit power of 1 MW with a gain of 40 dB. The pulse width is 1 μs. The receiver gain is 20 dB.

```
fc = 10.0e9;
lambda = physconst('LightSpeed')/fc;
tau = 1e-6;
Pt = 1e6;
txrvRng =[50e3 75e3];
Gain = [40 20];
snr = radareqsnr(lambda,txrvRng,Pt,tau,'Gain',Gain)

snr = 9.0547
```

## Input Arguments

### lambda — Wavelength of radar operating frequency

positive scalar

Wavelength of radar operating frequency, specified as a positive scalar. The wavelength is the ratio of the wave propagation speed to frequency. Units are in meters. For electromagnetic waves, the speed of propagation is the speed of light. Denoting the speed of light by  $c$  and the frequency (in hertz) of the wave by  $f$ , the equation for wavelength is:

$$\lambda = \frac{c}{f}$$

Data Types: double

### tgtrng — Target range

positive scalar | two-element row vector of positive values | length- $J$  column vector of positive values |  $J$ -by-2 matrix of positive values

Target ranges for a monostatic or bistatic radar.

- Monostatic radar - the transmitter and receiver are co-located. `tgtrng` is a real-valued positive scalar or length- $J$  real-valued positive column vector.  $J$  is the number of targets.
- Bistatic radar - the transmitter and receiver are separated. `tgtrng` is a 1-by-2 row vector with real-valued positive elements or a  $J$ -by-2 matrix with real-valued positive elements.  $J$  is the number of targets. Each row of `tgtrng` has the form `[TxRng RxRng]`, where `TxRng` is the range from the transmitter to the target and `RxRng` is the range from the receiver to the target.

Units are in meters.

Data Types: double

### Pt — Transmitted peak power

positive scalar

Transmitter peak power, specified as a positive scalar. Units are in watts.

Data Types: double

**tau — Single pulse duration**

positive scalar

Single pulse duration, specified as a positive scalar. Units are in seconds.

Data Types: double

**Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'RCS', 5.0, 'Ts', 295

**RCS — Radar cross section**

1 (default) | positive scalar | length-*J* vector of positive values

Radar cross section specified as a positive scalar or length-*J* vector of positive values. *J* is the number of targets. The target RCS is nonfluctuating (Swerling case 0). Units are in square meters.

Data Types: double

**Ts — System noise temperature**

290 (default) | positive scalar

System noise temperature, specified as a positive scalar. The system noise temperature is the product of the system temperature and the noise figure. Units are in Kelvin.

Data Types: double

**Gain — Transmitter and receiver gains**

20 (default) | scalar | real-valued 1-by-2 row vector

Transmitter and receiver gains, specified as a scalar or real-valued 1-by-2 row vector. When the transmitter and receiver are co-located (monostatic radar), `Gain` is a real-valued scalar. Then, the transmit and receive gains are equal. When the transmitter and receiver are not co-located (bistatic radar), `Gain` is a 1-by-2 row vector with real-valued elements. If `Gain` is a two-element row vector it has the form `[TxGain RxGain]` representing the transmit antenna and receive antenna gains.

Example: [15, 10]

Data Types: double

**Loss — System losses**

0 (default) | scalar | length-*J* real-valued vector

System losses, specified as a scalar. Units are in dB.

Example: 1

Data Types: double

**AtmosphericLoss — Atmospheric absorption loss**

0 (default) | scalar | two-element row vector of real values | length- $J$  column vector of real values |  $J$ -by-2 matrix of real values

Atmospheric absorption losses for the transmit and receive paths.

- When the absorption is a scalar or length- $J$  column vector, the loss specifies the atmospheric absorption loss for a one-way path.
- When the absorption is a 1-by-2 row vector or  $J$ -by-2 column vector, the first column specifies the atmospheric absorption loss for the transmit path and the second column contains the atmospheric absorption loss for the receive path

Example: [10, 20]

Data Types: double

**PropagationFactor — Propagation factor**

0 (default) | scalar | two-element row vector of real values | length- $J$  column vector of real values |  $J$ -by-2 matrix of real values

Propagation factor for the transmit and receive paths.

- When the propagation factor is a scalar or length- $J$  column vector, the propagation factor is specified for a one-way path.
- When the propagation factor is a 1-by-2 row vector or  $J$ -by-2 column vector, the first column specifies the propagation factor for the transmit path and the second column contains the propagation factor for the receive path

Units are in dB.

Example: [10, 20]

Data Types: double

**CustomFactor — Custom factor**

0 (default) | scalar | length- $J$  column vector of real values

Custom loss factors specified as a scalar or length- $J$  column vector of real values.  $J$  is the number of targets. These factors contribute to the reduction of the received signal energy and can include range-dependent STC, eclipsing, and beam-dwell factors. Units are in dB.

Example: [10, 20]

Data Types: double

**Output Arguments****SNR — Minimum output signal-to-noise ratio at receiver**

scalar

Minimum output signal-to-noise ratio at the receiver, returned as a scalar. Units are in dB.

Data Types: double

## More About

### Point Target Radar Range Equation

The point target radar range equation estimates the power at the input to the receiver for a target of a given radar cross section at a specified range. The model is deterministic and assumes isotropic radiators. The equation for the power at the input to the receiver is

$$P_r = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 R_t^2 R_r^2 L}$$

where the terms in the equation are:

- $P_t$  — Peak transmit power in watts
- $G_t$  — Transmit antenna gain
- $G_r$  — Receive antenna gain. If the radar is monostatic, the transmit and receive antenna gains are identical.
- $\lambda$  — Radar wavelength in meters
- $\sigma$  — Target's nonfluctuating radar cross section in square meters
- $L$  — General loss factor in decibels that accounts for both system and propagation loss
- $R_t$  — Range from the transmitter to the target
- $R_r$  — Range from the receiver to the target. If the radar is monostatic, the transmitter and receiver ranges are identical.

Terms expressed in decibels, such as the loss and gain factors, enter the equation in the form  $10^{x/10}$  where  $x$  denotes the variable. For example, the default loss factor of 0 dB results in a loss term of  $10^{0/10}=1$ .

### Receiver Output Noise Power

The equation for the power at the input to the receiver represents the *signal* term in the signal-to-noise ratio. To model the noise term, assume the thermal noise in the receiver has a white noise power spectral density (PSD) given by:

$$P(f) = kT$$

where  $k$  is the Boltzmann constant and  $T$  is the effective noise temperature. The receiver acts as a filter to shape the white noise PSD. Assume that the magnitude squared receiver frequency response approximates a rectangular filter with bandwidth equal to the reciprocal of the pulse duration,  $1/\tau$ . The total noise power at the output of the receiver is:

$$N = \frac{kTF_n}{\tau}$$

where  $F_n$  is the receiver *noise factor*.

The product of the effective noise temperature and the receiver noise factor is referred to as the *system temperature*. This value is denoted by  $T_s$ , so that  $T_s = TF_n$ .

### Receiver Output SNR

Define the output SNR. The receiver output SNR is:

$$\frac{P_r}{N} = \frac{P_t \tau G_t G_r \lambda^2 \sigma}{(4\pi)^3 k T_s R_t^2 R_r^2 L}$$

You can derive this expression using the following equations:

- Received signal power in “Point Target Radar Range Equation” on page 1-11
- Output noise power in “Receiver Output Noise Power” on page 1-11

### Theoretical Maximum Detectable Range

Compute the maximum detectable range of a target.

For monostatic radars, the range from the target to the transmitter and receiver is identical. Denoting this range by  $R$ , you can express this relationship as  $R^4 = R_t^2 R_r^2$ .

Solving for  $R$

$$R = \left( \frac{NP_t \tau G_t G_r \lambda^2 \sigma}{P_r (4\pi)^3 k T_s L} \right)^{1/4}$$

For bistatic radars, the theoretical maximum detectable range is the geometric mean of the ranges from the target to the transmitter and receiver:

$$\sqrt{R_t R_r} = \left( \frac{NP_t \tau G_t G_r \lambda^2 \sigma}{P_r (4\pi)^3 k T_s L} \right)^{1/4}$$

## References

- [1] Richards, M. A. *Fundamentals of Radar Signal Processing*. New York: McGraw-Hill, 2005.
- [2] Skolnik, M. *Introduction to Radar Systems*. New York: McGraw-Hill, 1980.
- [3] Willis, N. J. *Bistatic Radar*. Raleigh, NC: SciTech Publishing, 2005.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

### See Also

`phased.Transmitter` | `phased.ReceiverPreamp` | `noisepow` | `radareqpow` | `radareqrng` | `systemp`

### Topics

“Radar Vertical Coverage over Terrain”

**Introduced in R2021a**



# blakechart

Range-angle-height (Blake) chart

## Syntax

```
blakechart(vcp,vcpangles)
blakechart(vcp,vcpangles,rmax,hmax)
blakechart( ____,Name,Value)
```

## Description

`blakechart(vcp,vcpangles)` creates a range-angle-height plot (also called a Blake chart) for a narrowband radar antenna. This chart shows the maximum radar range as a function of target elevation. In addition, the Blake chart displays lines of constant range and lines of constant height. The input consists of the vertical coverage pattern `vcp` and vertical coverage pattern angles `vcpangles`, both produced by `radarvcd`.

The range in the range-height-angle chart is the propagated range and the height is relative to the origin of the ray. It is assumed that the antenna height is less than 1000 ft (about 305 meters) above ground level. Normal atmospheric refraction is taken into account using the “CRPL Exponential Reference Atmosphere Model” on page 1-29. Scattering and ducting are assumed to be negligible.

`blakechart(vcp,vcpangles,rmax,hmax)`, in addition, specifies the maximum range and height of the Blake chart. You can specify range and height units separately in the name-value arguments `RangeUnit` and `HeightUnit`.

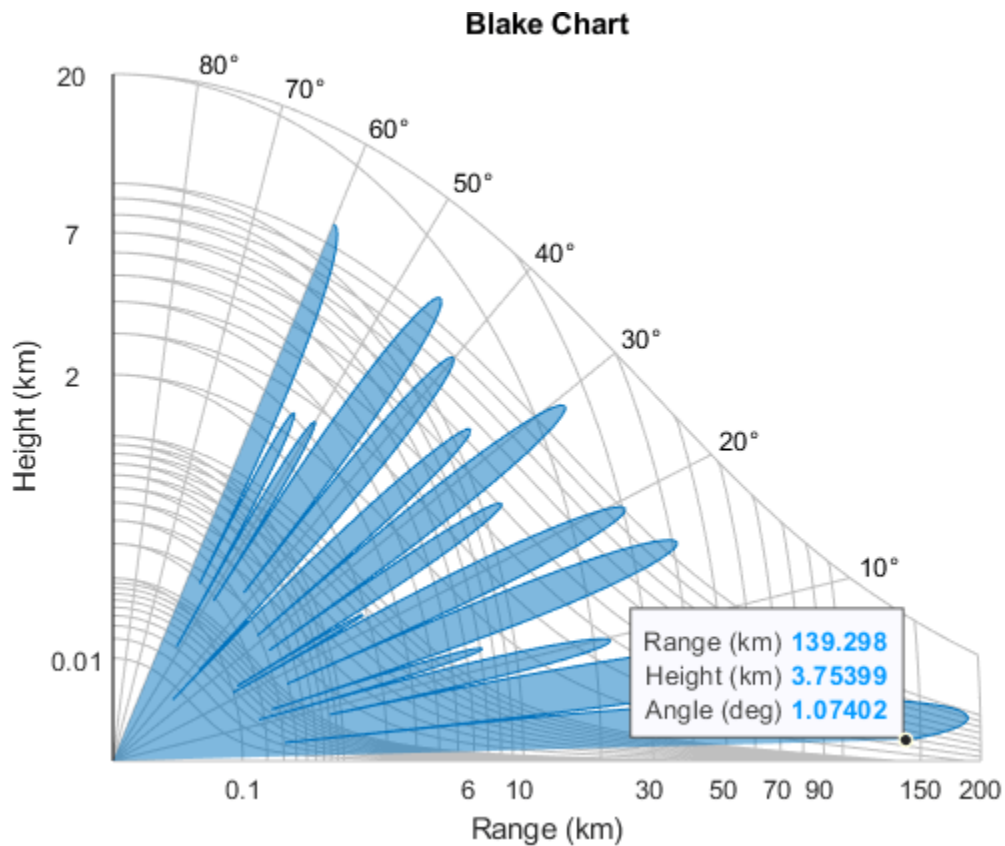
`blakechart( ____,Name,Value)` allows you to specify additional input parameters using name-value arguments. You can specify multiple name-value arguments in any order with any of the previous syntaxes.

## Examples

### Display Vertical Coverage Diagram

Display the vertical coverage diagram of an antenna transmitting at 100 MHz and placed 20 meters above the ground. Set the free-space range to 100 km. Use default plotting parameters.

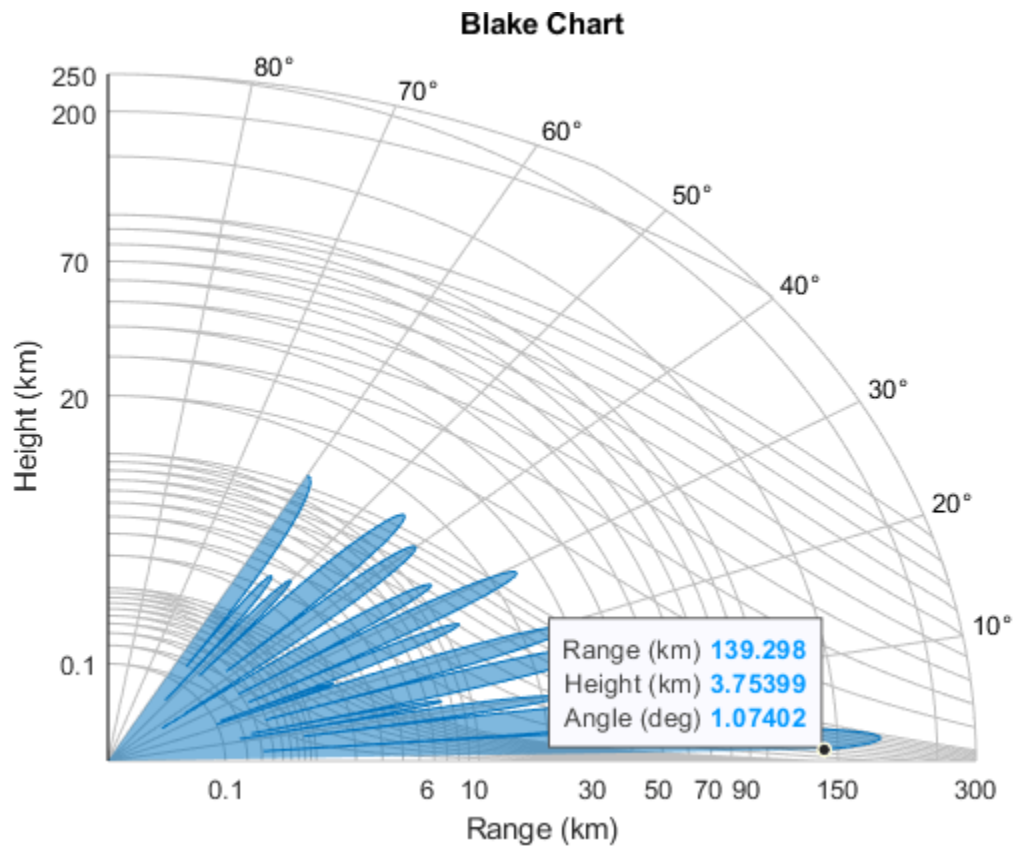
```
freq = 100e6;
ant_height = 20;
rng_fs = 100;
[vcp, vcpangles] = radarvcd(freq,rng_fs,ant_height);
blakechart(vcp, vcpangles);
```



### Display Vertical Coverage Diagram Specifying Maximum Range and Height

Display the vertical coverage diagram of an antenna transmitting at 100 MHz and placed 20 meters above the ground. Set the free-space range to 100 km. Set the maximum plotting range to 300 km and the maximum plotting height to 250 km.

```
freq = 100e6;
ant_height = 20;
rng_fs = 100;
[vcp, vcpangles] = radarvcd(freq, rng_fs, ant_height);
rmax = 300;
hmax = 250;
blakechart(vcp, vcpangles, rmax, hmax)
```



### Display Vertical Coverage Diagram of Sinc Pattern Antenna

Plot the range-height-angle curve of a radar having a sinc-function antenna pattern.

#### Specify antenna pattern

Specify the antenna pattern as a sinc function.

```
pat_angles = linspace(-90,90,361)';
pat_u = 1.39157/sind(90/2)*sind(pat_angles);
pat = sinc(pat_u/pi);
```

#### Specify radar and environment parameters

Set the transmitting frequency to 100 MHz, the free-space range to 100 km, the antenna tilt angle to 0°, and place the antenna 20 meters above the ground. Assume a surface roughness of one meter.

```
freq = 100e6;
ant_height = 10;
rng_fs = 100;
tilt_ang = 0;
surf_roughness = 1;
```

## Create radar range-height-angle data

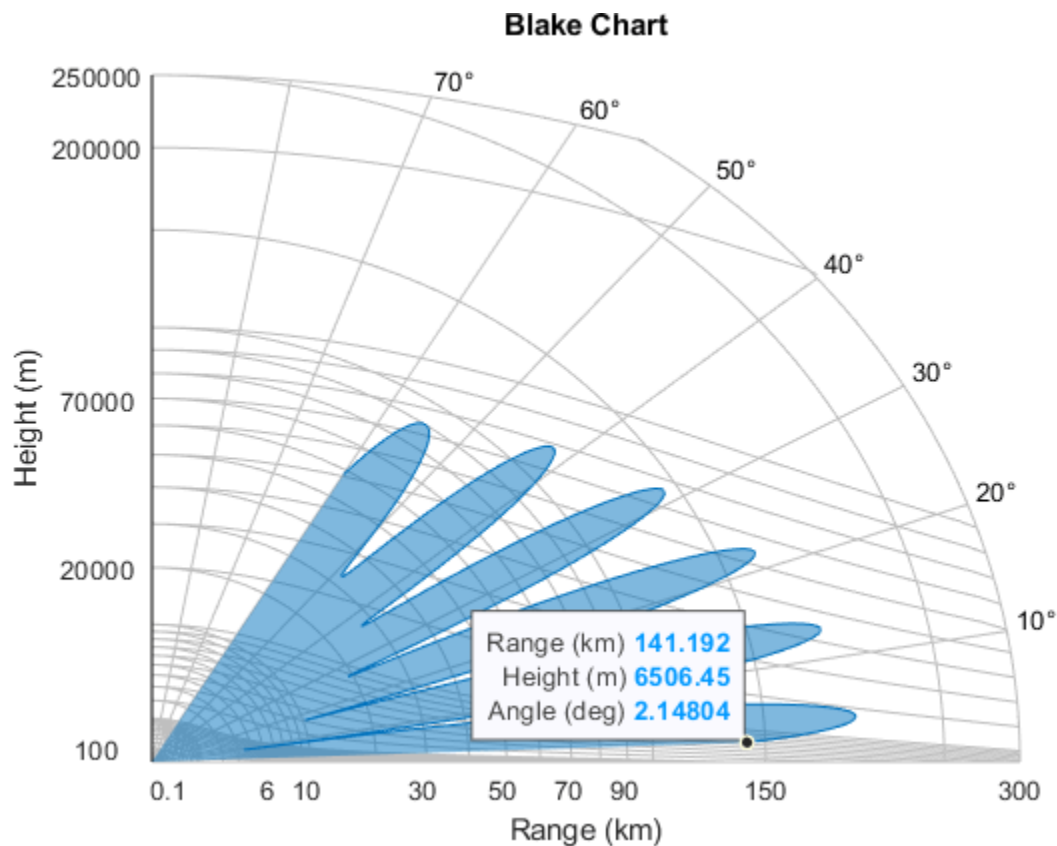
Obtain the vertical coverage pattern values and angles for the radar antenna.

```
[vcp, vcpangles] = radarvcd(freq, rng_fs, ant_height, ...
    'RangeUnit', 'km', 'HeightUnit', 'm', ...
    'AntennaPattern', pat, ...
    'PatternAngles', pat_angles, 'TiltAngle', tilt_ang, ...
    'SurfaceHeightStandardDeviation', surf_roughness/(2*sqrt(2)));
```

## Plot radar range-height-angle data

Set the maximum plotting range to 300 km and the maximum plotting height to 250,000 m. Choose the range units as kilometers, 'km', and the height units as meters, 'm'. Set the range and height axes scale powers to 1/2.

```
rmax = 300;
hmax = 250e3;
blakechart(vcp, vcpangles, rmax, hmax, 'RangeUnit', 'km', ...
    'ScalePower', 1/2, 'HeightUnit', 'm');
```



## Input Arguments

### vcp — Vertical coverage pattern

real-valued column vector | real-valued matrix

Vertical coverage pattern, specified as a real-valued column vector or matrix. The vertical coverage pattern is the actual maximum range of the radar. Each column of `vcp` corresponds to an individual vertical coverage pattern. Each row of `vcp` corresponds to one of the angles specified in `vcpangles`. Values are expressed in kilometers unless you change the unit of measure using the `RangeUnit` name-value argument.

Example: [282.3831; 291.0502; 299.4252]

Data Types: `double`

### **vcpangles — Vertical coverage pattern angles**

real-valued column vector

Vertical coverage pattern angles, specified as a real-valued column vector. Each element of `vcpangles` specifies the elevation angle in degrees at which a vertical coverage pattern is measured. The set of angles ranges from  $-90^\circ$  to  $90^\circ$ .

Example: [2.1480; 2.2340; 2.3199]

Data Types: `double`

### **rmax — Maximum range of plot**

real-valued scalar

Maximum range of plot, specified as a real-valued scalar. Range units are specified by the `'RangeUnit'` name-value argument.

Example: 200

Data Types: `double`

### **hmax — Maximum height of plot**

real-valued scalar

Maximum height of plot, specified as a real-valued scalar. Height units are specified by the `'HeightUnit'` name-value argument.

Example: 100000

Data Types: `double`

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'RangeUnit', 'm'`

### **RangeUnit — Radar range units**

`'km'` (default) | `'nmi'` | `'mi'` | `'ft'` | `'m'` | `'kft'`

Range units denoting nautical miles, miles, kilometers, feet, meters, or kilofeet. This name-value argument specifies the units for the vertical coverage pattern input argument, `vcp`, and the maximum range input argument `rmax`.

Example: `'mi'`

Data Types: char

**HeightUnit — Height units**

'km' (default) | 'nmi' | 'mi' | 'ft' | 'm' | 'kft'

Height units, specified as one of 'nmi', 'mi', 'km', 'ft', 'm', or 'kft' denoting nautical miles, miles, kilometers, feet, meters, or kilofeet, respectively. This name-value argument specifies the units for the maximum height `hmax`.

Example: 'm'

Data Types: char

**ScalePower — Scale power**

0.25 (default) | real-valued scalar

Scale power, specified as a scalar in the range [0, 1]. This argument specifies the range and height axis scale power.

Example: 0.5

Data Types: double

**SurfaceRefractivity — Surface refractivity**

313 (default) | real-valued scalar

Surface refractivity in N-units, specified as a nonnegative real-valued scalar. The surface refractivity is a parameter of the “CRPL Exponential Reference Atmosphere Model” on page 1-29 used by `blakechart`.

Data Types: double

**RefractionExponent — Refraction exponent**

0.143859 (default) | real-valued scalar

Refraction exponent, specified as a nonnegative real-valued scalar. The refraction exponent is a parameter of the “CRPL Exponential Reference Atmosphere Model” on page 1-29 used by `blakechart`.

Data Types: double

**AntennaHeight — Antenna height**

0 (default) | real-valued scalar

Antenna height, specified as a real-valued scalar. When you provide the antenna height, the height in the Blake chart is the height above ground level. Otherwise, the height in the Blake chart is relative to the origin of the ray, and the function assumes that the antenna is less than 1000 ft (about 305 m) above ground level. Use the `HeightUnit` argument to specify the units of `AntennaHeight`.

Data Types: double

**FaceColor — Face color of vertical coverage pattern patch**







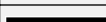

color name | short name | hexadecimal color code | RGB triplet | 'none'

Face color of vertical coverage pattern patch, specified as a color name, a short name, a hexadecimal color code, an RGB triplet, or 'none'. If you specify more than one color, the number of colors must match the number of columns of `vcp`.








For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range  $[0, 1]$ ; for example,  $[0.4 \ 0.6 \ 0.7]$ .
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB® uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'black'

Example: 'k'

Example: [0.850 0.325 0.098]

Example: '#D95319'

Data Types: double | char | string

### EdgeColor — Edge color of vertical coverage pattern patch

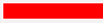







color name | short name | hexadecimal color code | RGB triplet | 'none'

Edge color of vertical coverage pattern patch, specified as a color name, a short name, a hexadecimal color code, an RGB triplet, or 'none'. If you specify more than one color, the number of colors must match the number of columns of `vcp`.








For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range  $[0, 1]$ ; for example,  $[0.4 \ 0.6 \ 0.7]$ .
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example: 'black'

Example: 'k'

Example: [0.850 0.325 0.098]

Example: '#D95319'



Data Types: `double` | `char` | `string`

## More About

### CRPL Exponential Reference Atmosphere Model

Atmospheric refraction evidences itself as a deviation in an electromagnetic ray from a straight line due to variation in air density as a function of height. The Central Radio Propagation Laboratory (CRPL) exponential reference atmosphere model treats refraction effects by assuming that the index of refraction  $n(h)$  and the refractivity  $N$  decay exponentially with height. The model defines

$$N = (n(h) - 1) \times 10^6 = N_s e^{-R_{\text{exp}} h},$$

where  $N_s$  is the atmospheric refractivity value (in units of  $10^{-6}$ ) at the surface of the earth,  $R_{\text{exp}}$  is the decay constant, and  $h$  is the height above the surface in kilometers. Thus

$$n(h) = 1 + (N_s \times 10^{-6}) e^{-R_{\text{exp}} h}.$$

The default value of  $N_s$  is 313 N-units and can be modified using the `SurfaceRefractivity` name-value argument in functions that accept it. The default value of  $R_{\text{exp}}$  is  $0.143859 \text{ km}^{-1}$  and can be modified using the `RefractionExponent` name-value argument in functions that accept it.

## References

- [1] Blake, Lamont V. *Machine Plotting of Radar Vertical-Plane Coverage Diagrams*. Naval Research Laboratory Report 7098, 1970.
- [2] Bean, B.R., and G.D. Thayer. "Central Radio Propagation Laboratory Exponential Reference Atmosphere." *Journal of Research of the National Bureau of Standards, Section D: Radio Propagation* 63D, no. 3 (November 1959): 315. <https://doi.org/10.6028/jres.063D.031>.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Apps

**Radar Designer**

### Functions

`el2height` | `height2el` | `height2range` | `height2grndrange` | `landroughness` | `radarvcd` | `range2height` | `refractionexp` | `searoughness`

### Topics

"Radar Vertical Coverage over Terrain"

"Modeling Target Position Errors due to Refraction"

### Introduced in R2021a

## radarvcd

Vertical coverage diagram

### Syntax

```
[vcp,vcpangles] = radarvcd(freq,rfs,anht)
[vcp,vcpangles] = radarvcd( ___,Name,Value)
```

```
radarvcd( ___ )
```

### Description

`[vcp,vcpangles] = radarvcd(freq,rfs,anht)` calculates the vertical coverage pattern of a narrowband radar antenna. The “Vertical Coverage Pattern” on page 1-38 is the range of the radar `vcp` as a function of elevation angle `vcpangles`. The vertical coverage pattern depends on three parameters: maximum free-space detection range of the radar `rfs`, the radar frequency `freq`, and the antenna height `anht`.

`[vcp,vcpangles] = radarvcd( ___,Name,Value)` allows you to specify additional input parameters using name-value arguments. You can specify multiple name-value arguments in any order.

`radarvcd( ___ )` displays the vertical coverage diagram for a radar system. The plot is the locus of points of maximum radar range as a function of target elevation. This plot is also known as the Blake chart. To create this chart, `radarvcd` invokes the function `blakechart` using default parameters. To produce a Blake chart with different parameters, first call `radarvcd` to obtain `vcp` and `vcpangles`. Then, call `blakechart` with user-specified parameters. This syntax can use any of the previous syntaxes.

### Examples

#### Plot Vertical Coverage Pattern Using Default Parameters

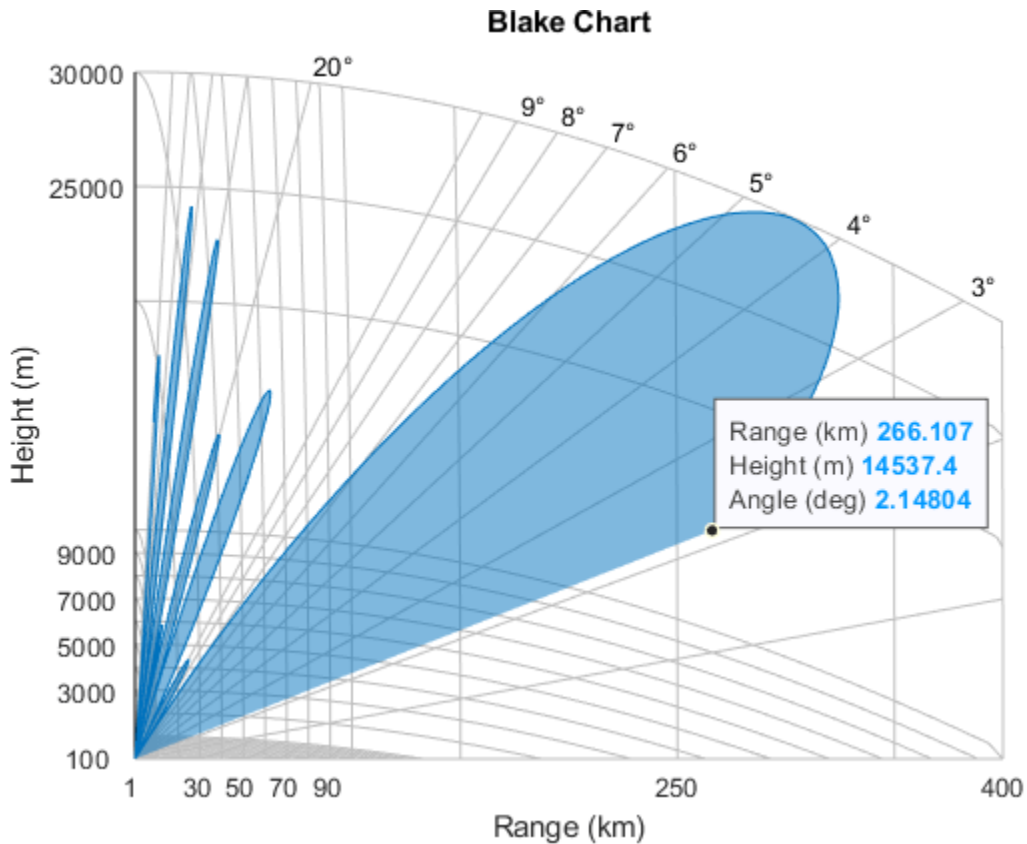
Set the frequency to 100 MHz, the antenna height to 10 m, and the free-space range to 200 km. The antenna pattern, surface roughness, antenna tilt angle, and field polarization assume their default values as specified in the `AntennaPattern`, `SurfaceRoughness`, `TiltAngle`, and `Polarization` properties.

Obtain an array of vertical coverage pattern values and angles.

```
freq = 100e6;
ant_height = 10;
rng_fs = 200;
[vcp,vcpangles] = radarvcd(freq,rng_fs,ant_height);
```

To see the vertical coverage pattern, omit the output arguments.

```
radarvcd(freq,rng_fs,ant_height);
```



### Vertical Coverage Pattern with Specified Antenna Pattern

Set the frequency to 100 MHz, the antenna height to 10 m, and the free-space range to 200 km. The antenna pattern is a sinc function with 45° half-power width. The surface height standard deviation is set to  $1/2\sqrt{2}$  m. The antenna tilt angle is set to 0°, and the field polarization is horizontal.

```
pat_angles = linspace(-90,90,361)';
freq = 100e6;

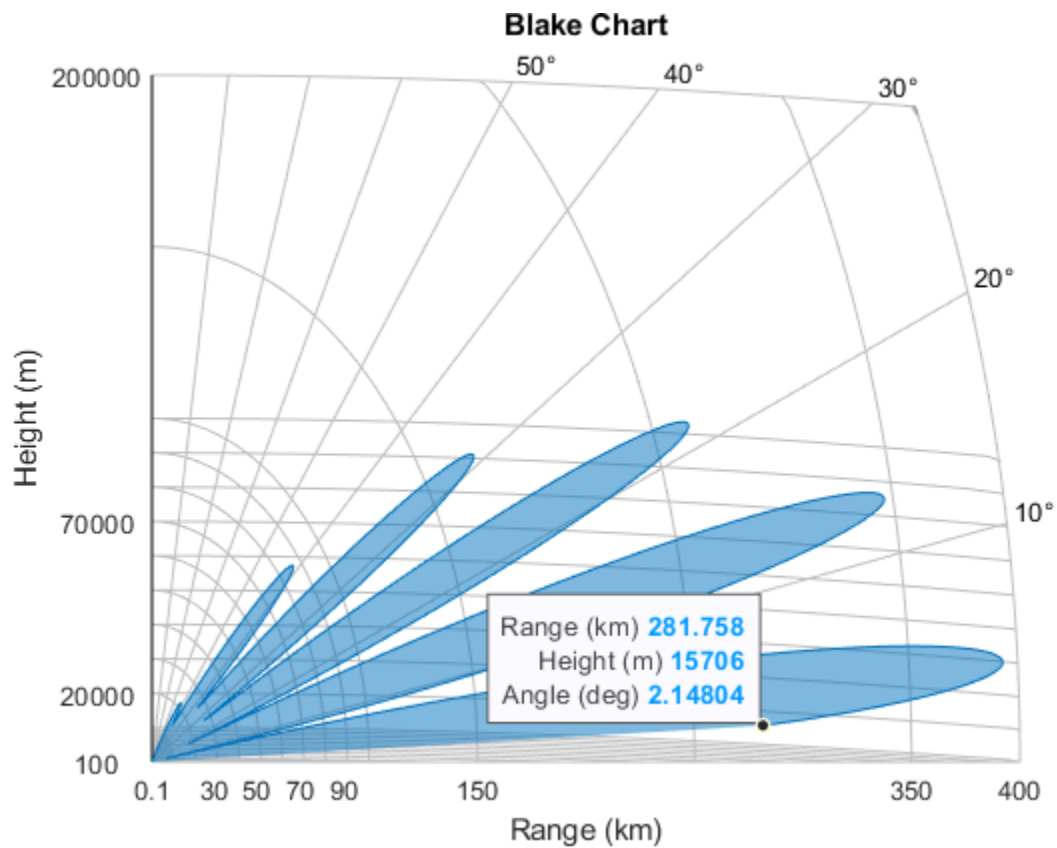
ntn = phased.SincAntennaElement('Beamwidth',45);
pat = ntn(freq,pat_angles');

ant_height = 10;
rng_fs = 200;
tilt_ang = 0;
[vcp,vcpangles] = radarvcd(freq,rng_fs,ant_height,...
    'RangeUnit','km','HeightUnit','m',...
    'AntennaPattern',pat,...
    'PatternAngles',pat_angles,...
    'TiltAngle',tilt_ang,'SurfaceHeightStandardDeviation',1/(2*sqrt(2)));
```

Call `radarvcd` with no output arguments to display the vertical coverage pattern.

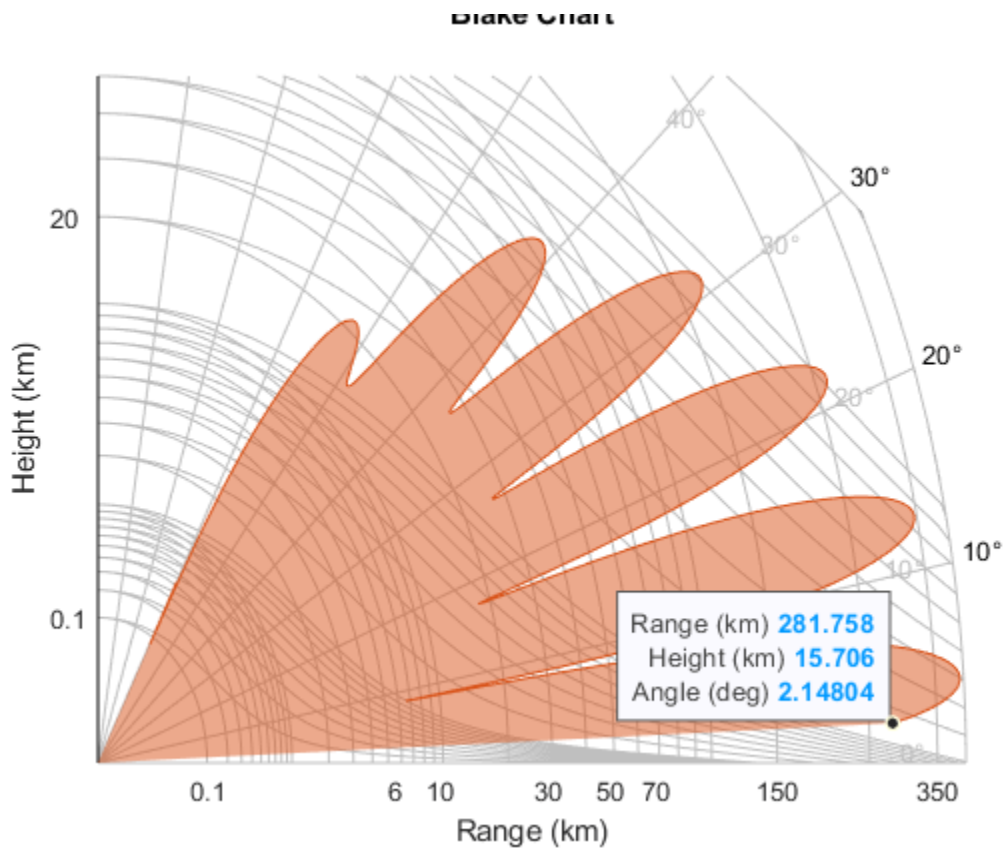
```
radarvcd(freq,rng_fs,ant_height,...
    'RangeUnit','km','HeightUnit','m',...)
```

```
'AntennaPattern',pat,...
'PatternAngles',pat_angles,...
'TiltAngle',tilt_ang,'SurfaceHeightStandardDeviation',1/(2*sqrt(2)));
```



Alternatively, use the `radarvcd` output arguments and the `blakechart` function to display the vertical coverage pattern to a maximum range of 400 km and a maximum height of 50 km. Customize the Blake chart by changing the color.

```
blakechart(vcp,vcpangles,400,50, ...
'FaceColor',[0.8500 0.3250 0.0980],'EdgeColor',[0.8500 0.3250 0.0980])
```



### Plot Vertical Coverage Diagram For User-Specified Antenna

Plot the range-height-angle curve (Blake chart) for a radar with a user-specified antenna pattern.

Define a sinc-function antenna pattern with a half-power beamwidth of 90 degrees. The radar transmits at 100 MHz.

```
pat_angles = linspace(-90,90,361)';
freq = 100e6;

ntn = phased.SincAntennaElement('Beamwidth',90);
pat = ntn(freq,pat_angles');
```

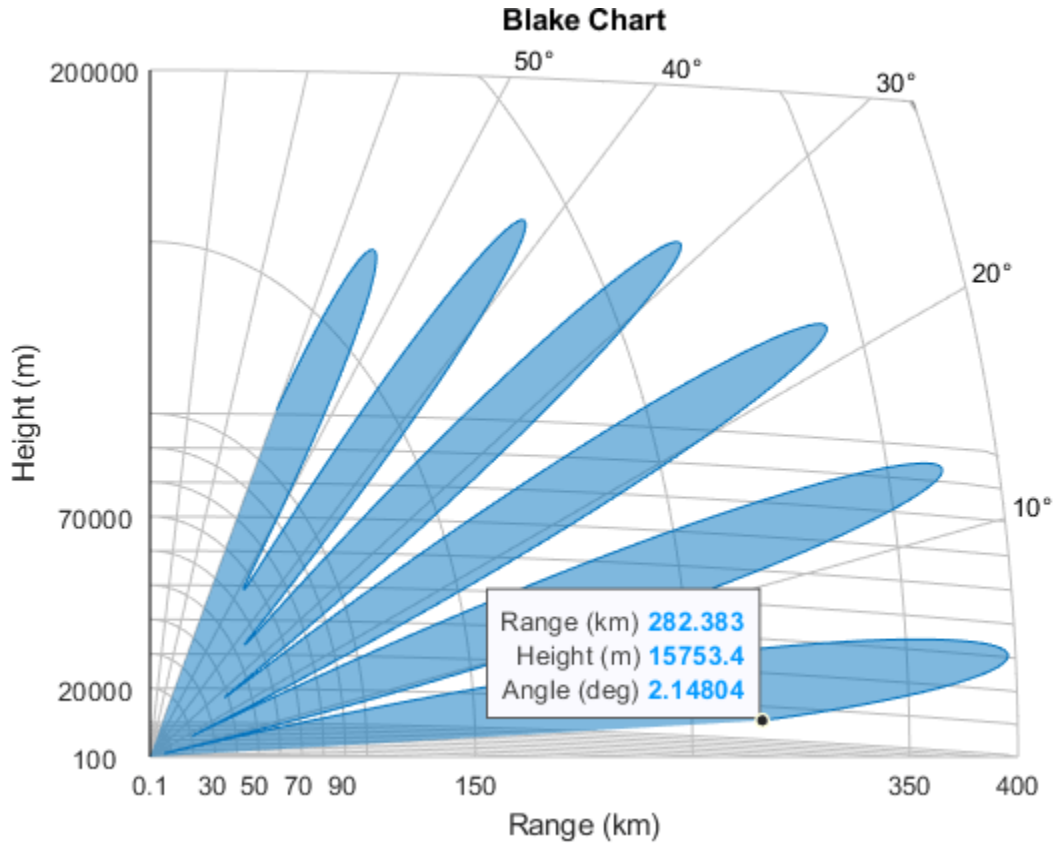
Specify a free-space range of 200 km. The antenna height is 10 meters, the antenna tilt angle is zero degrees, and the surface roughness is one meter.

```
rng_fs = 200;
ant_height = 10;
tilt_ang = 0;
surf_roughness = 1;
```

Create the radar range-height-angle plot.

```
radarvcd(freq,rng_fs,ant_height,...
    'RangeUnit','km','HeightUnit','m',...
```

```
'AntennaPattern', pat, ...
'PatternAngles', pat_angles, ...
'TiltAngle', tilt_ang, ...
'SurfaceHeightStandardDeviation', surf_roughness/(2*sqrt(2));
```



## Input Arguments

### freq — Radar frequency

real-valued scalar less than 10 GHz

Radar frequency, specified as a real-valued scalar less than 10 GHz ( $10^{10}$  Hz).

Example: 100e6

Data Types: double

### rfs — Free-space range

positive scalar | positive vector

Free-space range, specified as a positive scalar or vector. `rfs` is the calculated or assumed free-space range for a target or for a one-way RF system at which the field strength would have a specified value. Range units are set by the `RangeUnit` name-value argument.

Example: 100e3

Data Types: double

**anht — Radar antenna height**

real-valued scalar

Radar antenna height, specified as a real-valued scalar. The height is referenced to the surface. Height units are set by the `HeightUnit` name-value argument.

Example: 10

Data Types: double

**Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: 'HeightUnit', 'km'

**RangeUnit — Radar range units**

'km' (default) | 'nmi' | 'mi' | 'ft' | 'm' | 'kft'

Radar range units denoting kilometers, nautical miles, miles, feet, meters, or kilofeet. This argument specifies the units for the free-space range argument `rfs` and the output vertical coverage pattern `vcp`.

Example: 'mi'

Data Types: char

**HeightUnit — Antenna height units**

'm' (default) | 'nmi' | 'mi' | 'km' | 'ft' | 'kft'

Antenna height units denoting meters, nautical miles, miles, kilometers, feet, or kilofeet. This argument specifies the units for the antenna height `anht` and the `'SurfaceRoughness'` argument.

Example: 'm'

Data Types: char

**Polarization — Transmitted wave polarization**

'H' (default) | 'V'

Transmitted wave polarization, specified as 'H' for horizontal polarization or 'V' for vertical polarization.

Example: 'V'

Data Types: char

**SurfaceRelativePermittivity — Complex permittivity of reflecting surface**

frequency dependent model (default) | complex-valued scalar

Complex permittivity (dielectric constant) of the reflecting surface, specified as a complex-valued scalar. The default value of this argument depends on the value of `freq`. `radarvcd` uses a seawater model that is valid for frequencies up to 10 GHz.

Example: 70

Data Types: double

### SurfaceHeightStandardDeviation — Standard deviation of surface height

0 (default) | real-valued scalar

Standard deviation of surface height, specified as a nonnegative real-valued scalar. A value of 0 indicates a smooth surface. Use 'HeightUnit' to specify the units of height.

The surface height standard deviation relates to the crest-to-trough "surface roughness" height through

$$\text{Surface roughness} = 2 \times \sqrt{2} \times \text{Surface height standard deviation.}$$

Example: 2

Data Types: double

### SurfaceSlope — Surface slope

nonnegative scalar

Surface slope in degrees, specified as a nonnegative scalar. This value is expected to be 1.4 times the RMS surface slope. Given the condition that

$$2 \times \text{GRAZ}/\beta_0 < 1,$$

where GRAZ is the grazing angle of the geometry specified in degrees and  $\beta_0$  is the surface slope, the effective surface height standard deviation in meters is calculated as

$$\text{Effective HGTSD} = \text{HGTSD} \times (2 \times \text{GRAZ}/\beta_0)^{1/5}.$$

This calculation better accounts for shadowing. Otherwise, the effective height standard deviation is equal to HGTSD. This argument defaults to 0, indicating a smooth surface.

Data Types: double

### VegetationType — Vegetation type

'None' (default) | 'Trees' | 'Brush' | 'Weeds' | 'Grass'

Surface vegetation type, specified as 'Trees', 'Weeds', and 'Brush' are assumed to be dense vegetation. 'Grass' is assumed to be thin grass. Use this argument when using the function on surfaces different from the sea.

### ElevationBeamwidth — Half-power elevation beamwidth

10 (default) | scalar between 0° and 90°

Half-power elevation beamwidth in degrees, specified as a scalar between 0° and 90°. The elevation beamwidth is used in the calculation of a sinc antenna pattern. The default antenna pattern is symmetric with respect to the beam maximum and is of the form  $\sin(u)/u$ . The parameter  $u$  is given by  $u = k \sin(\theta)$ , where  $\theta$  is the elevation angle in radians and  $k$  is given by  $k = x_0 / \sin(\pi \times \text{ELBW}/360)$ , where ELBW is the half-power elevation beamwidth and  $x_0 \approx 1.3915573$  is a solution of  $\sin(x) = x/\sqrt{2}$ .

Data Types: double

### AntennaPattern — Antenna elevation pattern

real-valued column vector

Antenna elevation pattern, specified as a real-valued column vector. Values for 'AntennaPattern' must be specified together with values for 'PatternAngles'. Both vectors must have the same size. If both an antenna pattern and an elevation beamwidth are specified, radarvcd uses the antenna pattern and ignores the elevation beamwidth value. This argument defaults to a sinc antenna pattern.

Example: `cosd([-90:90])`



Data Types: double

### **PatternAngles — Antenna pattern elevation angles**

real-valued column vector

Antenna pattern elevation angles specified as a real-valued column vector. The size of the vector specified by `PatternAngles` must be the same as that specified by `AntennaPattern`. Angle units are expressed in degrees and must lie between  $-90^\circ$  and  $90^\circ$ . In general, the antenna pattern should fill the whole range from  $-90^\circ$  to  $90^\circ$  for the coverage to be computed properly.

Example: [-90:90]

Data Types: double

### **TiltAngle — Antenna tilt angle**

0 (default) | real-valued scalar

Antenna tilt angle, specified as a real-valued scalar. The tilt angle is the elevation angle of the antenna with respect to the surface. Angle units are expressed in degrees.

Example: 10

Data Types: double

### **EffectiveEarthRadius — Effective Earth radius**

positive scalar

Effective Earth radius in meters, specified as a positive scalar. The effective Earth radius is an approximation used for modeling refraction effects in the troposphere. The default value calculates the effective Earth radius using a refraction gradient of  $-39e-9$ , which results in approximately  $4/3$  of the real Earth radius.

Data Types: double

### **MaxElevation — Maximum elevation angle**

60 (default) | real-valued scalar

Maximum elevation angle, specified as a real-valued scalar. The maximum elevation angle is the largest angle for which the vertical coverage pattern is calculated. Angle units are expressed in degrees.

Example: 70

Data Types: double

### **MinElevation — Minimum elevation angle**

0 (default) | real-valued scalar

Minimum elevation angle, specified as a real-valued scalar. The minimum elevation angle is the smallest angle for which the vertical coverage pattern is calculated. Angle units are expressed in degrees.

Example: 10

Data Types: double

### **ElevationStepSize — Elevation angle increment**

positive scalar

Elevation angle increment, specified as a positive scalar in degrees. The elevation vector goes from the minimum value specified in `MinElevation` and the maximum value specified in `MaxElevation` in increments of `ElevationStepSize`. The default value of this argument is given by

$$\Delta = 885.6 / (\pi \times f_{\text{MHz}} \times h_{a,\text{ft}}),$$

where  $f_{\text{MHz}}$  is the frequency in MHz and  $h_{a,\text{ft}}$  is the antenna height in feet.

Data Types: `double`

## Output Arguments

### **vcp** — Vertical coverage pattern

real-valued vector | real-valued matrix

Vertical coverage pattern, returned as a real-valued column vector or matrix. The vertical coverage pattern is the actual maximum range of the radar. Each row of the vertical coverage pattern corresponds to one of the angles returned in `vcpangles`. The columns of `vcp` correspond to the ranges specified in `rfs`.

### **vcpangles** — Vertical coverage pattern angles

real-valued vector

Vertical coverage pattern angles, returned as a column vector. The angles range from  $-90^\circ$  to  $90^\circ$ . Each entry of `vcpangles` specifies the elevation angle at which the vertical coverage pattern is measured.

## More About

### Vertical Coverage Pattern

The maximum detection range of a radar antenna can differ depending on placement. Suppose you place a radar antenna near a reflecting surface, such as the earth's land or sea surface and computed maximum detection range. If you then move the same radar antenna to free space far from any boundaries, it results in a different maximum detection range. This is an effect of multipath interference that occurs when waves, reflected from the surface, constructively add to or nullify the direct path signal from the radar to a target. Multipath interference gives rise to a series of lobes in the vertical plane. The vertical coverage pattern is the plot of the actual maximum detection range of the radar versus target elevation and depends upon the maximum free-space detection range and target elevation angle. See Blake [1].

The vertical coverage pattern is generally considered to be valid for antenna heights that are within a few hundred feet of the surface and with targets at altitudes that are not too close to the radar horizon.

## References

[1] Blake, Lamont V. *Machine Plotting of Radar Vertical-Plane Coverage Diagrams*. Naval Research Laboratory Report 7098, 1970.

[2] Barton, David K. *Radar Equations for Modern Radar*. Norwood, MA: Artech House, 2013.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only when output arguments are specified.

## See Also

### Apps

**Radar Designer**

### Functions

[blakechart](#) | [el2height](#) | [height2el](#) | [height2range](#) | [height2grndrange](#) | [landroughness](#) | [range2height](#) | [refractionexp](#) | [searoughness](#)

### Topics

“Radar Vertical Coverage over Terrain”

“Modeling Target Position Errors due to Refraction”

**Introduced in R2021a**

## billingsleyicm

Billingsley's intrinsic clutter motion (ICM) model

### Syntax

```
P = billingsleyicm(fd,fc,wspeed)
P = billingsleyicm(fd,fc,wspeed,c)
```

### Description

`P = billingsleyicm(fd,fc,wspeed)` calculates the clutter Doppler spectrum shape, `P`, due to intrinsic clutter motion (ICM) at Doppler frequencies specified in `fd`. ICM arises when wind blows on vegetation or other clutter sources. This function uses Billingsley's model in the calculation. `fc` is the operating frequency of the system. `wspeed` is the wind speed.

`P = billingsleyicm(fd,fc,wspeed,c)` specifies the propagation speed `c` in meters per second.

### Input Arguments

#### `fd`

Doppler frequencies in hertz. This value can be a scalar or a vector.

#### `fc`

Operating frequency of the system in hertz.

#### `wspeed`

Wind speed in meters per second.

#### `c`

Propagation speed in meters per second.

**Default:** Speed of light

### Output Arguments

#### `P`

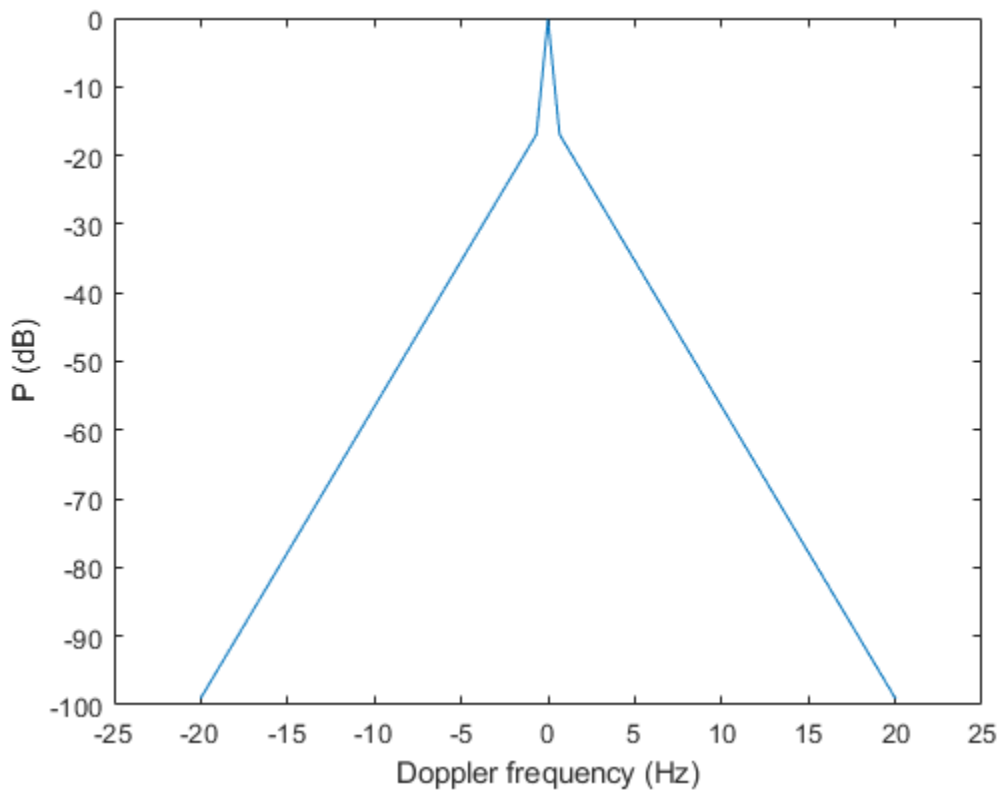
Shape of the clutter Doppler spectrum due to intrinsic clutter motion. The vector size of `P` is the same as that of `fd`.

### Examples

## Compute Billingsley Doppler Spectrum

Calculate and plot the Doppler spectrum shape predicted by the Billingsley ICM model. Assume the PRF is 2 kHz, the operating frequency is 1 GHz, and the wind speed is 5 m/s.

```
v = -3:0.1:3;
fc = 1e9;
wspeed = 5;
c = physconst('LightSpeed');
fd = 2*v/(c/fc);
p = billingsleyicm(fd,fc,wspeed);
plot(fd,pow2db(p))
xlabel('Doppler frequency (Hz)')
ylabel('P (dB)')
```



## References

- [1] Billingsley, J. *Low Angle Radar Clutter*. Norwich, NY: William Andrew Publishing, 2002.
- [2] Long, Maurice W. *Radar Reflectivity of Land and Sea*, 3rd Ed. Boston: Artech House, 2001.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

**Introduced in R2021a**

# depressionang

Depression angle of surface target

## Syntax

```
depAng = depressionang(H,R)
depAng = depressionang(ht,r,MODEL)
depAng = depressionang(H,R,MODEL,Re)
depAng = depressionang( ____, 'TargetHeight', TGTHT)
```

## Description

`depAng = depressionang(H,R)` returns the depression angle from the horizontal at an altitude of R meters to surface targets. The sensor is H meters above the surface. R is the range from the sensor to the surface targets. The computation assumes a curved earth model with an effective earth radius of approximately 4/3 times the actual earth radius.

`depAng = depressionang(ht,r,MODEL)` specifies the earth model used to compute the depression angle. MODEL is either 'Flat' or 'Curved'.

`depAng = depressionang(H,R,MODEL,Re)` specifies the effective earth radius. Effective earth radius applies to a curved earth model. When MODEL is 'Flat', the function ignores Re.

`depAng = depressionang( ____, 'TargetHeight', TGTHT)` specifies the target height, TGTHT above the surface as either a scalar or a vector. If any combination of H, R, and TGTHT are vectors, then the dimensions must be equal. r must be greater than or equal to the absolute value of the difference of ht and TGTHT.

## Input Arguments

### H

Height of the sensor above the surface, in meters. This argument can be a scalar or a vector. If both H and R are nonscalar, they must have the same dimensions.

### R

Distance in meters from the sensor to the surface target. This argument can be a scalar or a vector. If both H and R are nonscalar, they must have the same dimensions. R must be between H and the horizon range determined by TGTHT.

### MODEL

Earth model, as one of | 'Curved' | 'Flat' |.

**Default:** 'Curved'

### Re

Effective earth radius in meters. This argument must be a positive scalar value. You can use `effearthradius` to compute the effective radius.

**Default:** `effearthradius`, which is approximately 4/3 times the actual earth radius

### TGTHT

Target height above surface, specified as a scalar or vector. If any combination of `H`, `R`, and `TGTHT` are vectors, then their sizes must be equal. `R` must be greater than or equal to the absolute value of the difference of `H` and `TGTHT`. A surface target has a `TGTHT` of zero.

**Default:** 0

## Output Arguments

### depAng

Depression angle, in degrees, from the horizontal at the sensor altitude toward surface targets `R` meters from the sensor. The dimensions of `depAng` are the larger of `size(ht)` and `size(r)`.

## Examples

### Compute Depression Angle

Calculate the depression angle for a ground clutter patch that is 1.0 km away from a sensor. The sensor is located on a platform 300 m above the ground.

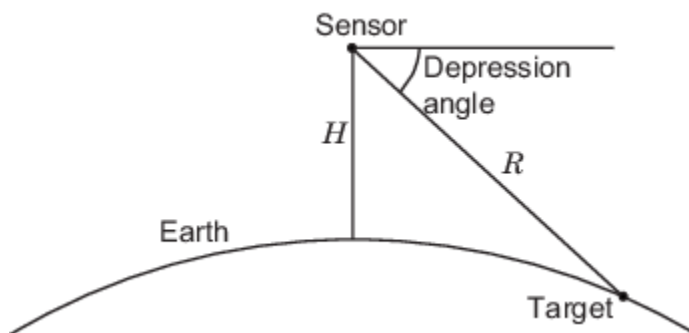
```
depang = depressionang(300,1000)
```

```
depang = 17.4608
```

## More About

### Depression Angle

The depression angle is the angle between a horizontal line containing the sensor and the line from the sensor to a surface target.



For the curved earth model with an effective earth radius of  $R_e$ , the depression angle is:

$$\sin^{-1}\left(\frac{H^2 + 2HR_e + R^2}{2R(H + R_e)}\right)$$



For the flat earth model, the depression angle is:

$$\sin^{-1}\left(\frac{H}{R}\right)$$

## References

- [1] Long, Maurice W. *Radar Reflectivity of Land and Sea*, 3rd Ed. Boston: Artech House, 2001.
- [2] Ward, J. "Space-Time Adaptive Processing for Airborne Radar Data Systems," *Technical Report 1015*, MIT Lincoln Laboratory, December, 1994.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

## See Also

[grazingang](#) | [horizonrange](#) | [effearthradius](#)

**Introduced in R2021a**

## effearthradius

Effective earth radius

### Syntax

Re = effearthradius

Re = effearthradius(refgrad)

Re = effearthradius(R,ha,ht)

Re = effearthradius(R,ha,ht,'SurfaceRefractivity',ns)

Re = effearthradius(R,ha,ht, \_\_\_\_, 'BreakPointAltitude',altbp)

Re = effearthradius(R,ha,ht, \_\_\_\_, 'BreakPointRefractivity',npb)

[Re,k] = effearthradius( \_\_\_\_ )

### Description

Re = effearthradius returns the effective radius Re of a spherical earth computed from the gradient of the index of refraction of the atmosphere. The radius is in meters. This syntax uses the default value of -39e-9 for the gradient, making the effective radius approximately 4/3 of the actual earth radius. For more information about the computation, see “Effective Earth Radius from Refractivity Gradient” on page 1-51.

Re = effearthradius(refgrad) computes the effective radius from the specified gradient of the refractivity, refgrad, of the atmosphere.

Re = effearthradius(R,ha,ht) returns the effective Earth radius, Re, using the average radius of curvature method (see[1]). R is the line-of-sight range to the target. ha is the radar altitude above mean sea level (MSL). ht is the target altitude above MSL. See “Effective Earth Radius from Average Radius of Curvature” on page 1-51.

Re = effearthradius(R,ha,ht,'SurfaceRefractivity',ns) also specifies the scalar surface refractivity, ns for the average radius of curvature method. See “Effective Earth Radius from Average Radius of Curvature” on page 1-51.

Re = effearthradius(R,ha,ht, \_\_\_\_, 'BreakPointAltitude',altbp) also specifies the altitude of the convergence point, altbp, for the average radius of curvature method.

Re = effearthradius(R,ha,ht, \_\_\_\_, 'BreakPointRefractivity',npb) also specifies the refractivity at the convergence point, npb, for the average radius of curvature method.

[Re,k] = effearthradius( \_\_\_\_ ) also outputs the effective radius factor, k. Use this option with any of the syntaxes described above. See “Effective Earth Radius” on page 1-50.

### Examples

#### Default Value of Effective Earth Radius

Return the default effective earth radius due to atmospheric refraction.

```
re = effearthradius
```

```
re = 8.4774e+06
```

Compute the ratio of the effective earth radius to the actual earth radius.

```
r = physconst('EarthRadius');
disp(re/r)
```

```
1.3306
```

### Compute Effective Earth Radius from Refractivity Gradient

Compute the effective earth radius from a specified refractivity gradient,  $-40e-9$ .

```
rgrad = -40e-9;
re = effearthradius(rgrad)
```

```
re = 8.5498e+06
```

### Compute Effective Earth Radius from Path Length

Calculate the effective Earth radii for a radar positioned at sea level aimed at two targets. The first target is at 8000 meters above sea level at a range of 100 km. The second target is at 9000 meters altitude at a range of 200 km.

```
rng = [100e3,200e3];
ha = [0];
ht = [8.0e3, 9.0e3];
re = effearthradius(rng,ha,ht)
```

```
re = 1×2
106 ×
```

```
7.4342 7.3525
```

### Compute Effective Earth Radius from Surface Refractivity

Calculate the effective Earth radii for a radar positioned at sea level and aimed at two targets. The first target is at 8000 meters above sea level at a range of 100 km. The second target is at 9000 meters altitude at a range of 200 km. Specify the surface refractivity as 100.0 N-units.

```
rng = [100e3,200e3];
ha = [0,0];
ht = [8.0e3,9.0e3];
re = effearthradius(rng,ha,ht,'SurfaceRefractivity',100)
```

```
re = 1×2
106 ×
```

6.3582    6.3582

### Compute Effective Earth Radius Using Breakpoint Height

Calculate the effective Earth radii for a radar positioned at sea level aimed at two targets. The first target is at 8000 meters above sea level at a range of 100 km. The second target is at 9000 meters altitude at a range of 200 km. The breakpoint altitude is 10000.0 meters and the surface refractivity is 350 N-units.

```
rng = [100e3,200e3];  
ha = [0,0];  
ht = [8.0e3,9.0e3];  
re = effearthradius(rng,ha,ht,'SurfaceRefractivity',350.0, ...  
    'BreakPointAltitude',10000.0)
```

```
re = 1×2  
106 ×
```

7.5877    7.4917

### Compute Effective Earth Radius Using Breakpoint Refractivity and Height

Calculate the effective Earth radii for a radar positioned at sea level and aimed at two targets. The first target is at 8000 meters above sea level at a range of 100 km. The second target is at 9000 meters altitude at a range of 200 km. The breakpoint altitude is 10000.0 meters, the breakpoint refractivity is 300 N-units, and the surface refractivity is 375 N-units.

```
rng = [100e3,200e3];  
ha = 0;  
ht = [8.0e3, 9.0e3];  
re = effearthradius(rng,ha,ht,'SurfaceRefractivity',375, ...  
    'BreakPointAltitude',10e3,'BreakPointRefractivity',300)
```

```
re = 1×2  
106 ×
```

6.6962    6.6930

### Compute Effective Earth Radius Factor

Calculate the effective Earth radius factors for a radar positioned at sea level aimed at two targets. The first target is at 8000 meters above sea level at a range of 100 km. The second target is at 9000 meters altitude at a range of 200 km. The breakpoint altitude is one kilometer, the breakpoint refractivity is 300.0 N-units, and the surface refractivity is 350.0 N-units.

```

rng = [100e3,200e3];
ha = [0,0];
ht = [8.0e3,9.0e3];
[re,k] = effearthradius(rng,ha,ht,'SurfaceRefractivity',350.0, ...
    'BreakPointAltitude',1000.0,'BreakPointRefractivity',300.0)

re = 1×2
106 ×

    7.7113    7.5724

k = 1×2

    1.2104    1.1886

```

## Input Arguments

### refgrad — Refractivity gradient

-39e-9 (default) | scalar

Refractivity gradient, specified as a scalar. Units are in N-units/meter.

Data Types: double

### R — Line-of-sight range to target

positive scalar | 1-by-*M* vector of positive values

Line-of-sight range to the target from the radar, specified as a positive scalar or a 1-by-*M* vector of positive values. *M* must be the same for R, ha, and ht. However, if one of R, ha, and ht is a scalar and another is a 1-by-*M* vector, the scalar is expanded into a 1-by-*M* vector. Units are in meters.

Data Types: double

### ha — Radar altitude above mean sea level

scalar | 1-by-*M* vector

Radar altitude above mean sea level, specified as a scalar or a 1-by-*M* vector. *M* must be the same for R, ha, and ht. However, if one of R, ha, and ht is a scalar and another is a 1-by-*M* vector, the scalar is expanded into a 1-by-*M* vector. Units are in meters.

Data Types: double

### ht — Target altitude above mean sea level

scalar | *M*-length vector

Target altitude above mean sea level, specified as a scalar or an *M*-length vector. *M* must be the same for R, ha, and ht. However, if one of R, ha, and ht is a scalar and another is a 1-by-*M* vector, the scalar is expanded into a 1-by-*M* vector. Units are in meters.

Data Types: double

### ns — Scalar surface refractivity

313 (default) | positive scalar

Scalar surface refractivity, specified as a positive scalar. Units are N-units.

**Dependencies**

To enable this argument, use the syntax specifying 'SurfaceRefractivity'.

Data Types: double

**altbp — Convergence point altitude**

12192 or 9144 (default) | scalar

Convergence point altitude, specified as a scalar. The convergence point altitude defaults to 12192 meters when any of the input altitudes specified in `ha` or `ht` are greater than 9144 meters. Otherwise, it defaults to 9144 meters. Setting the 'BreakPointAltitude' and 'BreakPointRefractivity' values can be used to tune the output to measured refraction values. For more information, see “Effective Earth Radius from Average Radius of Curvature” on page 1-51. Units are in meters.

**Dependencies**

To enable this argument, use the syntax specifying 'BreakPointAltitude'.

Data Types: double

**npb — Convergence point refractivity**

66.65 or 102.9 (default) | scalar

Convergence point refractivity, specified as a scalar. The refractivity defaults to 66.65 N-units when any of the input altitudes specified in `ha` or `ht` are greater than 9144 meters. Otherwise, the default is 102.9. Setting the 'BreakPointAltitude' and 'BreakPointRefractivity' values can be used to tune the output to measured refraction values. For more information, see “Effective Earth Radius from Average Radius of Curvature” on page 1-51. Units are N-units.

**Dependencies**

To enable this argument, use the syntax specifying 'BreakPointRefractivity'.

Data Types: double

**Output Arguments****Re — Effective earth radius**

4/3 actual earth radius (default) | positive scalar

Effective earth radius, returned as a positive scalar. Units are in meters.

**k — Effective earth radius factor**

4/3 (default) | positive scalar

Effective earth radius factor, returned as a positive scalar. The effective earth radius factor is the ratio of the effective earth radius to the physical earth radius. Units are dimensionless.

Data Types: double

**More About****Effective Earth Radius**

The effective earth radius method is an approximation used for modelling refraction effects in the troposphere. Changing the radius of the earth can account for refraction effects. The effective radius

method ignores other types of propagation phenomena such as ducting. A related quantity, the effective earth radius factor, is the ratio of the effective earth radius to the actual earth radius.

$$k = \frac{R_e}{r}$$

where  $r$  is the actual earth radius and  $R_e$  is the effective earth radius. Commonly, the effective earth radius factor,  $k$ , is chosen as 4/3. However, at long ranges and with shallow angles,  $k$  can deviate greatly from the 4/3. (With no atmospheric refraction,  $k = 1$ . An infinite value for  $k$  represents a flat Earth). The `effearthradius` function provides two methods for calculating the effective earth radius: the refractivity gradient method and the average radius of curvature method.

### Effective Earth Radius from Refractivity Gradient

An estimate of the effective earth radius factor,  $k$ , can be derived from the refractivity gradient using

$$k = \frac{1}{1 + r \cdot \text{refgrad}}$$

where  $r$  is the actual earth radius in meters. `refgrad` is the gradient of the index of refraction specified by the `refgrad` argument. The index of refraction for a given altitude is the ratio of the free-space propagation speed of electromagnetic waves to the propagation speed in air at that altitude. The gradient is the rate of change of the index of refraction with altitude. The value of 4/3 corresponds to an index of refraction gradient of  $-39 \times 10^{-9} \text{ m}^{-1}$ .

### Effective Earth Radius from Average Radius of Curvature

Another way of estimating the effective earth radius factor is by using the average radius of curvature method described in [1]. The first step in the method is to compute the average radius of curvature over the signal propagation path

$$\rho_{avg} = \frac{1}{h_a - h_t} \int_{h_t}^{h_a} \rho dh = \frac{H_b}{10^{-6} N_s \cos \psi_g} \frac{e^{\left(\frac{h_a - h_t}{H_b}\right)} - 1}{\frac{h_a - h_t}{H_b}}$$

where the integral spans the range from the radar altitude ( $h_a$ ) to the target altitude ( $h_t$ ).

The constants in the equation where

- $h_t$  is the altitude of the target, specified by the `ht` argument.
- $h_a$  is the altitude of the radar, specified by the `ha` argument.
- $h_b$  is the altitude of the convergence point or breakpoint, specified by the `altbp` argument.
- $N_b$  is the refractivity measure (in N-units) at the convergence point or breakpoint specified by the `npb` argument.
- $N_s$  is the refractivity measure (in N-units) at the surface.

Altitudes are with respect to mean sea level. The constant  $H_b$  is computed from

$$H_b = \frac{h_b - h_t}{\ln \frac{N_t}{N_b}}$$

Then, the effective earth radius factor is computed from the average radius of curvature using

$$k = \frac{1}{1 - \frac{R_e}{\rho_{avg}}}$$

### **Refractivity Measure and N-Units**

The refractivity measure,  $N$ , is related to the index of refraction,  $n$  by:

$$n = 1 + 10^{-6}N$$

$10^{-6}N$  represents the deviation of the index of refraction from the index of refraction of free space.  $N$  is expressed in N-units.

### **References**

- [1] Doerry, Armin. W. "*Earth Curvature and Atmospheric Refraction Effects on Radar Signal Propagation*", Sandia National Laboratories, SAND2012-10690, January 2013.
- [2] Long, Maurice W. *Radar Reflectivity of Land and Sea*, 2nd Ed. Artech House, 2001.
- [3] Mahafza, Bassem R. *Radar Signal Analysis and Processing Using MATLAB*, CRC Press, 2009.
- [4] Skolnik, Merrill I. *Introduction to Radar Systems*, Third edition, McGraw-Hill, 2001.
- [5] Ward, James. "*Space-Time Adaptive Processing for Airborne Radar*", Lincoln Lab Technical Report, 1994.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

### **See Also**

depressionang | grazingang

#### **Topics**

"Radar Vertical Coverage over Terrain"

**Introduced in R2021a**



# earthSurfacePermittivity

Permittivity and conductivity of earth surface materials

## Syntax

```
[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('pure-water', fc, temp)
```

```
[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('dry-ice', fc, temp)
```

```
[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('sea-water', fc, temp, salinity)
```

```
[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('wet-ice', fc, liqfrac)
```

```
[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('soil', fc, temp, sandpercent, claypercent, specificgravity, vwc)
```

```
[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('soil', ___, bulkdensity)
```

```
[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('vegetation', fc, temp, gwc)
```

## Description

The `earthSurfacePermittivity` function computes electrical characteristics (relative permittivity, conductivity, and complex relative permittivity) of earth surface materials based on the methods and equations presented in ITU-R P.527 [1]. The `earthSurfacePermittivity` function provides various syntaxes to account for characteristics germane to the specified surface material.

`[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('pure-water', fc, temp)` calculates the electrical characteristics for pure water at the specified frequency and temperature. For pure-water, the temperature setting must be greater than 0 °C.

`[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('dry-ice', fc, temp)` calculates the electrical characteristics for dry-ice at the specified frequency and temperature. For dry-ice, the temperature must be less than or equal to 0 °C.

`[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('sea-water', fc, temp, salinity)` calculates the electrical characteristics for sea water at the specified frequency, temperature, and salinity. For sea-water, the temperature must be greater than -2 °C.

`[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('wet-ice', fc, liqfrac)` calculates the electrical characteristics for wet ice at the specified frequency, and liquid water volume fraction. For wet-ice, the temperature is 0 °C.

`[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('soil', fc, temp, sandpercent, claypercent, specificgravity, vwc)` calculates the electrical characteristics for soil at the specified frequency, temperature, sand percentage, clay percentage, specific gravity, and volumetric water content.

`[epsilon, sigma, complexepsilon] = earthSurfacePermittivity('soil', ___, bulkdensity)` sets the soil bulk density in addition to input arguments from the previous syntax.

[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('vegetation',fc,temp,gwc) calculates the electrical characteristics for vegetation at the specified frequency, temperature, and gravimetric water content. For vegetation, the temperature must be greater than or equal to -20 °C.

## Examples

### Compare Permittivity and Conductivity of Salt-free Sea Water to Pure Water

Compare the relative permittivity and conductivity for salt-free (zero-salinity) sea water to pure water.

Specify a carrier frequency of 9 GHz, temperature of 30°C, and salinity of zero.

```
fc = 9e9; % Carrier frequency in Hz.  
temp = 30;  
salinity = 0;
```

Compute the relative permittivity and conductivity.

```
[epsilon_pure_water,sigma_pure_water] = earthSurfacePermittivity('pure-water',fc,temp);  
[epsilon_sea_water,sigma_sea_water] = earthSurfacePermittivity('sea-water',fc,temp,salinity);
```

Confirm that salt-free sea water and pure water have equal relative permittivity and conductivity.

```
isequal(epsilon_pure_water,epsilon_sea_water)
```

```
ans = logical  
     1
```

```
isequal(sigma_pure_water,sigma_sea_water)
```

```
ans = logical  
     1
```

### Compare Permittivity and Conductivity of Wet Ice to Dry Ice

Compare the relative permittivity and conductivity for wet ice with no liquid water to dry ice at 0°C. Confirm the results differ by a negligible amount.

Specify a carrier frequency of 12 GHz.

```
fc = 12e9; % Carrier frequency in Hz.
```

Calculate the relative permittivity and conductivity for wet ice with zero liquid water by volume.

```
liqfrac = 0;  
[epsilon_wet_ice_0,sigma_wet_ice_0] = earthSurfacePermittivity('wet-ice',fc,liqfrac); % Set liquid water fraction to zero
```

Calculate the relative permittivity and conductivity for dry ice at 0 °C.

```
temp = 0;
[epsilon_dry_ice_0, sigma_dry_ice_0] = earthSurfacePermittivity('dry-ice', fc, temp); % Set tempera
```

Compare the relative permittivity and conductivity for wet ice with no liquid to dry ice at 0°C. Confirm that wet ice with no liquid and dry ice at 0°C have essentially equal relative permittivity and conductivity.

```
epsilon_wet_ice_0-epsilon_dry_ice_0
```

```
ans = 8.8818e-16
```

```
sigma_wet_ice_0-sigma_dry_ice_0
```

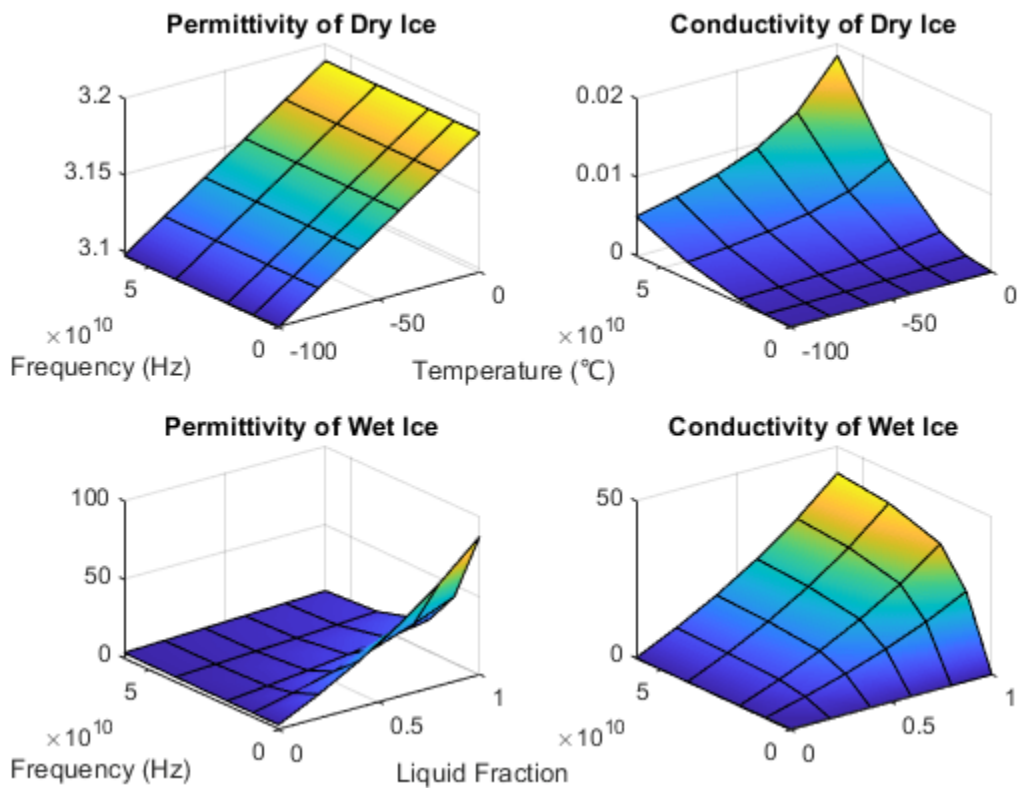
```
ans = -9.2179e-16
```

Plot permittivity and conductivity versus frequency for dry ice and for wet ice. For dry ice, vary the temperature. For wet ice, vary the liquid water volume fraction. Calculate the permittivity and conductivity values by using `arrayfun` to apply the `earthSurfacePermittivity` function to the elements of the arrayed inputs.

```
freq = repmat([0.1,10,20,40,60]*1e9,6,1);
temp = repmat((-100:20:0)',1,5);
liqfrac = repmat((0:0.2:1)',1,5);
[epsilon_dry_ice, sigma_dry_ice] = arrayfun(@(x,y)earthSurfacePermittivity('dry-ice',x,y),freq,temp);
[epsilon_wet_ice, sigma_wet_ice] = arrayfun(@(x,y)earthSurfacePermittivity('wet-ice',x,y),freq,liqfrac);
```

Display tiled surface plots across specified ranges.

```
figure
tiledlayout(2,2)
nexttile
surf(temp,freq,epsilon_dry_ice,'FaceColor','interp')
title('Permittivity of Dry Ice')
xlabel('Temperature (°C)')
ylabel('Frequency (Hz)')
nexttile
surf(temp,freq,sigma_dry_ice,'FaceColor','interp')
title('Conductivity of Dry Ice')
nexttile
surf(liqfrac,freq,epsilon_wet_ice,'FaceColor','interp')
title('Permittivity of Wet Ice')
xlabel('Liquid Fraction')
ylabel('Frequency (Hz)')
nexttile
surf(liqfrac,freq,sigma_wet_ice,'FaceColor','interp')
title('Conductivity of Wet Ice')
```



### Calculate Permittivity and Conductivity of Various Soil Mixtures

Calculate relative permittivity and conductivity for various soil mixtures as defined by textual classifications in ITU-R P.527, Table 1.

Initialize computation variables for constant values and arrayed values.

```
fc = 28e9; % Frequency in Hz
temp = 23; % Temperature in  $^{\circ}\text{C}$ 
vwc = 0.5; % Volumetric water content
pSand = [51.52; 41.96; 30.63; 5.02]; % Sand percentage
pClay = [13.42; 8.53; 13.48; 47.38]; % Clay percentage
sg = [2.66; 2.70; 2.59; 2.56]; % Specific gravity
bd = [1.6006; 1.5781; 1.5750; 1.4758]; % Bulk density ( $\text{g}/\text{cm}^3$ )
```

Calculate the relative permittivity and conductivity for these textual classifications: sandy loam, loam, silty loam, and silty clay. Use `arrayfun` to apply the `earthSurfacePermittivity` function to the elements of the arrayed inputs. Tabulate the results.

```
[Permittivity,Conductivity] = arrayfun(@(w,x,y,z)earthSurfacePermittivity( ...
    'soil',fc,temp,w,x,y,vwc,z),pSand,pClay,sg,bd);
```

```
pSilt = 100 - (pSand + pClay); % Silt percentage
soilType = ["Sandy Loam";"Loam";"Silty Loam";"Silty Clay"];
```

```
varNames1 = ["Soil Textual Classification";"Sand";"Clay";"Silt";"Specific Gravity";"Bulk Density"];
varNames2 = ["Soil Textual Classification";"Permittivity";"Conductivity"];
```

ITU-R P.527, Table 1 specifies the sand percentage, clay percentage, specific gravity, and bulk density for soil mixtures with these soil textual classifications.

```
table(soilType,pSand,pClay,pSilt,sg,bd,'VariableNames',varNames1)
```

ans=4×6 table

Soil Textual Classification	Sand	Clay	Silt	Specific Gravity	Bulk Density
"Sandy Loam"	51.52	13.42	35.06	2.66	1.6006
"Loam"	41.96	8.53	49.51	2.7	1.5781
"Silty Loam"	30.63	13.48	55.89	2.59	1.575
"Silty Clay"	5.02	47.38	47.6	2.56	1.4758

The relative permittivity and conductivity for these soil textual classifications are included in this table.

```
table(soilType,Permittivity,Conductivity,'VariableNames',varNames2)
```

ans=4×3 table

Soil Textual Classification	Permittivity	Conductivity
"Sandy Loam"	15.281	18.2
"Loam"	14.563	16.998
"Silty Loam"	13.965	16.011
"Silty Clay"	12.861	14.647

### Calculate Permittivity and Conductivity of Vegetation

Calculate relative permittivity and conductivity versus frequency for vegetation, varying gravimetric water content and temperature.

Calculate relative permittivity and conductivity for vegetation at specified settings.

```
fc = 10e9; % Frequency in Hz
temp = 23; % Temperature in °C
gwc = 0.68; % Gravimetric water content
[epsilon_veg,sigma_veg] = ...
    earthSurfacePermittivity('vegetation',fc,temp,gwc)
```

```
epsilon_veg = 20.5757
```

```
sigma_veg = 4.9320
```

Calculate values necessary to plot permittivity and conductivity by using `arrayfun` to apply the `earthSurfacePermittivity` function to the elements of the arrayed inputs.

For a range of temperatures, calculate values to plot permittivity and conductivity versus frequency for vegetation at a 0.68 gravimetric water content.

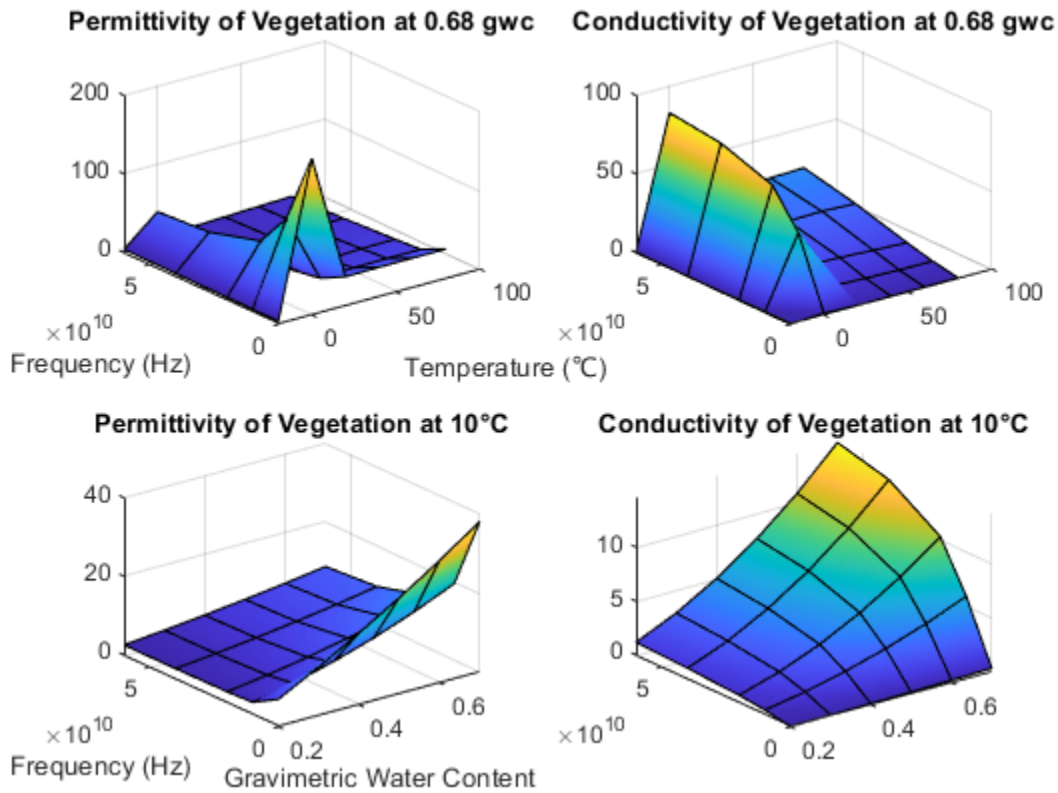
```
fc = repmat([0.1,10,20,40,60]*1e9,6,1);
gwc1 = 0.68;
temp1 = repmat((-20:20:80)',1,5);
[epsilon_veg_gwc, sigma_veg_gwc] = ...
    arrayfun(@(x,y)earthSurfacePermittivity('vegetation',x,y,gwc1),fc,temp1);
```

For a range of gravimetric water contents, calculate values to plot permittivity and conductivity versus frequency for vegetation at 10°C.

```
temp2 = 10;
gwc2 = repmat((0.2:0.1:0.7)',1,5);
[epsilon_veg_tmp, sigma_veg_tmp] = ...
    arrayfun(@(x,z)earthSurfacePermittivity('vegetation',x,temp2,z),fc,gwc2);
```

Display tiled surface plots across specified ranges.

```
figure
tiledlayout(2,2)
nexttile
surf(temp1,fc,epsilon_veg_gwc,'FaceColor','interp')
title('Permittivity of Vegetation at 0.68 gwc')
xlabel('Temperature (°C)')
ylabel('Frequency (Hz)')
nexttile
surf(temp1,fc,sigma_veg_gwc,'FaceColor','interp')
title('Conductivity of Vegetation at 0.68 gwc')
nexttile
surf(gwc2,fc,epsilon_veg_tmp,'FaceColor','interp')
title('Permittivity of Vegetation at 10°C')
xlabel('Gravimetric Water Content')
ylabel('Frequency (Hz)')
nexttile
surf(gwc2,fc,sigma_veg_tmp,'FaceColor','interp')
title('Conductivity of Vegetation at 10°C')
```



## Input Arguments

### fc – Carrier frequency

scalar in the range (0, 1e12]

Carrier frequency in Hz, specified as a scalar in the range (0, 1e12].

Data Types: double

### temp – Temperature

numeric scalar

Temperature in °C, specified as a numeric scalar. Valid surfaces and associated temperature limits are indicated in this table.

Surface	Valid Temperature (°C)
pure-water	greater than 0
dry-ice	less than or equal to 0
sea-water	greater than or equal to -2
soil	any numeric
vegetation	≥ -20

---

**Note** When the surface is wet-ice, the temperature is 0 °C.

---

Data Types: double

**salinity — Salinity of sea water**

nonnegative scalar

Salinity of the sea water in g/Kg, specified as a nonnegative scalar.

Data Types: double

**liqfrac — Liquid water volume fraction of wet ice**

numeric scalar in the range [0, 1]

Liquid water volume fraction of the wet ice, specified as a numeric scalar in the range [0, 1].

Data Types: double

**sandpercent — Sand percentage of soil**

numeric scalar in the range [0, 100]

Sand percentage of the soil, specified as a numeric scalar in the range [0, 100]. The sum of sandpercent and claypercent must be less than or equal to 100.

Data Types: double

**claypercent — Clay percentage of soil**

numeric scalar in the range [0, 100]

Clay percentage of the soil, specified as a numeric scalar in the range [0, 100]. The sum of sandpercent and claypercent must be less than or equal to 100.

Data Types: double

**specificgravity — Specific gravity of soil**

nonnegative scalar

Specific gravity of the soil, specified as a nonnegative scalar. The specific gravity is the mass density of the soil sample divided by the mass density of the amount of water in the soil sample.

Data Types: double

**vwc — Volumetric water content of soil**

numeric scalar in the range [0, 1]

Volumetric water content of the soil, specified as a numeric scalar in the range [0, 1]. For more information, see “Soil Water Content” on page 1-62.

Data Types: double

**bulkdensity — Bulk density of soil**

nonnegative scalar

Bulk density, in g/cm<sup>3</sup>, of the soil, specified as a nonnegative scalar. For more information, see “Soil Water Content” on page 1-62.

Data Types: double



**gwc — Gravimetric water content of vegetation**

numeric scalar in the range [0, 0.7]

Gravimetric water content of the vegetation, specified as a numeric scalar in the range [0, 0.7]. For more information, see “Soil Water Content” on page 1-62.

Data Types: double

**Output Arguments****epsilon — Relative permittivity**

nonnegative scalar

Relative permittivity of the earth surface, returned as a nonnegative scalar.

**sigma — Conductivity**

nonnegative scalar

Conductivity of the earth surface in Siemens per meter (S/m), returned as a nonnegative scalar.

**complexepsilon — Complex relative permittivity**

complex scalar

Complex relative permittivity of the earth surface, returned as a complex scalar calculated as

$$\text{complexepsilon} = \text{epsilon} - 1i \text{sigma} / (2\pi f c \epsilon_0).$$

The computation of `complexepsilon` is based on Equations (59) and (9b) in ITU-R P.527 [1].  $f$  is the frequency in GHz.  $c$  is the velocity of light in free space.  $\epsilon_0 = 8.854187817 \times 10^{-12}$  Farads/m, where  $\epsilon_0$  is the electric constant for the permittivity of free space.

**More About****ITU Terrain Materials**

ITU-R P.527 [1] presents methods and equations to calculate complex relative permittivity at carrier frequencies up to 1,000 GHz for these common earth surface materials.

- Water
- Sea Water
- Dry or Wet Ice
- Dry or Wet Soil (combination of sand, clay, and silt)
- Vegetation (above and below freezing)

As described in ITU-R P.527, specific textural classification applies to these mixtures of sand, clay, and silt in soil with associated specific gravities and bulk densities.

Soil Designation Textural Class	Sandy Loam	Loam	Silty Loam	Silty Clay
% Sand	51.52	41.96	30.63	5.02
% Clay	13.42	8.53	13.48	47.38
% Silt	35.06	49.51	55.89	47.60

Soil Designation Textural Class	Sandy Loam	Loam	Silty Loam	Silty Clay
Specific gravity ( $\rho_s$ )	2.66	2.70	2.59	2.56
Bulk Density ( $\rho_b$ ) in g/cm <sup>3</sup>	1.6006	1.5781	1.5750	1.4758

### Soil Water Content

Soil water content is expressed on a gravimetric or volumetric basis. Gravimetric water content, *gwc*, is the mass of water per mass of dry soil. Volumetric water content, *vwc*, is the volume of liquid water per volume of soil. The bulk density, *bulkdensity*, is the ratio of the dry soil weight to the volume of the soil sample. The relationship between *gwc* and *vwc* is  $vwc = gwc \div bulkdensity$ . When bulk density is not specified, the value of *bulkdensity* is computed by using ITU-R P.527, Equation 36:

$$bulkdensity = 1.07256 + 0.078886 \ln(pSand) + 0.038753 \ln(pClay) + 0.032732 \ln(pSilt),$$

where

- *pSand* = sandpercent
- *pClay* = claypercent
- *pSilt* = 100 - (sandpercent + claypercent)

### References

- [1] International Telecommunications Union Radiocommunication Sector. *Electrical characteristics of the surface of the Earth*. Recommendation P.527-5. ITU-R, approved August 14, 2019. <https://www.itu.int/rec/R-REC-P.527-5-201908-I/en>.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

#### Functions

#### Objects

#### Introduced in R2020a

# grazingang

Grazing angle of surface target

## Syntax

```
grazAng = grazingang(H,R)
grazAng = grazingang(H,R,MODEL)
grazAng = grazingang(H,R,MODEL,Re)
grazAng = grazAng = grazingang( ____, 'TargetHeight', TGTHT)
```

## Description

`grazAng = grazingang(H,R)` returns the grazing angle for a sensor H meters above the surface, to surface targets R meters away. The computation assumes a curved earth model with an effective earth radius of approximately 4/3 times the actual earth radius.

`grazAng = grazingang(H,R,MODEL)` also specifies the earth model used to compute the grazing angle. MODEL is either 'Flat' or 'Curved'.

`grazAng = grazingang(H,R,MODEL,Re)` also specifies the effective earth radius. Effective earth radius applies to a curved earth model. When MODEL is 'Flat', the function ignores Re.

`grazAng = grazAng = grazingang( ____, 'TargetHeight', TGTHT)` also specifies the target height, TGTHT above the surface as either a scalar or a vector. If any combination of ht, R, and TGTHT are vectors, then the dimensions must be equal. R must be greater than or equal to the absolute value of the difference of HT and TGTHT.

## Input Arguments

### H

Height of the sensor above the surface, in meters. This argument can be a scalar or a vector. If both H and R are nonscalar, they must have the same dimensions.

### R

Distance in meters from the sensor to the surface target. This argument can be a scalar or a vector. If both H and R are nonscalar, they must have the same dimensions. R must be between H and the horizon range determined by TGTHT.

### MODEL

Earth model, as one of | 'Curved' | 'Flat' |.

**Default:** 'Curved'

### Re

Effective earth radius in meters. This argument must be a positive scalar value. You can use `effearthradius` to compute the effective radius.

**Default:** `effearthradius`, which is approximately 4/3 times the actual earth radius

### TGTHT

Target height above surface, specified as a scalar or vector. If any combination of `H`, `R`, and `TGTHT` are vectors, then their sizes must be equal. `R` must be greater than or equal to the absolute value of the difference of `H` and `TGTHT`. A surface target has a `TGTHT` of zero.

**Default:** 0

## Output Arguments

### grazAng

Grazing angle, in degrees. The size of `grazAng` is the larger of `size(H)` and `size(R)`.

## Examples

### Compute Grazing Angle

Determine the grazing angle (in degrees) of a path to a ground target located 1.0 km from a sensor. The sensor is mounted on a platform that is 300 m above the ground.

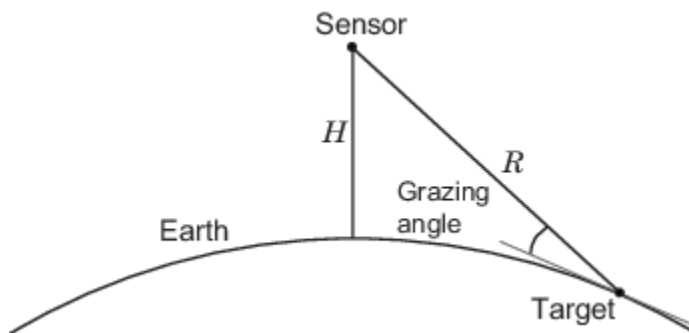
```
grazAng = grazingang(300,1.0e3)
```

```
grazAng = 17.4544
```

## More About

### Grazing Angle

The grazing angle is the angle between a line from the sensor to a surface target, and a tangent to the earth at the site of that target.



For the curved earth model with an effective earth radius of  $R_e$ , the grazing angle is:

$$\sin^{-1}\left(\frac{H^2 + 2HR_e - R^2}{2RR_e}\right)$$

For the flat earth model, the grazing angle is:

$$\sin^{-1}\left(\frac{H}{R}\right)$$

## References

- [1] Long, Maurice W. *Radar Reflectivity of Land and Sea*, 3rd Ed. Boston: Artech House, 2001.
- [2] Ward, J. "Space-Time Adaptive Processing for Airborne Radar Data Systems," *Technical Report 1015*, MIT Lincoln Laboratory, December, 1994.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

### See Also

[horizonrange](#) | [depressionang](#) | [effearthradius](#)

**Introduced in R2021a**

## height2grndrange

Convert target height to ground range

### Syntax

```
gr = height2grndrange(tgtht,anht,el)
gr = height2grndrange(tgtht,anht,el,Name=Value)
```

### Description

`gr = height2grndrange(tgtht,anht,el)` returns the ground range to the target, `gr`, as a function of the target height `tgtht`, the sensor height `anht`, and the local elevation angle `el` assuming a “Curved Earth Model” on page 1-69 with a 4/3 effective Earth radius.

`gr = height2grndrange(tgtht,anht,el,Name=Value)` specifies additional inputs using name-value arguments. For example, you can specify a flat Earth model, a curved Earth model with a given radius, or a “CRPL Exponential Reference Atmosphere Model” on page 1-70 with custom values.

### Examples

#### Ground Range Along Propagated Path

Compute the range along the propagated path for a target height of 1 km, an antenna height of 10 meters, and an elevation angle of 2 degrees at the radar. Assume a curved Earth model with a 4/3 effective Earth radius.

```
r = height2grndrange(1e3,10,2)
r = 2.7106e+04
```

#### Ground Range Using CRPL Atmosphere

Compute the range along the propagated path using the CRPL exponential reference atmosphere. Assume a target height of 1 km, an antenna height of 10 meters, and an elevation angle of 2 degrees at the radar.

```
gr = height2grndrange(1e3,10,2,Method="CRPL")
gr = 2.7143e+04
```

### Input Arguments

#### **tgtht** — Target height

nonnegative real-valued scalar | nonnegative real-valued vector

Target height in meters, specified as a nonnegative real-valued scalar or vector. If `tgtht` is a vector, it must have the same size as the other vector input arguments of `height2grndrange`. Heights are referenced to the ground.

Data Types: `double`

### **anht — Sensor height**

nonnegative real-valued scalar | nonnegative real-valued vector

Sensor height in meters, specified as a nonnegative real-valued scalar or vector. If `anht` is a vector, it must have the same size as the other vector input arguments of `height2grndrange`. Heights are referenced to the ground.

Data Types: `double`

### **e1 — Local elevation angle**

real-valued scalar | real-valued vector

Local elevation angle in degrees, specified as a real-valued scalar or vector. The local elevation angle is the initial elevation angle of the ray leaving the sensor. If `e1` is a vector, it must have the same size as the other vector input arguments of `height2grndrange`.

Data Types: `double`

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `Method="CRPL", SurfaceRefractivity=300, RefractionExponent=0.15`

### **Method — Earth model**

"Curved" (default) | "Flat" | "CRPL"

Earth model used for the computation, specified as "Curved", "Flat", or "CRPL".

- "Curved" — Assumes a "Curved Earth Model" on page 1-69 with a 4/3 effective Earth radius, which is an approximation used for modeling refraction effects in the troposphere. To specify another value for the effective Earth radius, use the `EffectiveEarthRadius` name-value argument.
- "Flat" — Assumes a "Flat Earth Model" on page 1-68. In this case, the effective Earth radius is infinite.
- "CRPL" — Assumes a curved Earth model with the atmosphere defined by the "CRPL Exponential Reference Atmosphere Model" on page 1-70 with a refractivity of 313 N-units and a refraction exponent of  $0.143859 \text{ km}^{-1}$ . To specify other values for the refractivity and the refraction exponent, use the `SurfaceRefractivity` and `RefractionExponent` name value arguments. This method requires `e1` to be positive. For more information, see "CRPL Model Geometry" on page 1-71.

Data Types: `char` | `string`

### **EffectiveEarthRadius — Effective Earth radius**

4/3 of Earth's radius (default) | positive scalar

Effective Earth radius in meters, specified as a positive scalar. If this argument is not specified, `height2grndrange` calculates the effective Earth radius using a refractivity gradient of  $-39 \times 10^{-9}$

N-units/meter, which results in approximately 4/3 of the real Earth radius. This argument applies only if `Method` is specified as "Curved".

Data Types: `double`

### **SurfaceRefractivity** — Surface refractivity

313 (default) | real-valued scalar

Surface refractivity in N-units, specified as a nonnegative real-valued scalar. The surface refractivity is a parameter of the "CRPL Exponential Reference Atmosphere Model" on page 1-70 used by `height2grndrange`. This argument applies only if `Method` is specified as "CRPL".

Data Types: `double`

### **RefractionExponent** — Refraction exponent

0.143859 (default) | real-valued scalar

Refraction exponent, specified as a nonnegative real-valued scalar. The refraction exponent is a parameter of the "CRPL Exponential Reference Atmosphere Model" on page 1-70 used by `height2grndrange`. This argument applies only if `Method` is specified as "CRPL".

Data Types: `double`

## **Output Arguments**

### **gr** — Ground range to target

real-valued scalar | real-valued row vector

Ground range to target in meters, specified as a real-valued scalar or row vector. If `gr` is a vector, it has the same size as the vector input arguments of `height2grndrange`.

## **More About**

### **Flat Earth Model**

The flat Earth model assumes that the Earth has infinite radius and that the index of refraction of air is uniform throughout the atmosphere. The flat Earth model is applicable over short distances and is used in applications like communications, automotive radar, and synthetic aperture radar (SAR).

Given the antenna height  $h_a$  and the initial elevation angle  $\theta_0$ , the model relates the target height  $h_T$  and the slant range  $R_T$  by

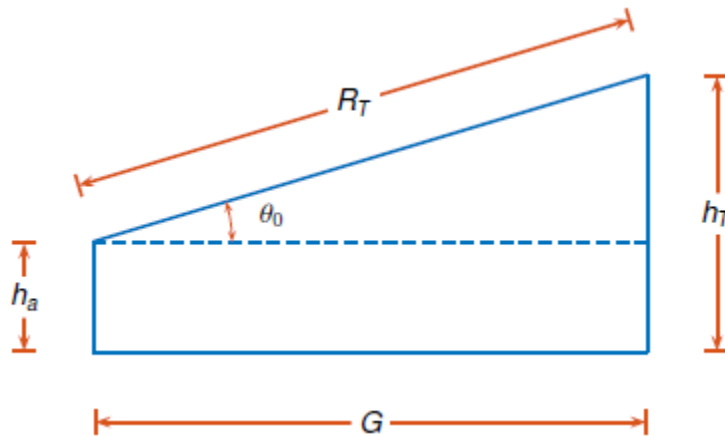
$$h_T = h_a + R_T \sin \theta_0 \quad \Leftrightarrow \quad R_T = (h_T - h_a) \csc \theta_0,$$

so knowing one of those magnitudes enables you to compute the other. The actual range  $R$  is equal to the slant range. The true elevation angle  $\theta_T$  is equal to the initial elevation angle.

To compute the ground range  $G$ , use

$$G = (h_T - h_a) \cot \theta_0.$$





### Curved Earth Model

The fact that the index of refraction of air depends on height can be treated approximately by using an effective Earth's radius larger than the actual value.

Given the effective Earth's radius  $R_0$ , the antenna height  $h_a$ , and the initial elevation angle  $\theta_0$ , the model relates the target height  $h_T$  and the slant range  $R_T$  by

$$(R_0 + h_T)^2 = (R_0 + h_a)^2 + R_T^2 + 2R_T(R_0 + h_a)\sin\theta_0,$$

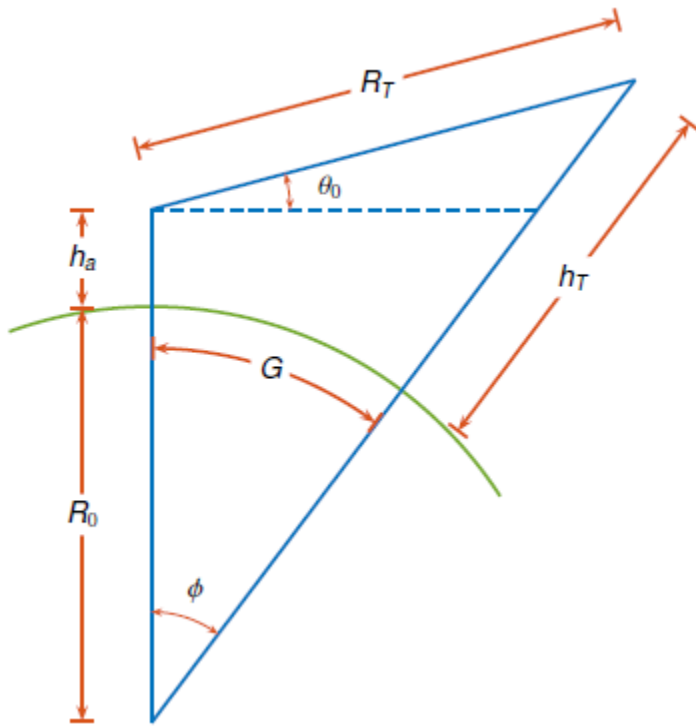
so knowing one of those magnitudes enables you to compute the other. In particular,

$$h_T = \sqrt{(R_0 + h_a)^2 + R_T^2 + 2R_T(R_0 + h_a)\sin\theta_0} - R_0.$$

The actual range  $R$  is equal to the slant range. The true elevation angle  $\theta_T$  is equal to the initial elevation angle.

To compute the ground range  $G$ , use

$$G = R_0\phi = R_0\arcsin\frac{R_T\cos\theta_0}{R_0 + h_T}.$$



A standard propagation model uses an effective Earth's radius that is 4/3 times the actual value. This model has two major limitations:

- 1 The model implies a value for the index of refraction near the Earth's surface that is valid only for certain areas and at certain times of the year. To mitigate this limitation, use an effective Earth's radius based on the near-surface refractivity value.
- 2 The model implies a value for the gradient of the index of refraction that is unrealistically low at heights of around 8 km. To partially mitigate this limitation, use an effective Earth's radius based on the platform altitudes.

For more information, see `effearthradius`.

### CRPL Exponential Reference Atmosphere Model

Atmospheric refraction evidences itself as a deviation in an electromagnetic ray from a straight line due to variation in air density as a function of height. The Central Radio Propagation Laboratory (CRPL) exponential reference atmosphere model treats refraction effects by assuming that the index of refraction  $n(h)$  and the refractivity  $N$  decay exponentially with height. The model defines

$$N = (n(h) - 1) \times 10^6 = N_s e^{-R_{\text{exp}} h},$$

where  $N_s$  is the atmospheric refractivity value (in units of  $10^{-6}$ ) at the surface of the earth,  $R_{\text{exp}}$  is the decay constant, and  $h$  is the height above the surface in kilometers. Thus

$$n(h) = 1 + (N_s \times 10^{-6}) e^{-R_{\text{exp}} h}.$$

The default value of  $N_s$  is 313 N-units and can be modified using the `SurfaceRefractivity` name-value argument in functions that accept it. The default value of  $R_{\text{exp}}$  is  $0.143859 \text{ km}^{-1}$  and can be modified using the `RefractionExponent` name-value argument in functions that accept it.

### CRPL Model Geometry

When the refractivity of air is incorporated into the curved Earth model, the ray paths do not follow a straight line but curve downward. (This statement assumes standard atmospheric propagation and nonnegative elevation angles.) The true elevation angle  $\theta_T$  is different from the initial  $\theta_0$ . The actual range  $R$ , which is the distance along the curved path  $R'$ , is different from the slant range  $R_T$ .

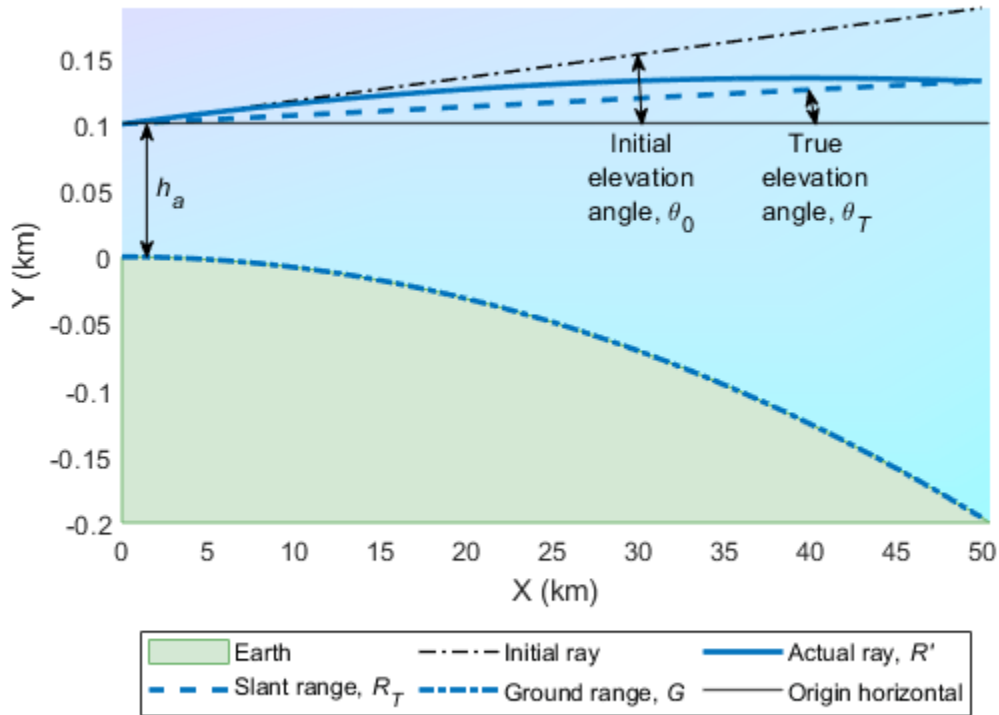
Given the Earth's radius  $R_0$ , the antenna height  $h_a$ , the initial elevation angle  $\theta_0$ , and the height-dependent index of refraction  $n(h)$  with value  $n_0$  at  $h = 0$ , the modified model relates the target height  $h_T$  and the actual range  $R$  by

$$R = \int_0^{h_T - h_a} n(h) dh \left( 1 - \left( \frac{n_0 \cos \theta_0}{n(h) \left( 1 + \frac{h}{R_0 + h_a} \right)} \right)^2 \right)^{-1/2}.$$

When Method is specified as "CRPL", the integral is solved using  $n(h)$  from "CRPL Exponential Reference Atmosphere Model" on page 1-70.

To compute the ground range  $G$ , use

$$G = \int_0^{h_T - h_a} \frac{dh}{1 + \frac{h}{R_0 + h_a}} \left( \left( \frac{n(h) \left( 1 + \frac{h}{R_0 + h_a} \right)}{n_0 \cos \theta_0} \right)^2 - 1 \right)^{-1/2}.$$



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Apps

**Radar Designer**

### Functions

blakechart | el2height | height2el | height2range | radarvcd | range2height | refractionexp

### Topics

“Radar Vertical Coverage over Terrain”  
 “Modeling Target Position Errors due to Refraction”

**Introduced in R2021b**

# height2range

Convert target height to propagated range

## Syntax

```
r = height2range(tgtht,anht,el)
r = height2range(tgtht,anht,el,Name=Value)
[r,trueSR,trueEL] = height2range( ____,Method="CRPL")
```

## Description

`r = height2range(tgtht,anht,el)` returns the propagated range to the target, `r`, as a function of the target height `tgtht`, the sensor height `anht`, and the local elevation angle `el` assuming a "Curved Earth Model" on page 1-76 with a 4/3 effective Earth radius.

`r = height2range(tgtht,anht,el,Name=Value)` specifies additional inputs using name-value arguments. For example, you can specify a flat Earth model, a curved Earth model with a given radius, or a "CRPL Exponential Reference Atmosphere Model" on page 1-77 with custom values.

`[r,trueSR,trueEL] = height2range( ____,Method="CRPL")` also returns the true slant range and the true elevation angle when you specify the Earth model as "CRPL".

## Examples

### Range Along Propagated Path

Compute the range along the propagated path for a target height of 1 km, an antenna height of 10 meters, and an elevation angle of 2 degrees at the radar. Assume a curved Earth model with a 4/3 effective Earth radius.

```
r = height2range(1e3,10,2)
r = 2.7125e+04
```

### Propagated Range Using CRPL Atmosphere

Compute the range along the propagated path using the CRPL exponential reference atmosphere. Assume a target height of 1 km, an antenna height of 10 meters, and an elevation angle of 2 degrees at the radar. Additionally, compute the true slant range and the true elevation angle to the target.

```
[R,SRtrue,elTrue] = height2range(1e3,10,2,Method="CRPL")
R = 2.7171e+04
SRtrue = 2.7163e+04
elTrue = 1.9666
```

## Input Arguments

### **tgtht — Target height**

nonnegative real-valued scalar | nonnegative real-valued vector

Target height in meters, specified as a nonnegative real-valued scalar or vector. If `tgtht` is a vector, it must have the same size as the other vector input arguments of `height2range`. Heights are referenced to the ground.

Data Types: `double`

### **anht — Sensor height**

nonnegative real-valued scalar | nonnegative real-valued vector

Sensor height in meters, specified as a nonnegative real-valued scalar or vector. If `anht` is a vector, it must have the same size as the other vector input arguments of `height2range`. Heights are referenced to the ground.

Data Types: `double`

### **e1 — Local elevation angle**

real-valued scalar | real-valued vector

Local elevation angle in degrees, specified as a real-valued scalar or vector. The local elevation angle is the initial elevation angle of the ray leaving the sensor. If `e1` is a vector, it must have the same size as the other vector input arguments of `height2range`.

Data Types: `double`

## Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `Method="CRPL", SurfaceRefractivity=300, RefractionExponent=0.15`

### **Method — Earth model**

"Curved" (default) | "Flat" | "CRPL"

Earth model used for the computation, specified as "Curved", "Flat", or "CRPL".

- "Curved" — Assumes a "Curved Earth Model" on page 1-76 with a 4/3 effective Earth radius, which is an approximation used for modeling refraction effects in the troposphere. To specify another value for the effective Earth radius, use the `EffectiveEarthRadius` name-value argument.
- "Flat" — Assumes a "Flat Earth Model" on page 1-76. In this case, the effective Earth radius is infinite.
- "CRPL" — Assumes a curved Earth model with the atmosphere defined by the "CRPL Exponential Reference Atmosphere Model" on page 1-77 with a refractivity of 313 N-units and a refraction exponent of  $0.143859 \text{ km}^{-1}$ . To specify other values for the refractivity and the refraction exponent, use the `SurfaceRefractivity` and `RefractionExponent` name value arguments. This method requires `e1` to be positive. For more information, see "CRPL Model Geometry" on page 1-78.

Data Types: `char` | `string`

**EffectiveEarthRadius — Effective Earth radius**

4/3 of Earth's radius (default) | positive scalar

Effective Earth radius in meters, specified as a positive scalar. If this argument is not specified, `height2range` calculates the effective Earth radius using a refractivity gradient of  $-39 \times 10^{-9}$  N-units/meter, which results in approximately 4/3 of the real Earth radius. This argument applies only if `Method` is specified as "Curved".

Data Types: double

**SurfaceRefractivity — Surface refractivity**

313 (default) | real-valued scalar

Surface refractivity in N-units, specified as a nonnegative real-valued scalar. The surface refractivity is a parameter of the "CRPL Exponential Reference Atmosphere Model" on page 1-77 used by `height2range`. This argument applies only if `Method` is specified as "CRPL".

Data Types: double

**RefractionExponent — Refraction exponent**

0.143859 (default) | real-valued scalar

Refraction exponent, specified as a nonnegative real-valued scalar. The refraction exponent is a parameter of the "CRPL Exponential Reference Atmosphere Model" on page 1-77 used by `height2range`. This argument applies only if `Method` is specified as "CRPL".

Data Types: double

**Output Arguments****r — Propagated range**

real-valued scalar | real-valued row vector

Propagated range between the target and the sensor in meters, returned as a real-valued scalar or row vector. If `r` is a vector, it has the same size as the vector input arguments of `height2range`.

**trueSR — True slant range**

real-valued scalar | real-valued row vector

True slant range in meters, returned as a real-valued scalar or row vector. If `trueSR` is a vector, it has the same size as the vector input arguments of `height2range`. This argument is available only if `Method` is specified as "CRPL".

**trueEL — True elevation angle**

real-valued scalar | real-valued row vector

True elevation angle in degrees, returned as a real-valued scalar or row vector. If `trueEL` is a vector, it has the same size as the vector input arguments of `height2range`. This argument is available only if `Method` is specified as "CRPL".

## More About

### Flat Earth Model

The flat Earth model assumes that the Earth has infinite radius and that the index of refraction of air is uniform throughout the atmosphere. The flat Earth model is applicable over short distances and is used in applications like communications, automotive radar, and synthetic aperture radar (SAR).

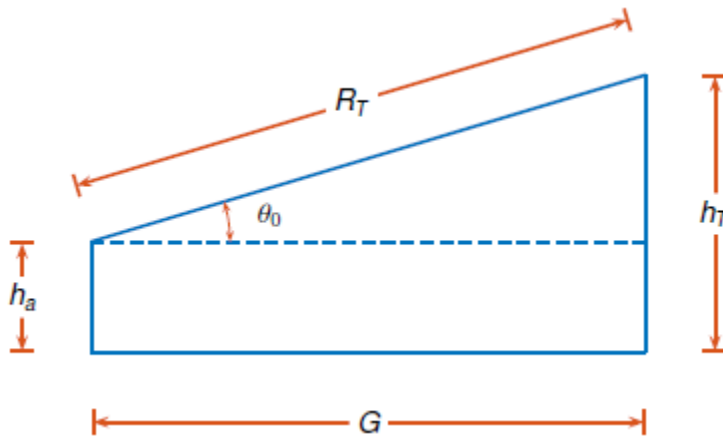
Given the antenna height  $h_a$  and the initial elevation angle  $\theta_0$ , the model relates the target height  $h_T$  and the slant range  $R_T$  by

$$h_T = h_a + R_T \sin \theta_0 \quad \Leftrightarrow \quad R_T = (h_T - h_a) \csc \theta_0,$$

so knowing one of those magnitudes enables you to compute the other. The actual range  $R$  is equal to the slant range. The true elevation angle  $\theta_T$  is equal to the initial elevation angle.

To compute the ground range  $G$ , use

$$G = (h_T - h_a) \cot \theta_0.$$



### Curved Earth Model

The fact that the index of refraction of air depends on height can be treated approximately by using an effective Earth's radius larger than the actual value.

Given the effective Earth's radius  $R_0$ , the antenna height  $h_a$ , and the initial elevation angle  $\theta_0$ , the model relates the target height  $h_T$  and the slant range  $R_T$  by

$$(R_0 + h_T)^2 = (R_0 + h_a)^2 + R_T^2 + 2R_T(R_0 + h_a)\sin\theta_0,$$

so knowing one of those magnitudes enables you to compute the other. In particular,

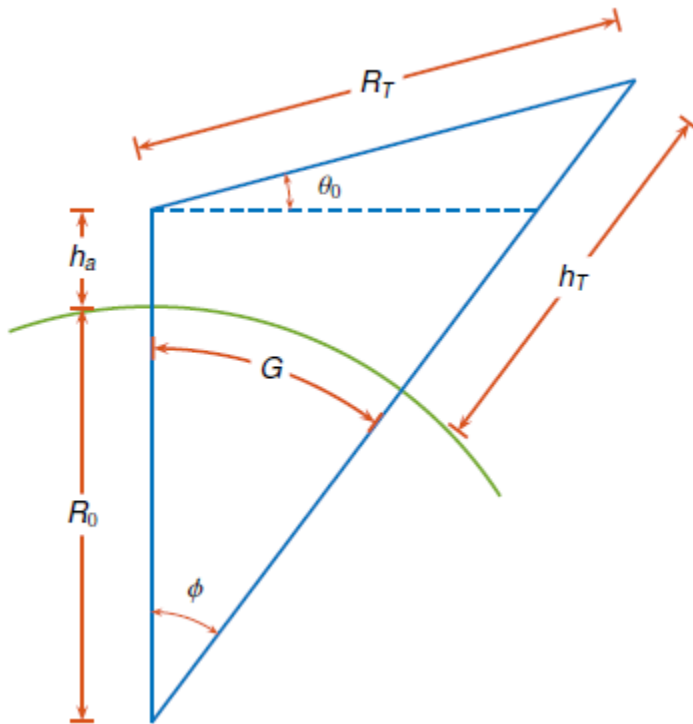
$$h_T = \sqrt{(R_0 + h_a)^2 + R_T^2 + 2R_T(R_0 + h_a)\sin\theta_0} - R_0.$$

The actual range  $R$  is equal to the slant range. The true elevation angle  $\theta_T$  is equal to the initial elevation angle.

To compute the ground range  $G$ , use



$$G = R_0\phi = R_0\arcsin\frac{R_T\cos\theta_0}{R_0 + h_T}.$$



A standard propagation model uses an effective Earth's radius that is 4/3 times the actual value. This model has two major limitations:

- 1 The model implies a value for the index of refraction near the Earth's surface that is valid only for certain areas and at certain times of the year. To mitigate this limitation, use an effective Earth's radius based on the near-surface refractivity value.
- 2 The model implies a value for the gradient of the index of refraction that is unrealistically low at heights of around 8 km. To partially mitigate this limitation, use an effective Earth's radius based on the platform altitudes.

For more information, see `effearthradius`.

### CRPL Exponential Reference Atmosphere Model

Atmospheric refraction evidences itself as a deviation in an electromagnetic ray from a straight line due to variation in air density as a function of height. The Central Radio Propagation Laboratory (CRPL) exponential reference atmosphere model treats refraction effects by assuming that the index of refraction  $n(h)$  and the refractivity  $N$  decay exponentially with height. The model defines

$$N = (n(h) - 1) \times 10^6 = N_s e^{-R_{\text{exp}} h},$$

where  $N_s$  is the atmospheric refractivity value (in units of  $10^{-6}$ ) at the surface of the earth,  $R_{\text{exp}}$  is the decay constant, and  $h$  is the height above the surface in kilometers. Thus

$$n(h) = 1 + (N_s \times 10^{-6}) e^{-R_{\text{exp}} h}.$$

The default value of  $N_s$  is 313 N-units and can be modified using the `SurfaceRefractivity` name-value argument in functions that accept it. The default value of  $R_{\text{exp}}$  is  $0.143859 \text{ km}^{-1}$  and can be modified using the `RefractionExponent` name-value argument in functions that accept it.

### CRPL Model Geometry

When the refractivity of air is incorporated into the curved Earth model, the ray paths do not follow a straight line but curve downward. (This statement assumes standard atmospheric propagation and nonnegative elevation angles.) The true elevation angle  $\theta_T$  is different from the initial  $\theta_0$ . The actual range  $R$ , which is the distance along the curved path  $R'$ , is different from the slant range  $R_T$ .

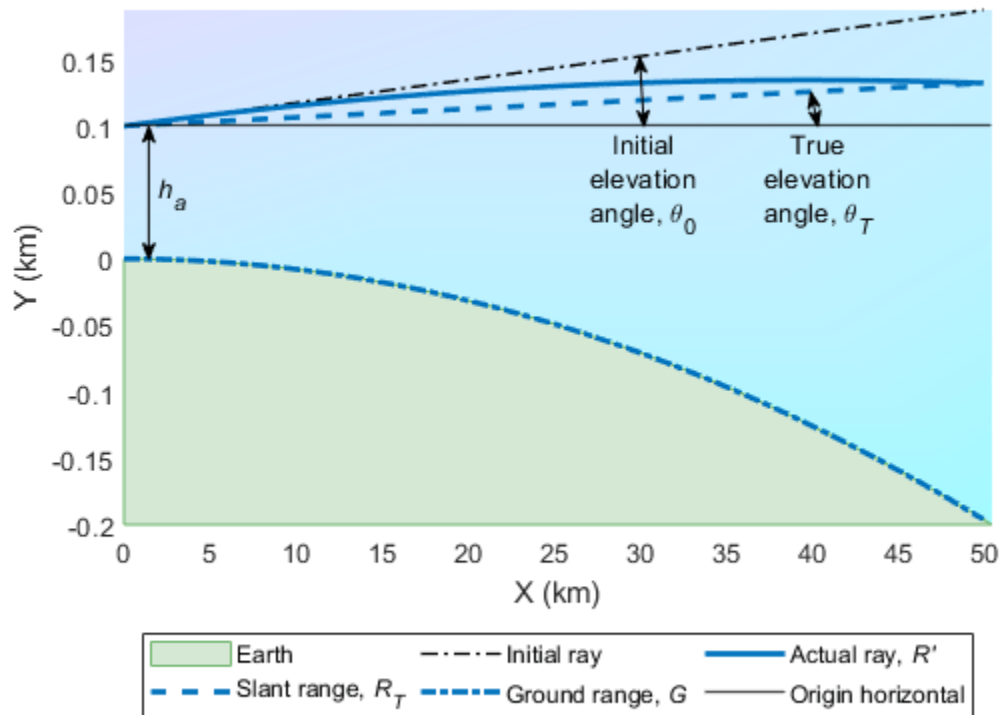
Given the Earth's radius  $R_0$ , the antenna height  $h_a$ , the initial elevation angle  $\theta_0$ , and the height-dependent index of refraction  $n(h)$  with value  $n_0$  at  $h = 0$ , the modified model relates the target height  $h_T$  and the actual range  $R$  by

$$R = \int_0^{h_T - h_a} n(h) dh \left( 1 - \left( \frac{n_0 \cos \theta_0}{n(h) \left( 1 + \frac{h}{R_0 + h_a} \right)} \right)^2 \right)^{-1/2}.$$

When `Method` is specified as "CRPL", the integral is solved using  $n(h)$  from "CRPL Exponential Reference Atmosphere Model" on page 1-77.

To compute the ground range  $G$ , use

$$G = \int_0^{h_T - h_a} \frac{dh}{1 + \frac{h}{R_0 + h_a}} \left( \left( \frac{n(h) \left( 1 + \frac{h}{R_0 + h_a} \right)}{n_0 \cos \theta_0} \right)^2 - 1 \right)^{-1/2}.$$



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Apps

Radar Designer

### Functions

blakechart | el2height | height2el | height2grndrange | radarvcd | range2height | refractionexp

### Topics

“Radar Vertical Coverage over Terrain”

“Modeling Target Position Errors due to Refraction”

Introduced in R2021b

## horizonrange

Horizon range

### Syntax

```
Rh = horizonrange(H)
Rh = horizonrange(H,Re)
Rh = horizonrange( __ , 'SurfaceHeight' , surfht)
```

### Description

`Rh = horizonrange(H)` returns the horizon range, `Rh`, of a radar system `H` meters above the surface. The computation uses an effective earth radius of approximately 4/3 times the actual earth radius.

`Rh = horizonrange(H,Re)` specifies the effective earth radius, `Re`.

`Rh = horizonrange( __ , 'SurfaceHeight' , surfht)` also specifies the surface height, `surfht`.

### Input Arguments

#### **H**

Height of radar system above surface, specified as a scalar or a length-*M* vector. Units are in meters.

#### **Re**

Effective earth radius, specified as a positive scalar. Units are in meters

**Default:** `effearthradius`, which is approximately 4/3 times the actual earth radius

#### **surfht**

Height of earth surface at the horizon, specified as a scalar or length-*M* vector. This input can also be interpreted as the height of significant ground clutter at the horizon. If `H` and `surfht` are vectors, their lengths must be equal. Defaults to 0 m.

### Output Arguments

#### **Rh**

Horizon range in meters of radar system at altitude `H`.

### Examples

#### **Compute Range to Horizon**

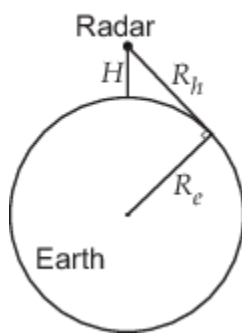
Find the range to the horizon from an antenna that is 30 m high.

`R = horizonrange(30)`

## More About

### Horizon Range

The horizon range of a radar system is the distance from the radar system to the earth along a tangent. Beyond the horizon range, the radar system detects no return from the surface through a direct path.



The value of the horizon range is:

$$\sqrt{2R_e H + H^2}$$

where  $R_e$  is the effective earth radius and  $H$  is the altitude of the radar system.

## References

- [1] Long, Maurice W. *Radar Reflectivity of Land and Sea*, 3rd Ed. Boston: Artech House, 2001.
- [2] Skolnik, M. *Introduction to Radar Systems*, 3rd Ed. New York: McGraw-Hill, 2001.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

## See Also

`depressionang` | `effearthradius` | `grazingang`

**Introduced in R2021a**

## surfacegamma

Gamma value for different terrains

### Syntax

```
G = surfacegamma(TerrainType)
G = surfacegamma(TerrainType,FREQ)
surfacegamma
```

### Description

`G = surfacegamma(TerrainType)` returns the  $\gamma$  value for the specified terrain. The  $\gamma$  value is for an operating frequency of 10 GHz.

`G = surfacegamma(TerrainType, FREQ)` specifies the operating frequency of the system.

`surfacegamma` displays several terrain types and their corresponding  $\gamma$  values. These  $\gamma$  values are for an operating frequency of 10 GHz.

### Input Arguments

#### TerrainType

Character vectors that describe the terrain type. Valid values are:

- 'sea state 3'
- 'sea state 5'
- 'woods'
- 'metropolitan'
- 'rugged mountain'
- 'farmland'
- 'wooded hill'
- 'flatland'

#### FREQ

Operating frequency of radar system in hertz. This value can be a scalar or vector.

**Default:** 10e9

### Output Arguments

#### G

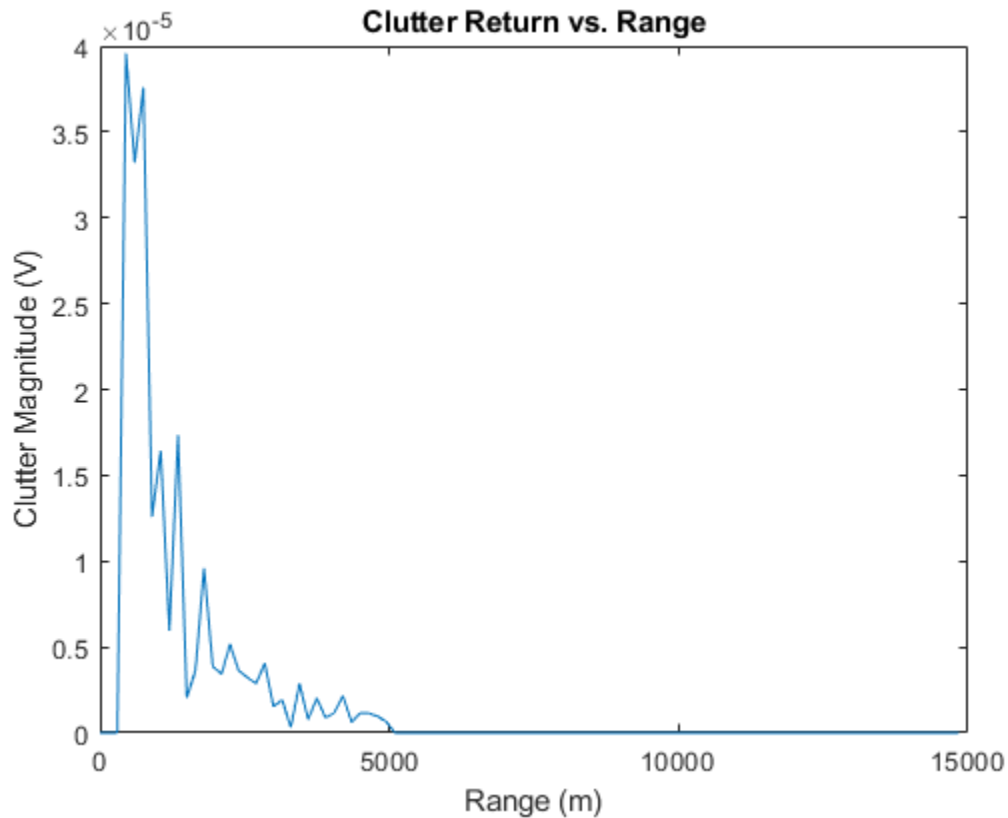
Value of  $\gamma$  in decibels, for constant  $\gamma$  clutter model.

## Examples

### Simulate Constant Gamma Clutter

Determine the  $\gamma$  value for a wooded area, and then simulate the clutter return from that area. Assume the radar system uses a single cosine pattern antenna element and has an operating frequency of 300 MHz.

```
fc = 300e6;
g = surfacegamma('woods',fc);
clutter = constantGammaClutter('Gamma',g, ...
    'Sensor',phased.CosineAntennaElement, ...
    'OperatingFrequency',fc);
x = clutter();
r = (0:numel(x)-1)/(2*clutter.SampleRate) * ...
    clutter.PropagationSpeed;
plot(r,abs(x))
xlabel('Range (m)')
ylabel('Clutter Magnitude (V)')
title('Clutter Return vs. Range')
```



## More About

### Gamma

A frequently used model for clutter simulation is the constant gamma model. This model uses a parameter,  $\gamma$ , to describe clutter characteristics of different types of terrain. Values of  $\gamma$  are derived from measurements.

## Algorithms

The  $\gamma$  values for the terrain types 'sea state 3', 'sea state 5', 'woods', 'metropolitan', and 'rugged mountain' are from [2].

The  $\gamma$  values for the terrain types 'farmland', 'wooded hill', and 'flatland' are from [3].

Measurements provide values of  $\gamma$  for a system operating at 10 GHz. The  $\gamma$  value for a system operating at frequency  $f$  is:

$$\gamma = \gamma_0 + 5\log\left(\frac{f}{f_0}\right)$$

where  $\gamma_0$  is the value at frequency  $f_0 = 10$  GHz.

## References

- [1] Barton, David. "Land Clutter Models for Radar Design and Analysis," *Proceedings of the IEEE*. Vol. 73, Number 2, February, 1985, pp. 198-204.
- [2] Long, Maurice W. *Radar Reflectivity of Land and Sea*, 3rd Ed. Boston: Artech House, 2001.
- [3] Nathanson, Fred E., J. Patrick Reilly, and Marvin N. Cohen. *Radar Design Principles*, 2nd Ed. Mendham, NJ: SciTech Publishing, 1999.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

## See Also

grazingang | horizonrange | constantGammaClutter

**Introduced in R2021a**



# surfclutterrcs

Surface clutter radar cross section (RCS)

## Syntax

```
RCS = surfclutterrcs(NRCS,R,az,el,graz,tau)
RCS = surfclutterrcs(NRCS,R,az,el,graz,tau,c)
```

## Description

`RCS = surfclutterrcs(NRCS,R,az,el,graz,tau)` returns the radar cross section (RCS) of a clutter patch that is of range `R` meters away from the radar system. `az` and `el` are the radar system azimuth and elevation beamwidths, respectively, corresponding to the clutter patch. `graz` is the grazing angle of the clutter patch relative to the radar. `tau` is the pulse width of the transmitted signal. The calculation automatically determines whether the surface clutter area is beam limited or pulse limited, based on the values of the input arguments.

`RCS = surfclutterrcs(NRCS,R,az,el,graz,tau,c)` specifies the propagation speed in meters per second.

## Input Arguments

### NRCS

Normalized radar cross section of clutter patch in units of square meters/square meters.

### R

Range of clutter patch from radar system, in meters.

### az

Azimuth beamwidth of radar system corresponding to clutter patch, in degrees.

### el

Elevation beamwidth of radar system corresponding to clutter patch, in degrees.

### graz

Grazing angle of clutter patch relative to radar system, in degrees.

### tau

Pulse width of transmitted signal, in seconds.

### c

Propagation speed, in meters per second.

**Default:** Speed of light

## Output Arguments

### RCS

Radar cross section of clutter patch.

## Examples

### Compute Surface Clutter RCS

Calculate the RCS of a clutter patch and estimate the clutter-to-noise ratio (CNR) at the receiver. Assume that the patch has a normalized radar cross section (NRCS) of 1 m<sup>2</sup>/m<sup>2</sup> and is 1.0 km away from the radar system. The azimuth and elevation beamwidths are 1° and 3°, respectively. The grazing angle is 10°. The pulse width is 10μs. The radar operates at a wavelength of 1 cm with a peak power of 5 kW.

```
nrcs = 1;  
rng = 1.0e3;  
az = 1;  
el = 3;  
graz = 10;  
tau = 10e-6;  
lambda = 0.01;  
ppow = 5000;  
rcs = surfclutterrcs(nrcs,rng,az,el,graz,tau)  
  
rcs = 5.2627e+03  
  
cnr = radareqsnr(lambda,rng,ppow,tau,'rcs',rcs)  
  
cnr = 75.2006
```

## Tips

- You can calculate the clutter-to-noise ratio using the output of this function as the RCS input argument value in `radareqsnr`.

## Algorithms

See [1].

## References

- [1] Richards, M. A. *Fundamentals of Radar Signal Processing*. New York: McGraw-Hill, 2005, pp. 57-63.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

### **See Also**

grazingang | surfacegamma | radareqsnr | uv2azel | phitheta2azel

**Introduced in R2021a**

## range2height

Convert propagated range to target height

### Syntax

```
tgtht = range2height(r,anht,el)
tgtht = range2height(r,anht,el,Name=Value)
```

### Description

`tgtht = range2height(r,anht,el)` returns the target height `tgtht` as a function of the propagated range `r`, the sensor height `anht`, and the local elevation angle `el` assuming a “Curved Earth Model” on page 1-92 with a 4/3 effective Earth radius.

`tgtht = range2height(r,anht,el,Name=Value)` specifies additional inputs using name-value arguments. For example, you can specify a flat Earth model, a curved Earth model with a given radius, or a “CRPL Exponential Reference Atmosphere Model” on page 1-93 with custom values.

### Examples

#### Target Height from Propagated Range

Determine the target height in meters given a range of 300 km, a sensor height of 10 meters, and an elevation angle of 0.5 degrees. Assume a curved Earth with an effective radius equal to 4/3 times the Earth's actual radius.

```
R = 300e3;
anht = 10;
el = 0.5;

range2height(R,anht,el)

ans = 7.9325e+03
```

#### Target Height Using Different Earth Models

Compute target heights in meters using different Earth models and compare the values you obtain. Assume a range of 200 km and an antenna height of 100 meters. Use a range of elevation angles from 0 to 5 degrees.

```
R = 200e3;
anht = 100;
el = (0:0.1:5)';
```

Compute the target height for the given parameters assuming a flat Earth.

```
tgthtFlat = range2height(R,anht,el,Method="Flat");
```

Compute the target height for the given parameters assuming free-space propagation with a curved Earth.

```
r0 = physconst("EarthRadius");
tgthtFS = range2height(R,anht,el,Method="Curved", ...
    EffectiveEarthRadius=r0);
```

Compute the target height for the given parameters assuming a 4/3 effective Earth radius.

```
tgthtEffRad = range2height(R,anht,el);
```

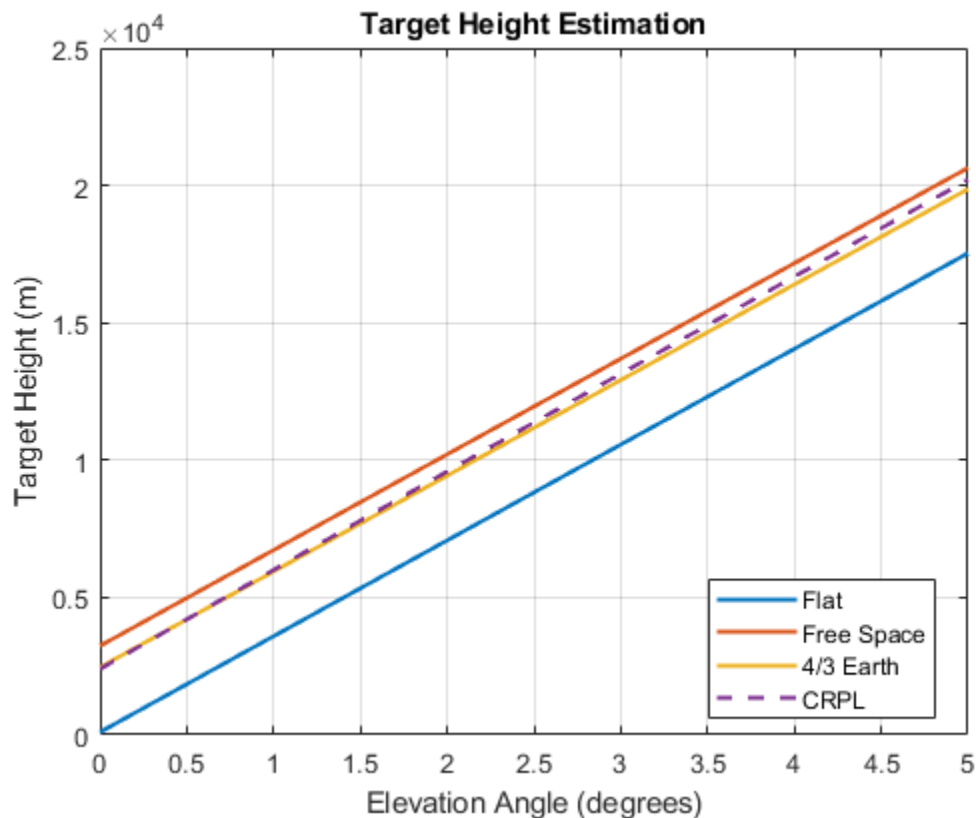
Compute the target height for the given parameters assuming the CRPL atmospheric model.

```
tgthtCRPL = range2height(R,anht,el,Method="CRPL");
```

Plot the results.

```
plot(el,[tgthtFlat(:) tgthtFS(:) tgthtEffRad(:)], ...
    el,tgthtCRPL,'--',LineWidth=1.5)
grid on

xlabel("Elevation Angle (degrees)")
ylabel("Target Height (m)")
legend(["Flat" "Free Space" "4/3 Earth" "CRPL"],Location="best")
title("Target Height Estimation")
```



## Input Arguments

### **r** — Propagated range

real-valued scalar | real-valued vector

Propagated range between the target and the sensor in meters, specified as a real-valued scalar or vector. If **r** is a vector, it must have the same size as the other vector input arguments of `range2height`.

Data Types: double

### **anht** — Sensor height

nonnegative real-valued scalar | nonnegative real-valued vector

Sensor height in meters, specified as a nonnegative real-valued scalar or vector. If **anht** is a vector, it must have the same size as the other vector input arguments of `range2height`. Heights are referenced to the ground.

Data Types: double

### **e1** — Local elevation angle

real-valued scalar | real-valued vector

Local elevation angle in degrees, specified as a real-valued scalar or vector. The local elevation angle is the initial elevation angle of the ray leaving the sensor. If **e1** is a vector, it must have the same size as the other vector input arguments of `range2height`.

Data Types: double

## Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example:

```
Method="CRPL", SurfaceRefractivity=300, RefractionExponent=0.15, MaxNumIterations=8, Tolerance=1e-7
```

### **Method** — Earth model

"Curved" (default) | "Flat" | "CRPL"

Earth model used for the computation, specified as "Curved", "Flat", or "CRPL".

- "Curved" — Assumes a "Curved Earth Model" on page 1-92 with a 4/3 effective Earth radius, which is an approximation used for modeling refraction effects in the troposphere. To specify another value for the effective Earth radius, use the `EffectiveEarthRadius` name-value argument.
- "Flat" — Assumes a "Flat Earth Model" on page 1-92. In this case, the effective Earth radius is infinite.
- "CRPL" — Assumes a curved Earth model with the atmosphere defined by the "CRPL Exponential Reference Atmosphere Model" on page 1-93 with a refractivity of 313 N-units and a refraction exponent of  $0.143859 \text{ km}^{-1}$ . To specify other values for the refractivity and the refraction exponent, use the `SurfaceRefractivity` and `RefractionExponent` name value arguments. This method requires **e1** to be positive. For more information, see "CRPL Model Geometry" on page 1-94.

Data Types: char | string

### **EffectiveEarthRadius — Effective Earth radius**

4/3 of Earth's radius (default) | positive scalar

Effective Earth radius in meters, specified as a positive scalar. If this argument is not specified, `range2height` calculates the effective Earth radius using a refractivity gradient of  $-39 \times 10^{-9}$  N-units/meter, which results in approximately 4/3 of the real Earth radius. This argument applies only if `Method` is specified as "Curved".

Data Types: double

### **SurfaceRefractivity — Surface refractivity**

313 (default) | real-valued scalar

Surface refractivity in N-units, specified as a nonnegative real-valued scalar. The surface refractivity is a parameter of the "CRPL Exponential Reference Atmosphere Model" on page 1-93 used by `range2height`. This argument applies only if `Method` is specified as "CRPL".

Data Types: double

### **RefractionExponent — Refraction exponent**

0.143859 (default) | real-valued scalar

Refraction exponent, specified as a nonnegative real-valued scalar. The refraction exponent is a parameter of the "CRPL Exponential Reference Atmosphere Model" on page 1-93 used by `range2height`. This argument applies only if `Method` is specified as "CRPL".

Data Types: double

### **MaxNumIterations — Maximum number of iterations for the CRPL method**

10 (default) | nonnegative scalar integer

Maximum number of iterations for the CRPL method, specified as a nonnegative scalar integer. This input acts as a safeguard to preempt long iterative calculations. This argument applies only if `Method` is specified as "CRPL".

If `MaxNumIterations` is set to 0, `range2height` performs a faster but less accurate noniterative CRPL calculation. The noniterative calculation has a maximum height error of 0.056388 m (0.185 ft) at a target height of 30,480 m (100,000 ft) and an elevation angle of 0. The height error for the noniterative method decreases with decreasing target height and increasing elevation angle.

Data Types: double

### **Tolerance — Numerical tolerance for the CRPL method**

1e-6 (default) | positive real scalar

Numerical tolerance for the CRPL method, specified as a positive real scalar. The iterative process terminates when the numerical tolerance is achieved. This argument applies only if `Method` is specified as "CRPL" and `MaxNumIterations` is greater than 0.

Data Types: double

## **Output Arguments**

### **tgtht — Target height**

nonnegative real-valued scalar | nonnegative real-valued row vector

Target height in meters, returned as a nonnegative real-valued scalar or row vector. If `tgtht` is a vector, it has the same size as the vector input arguments of `range2height`. The height is referenced to the ground.

## More About

### Flat Earth Model

The flat Earth model assumes that the Earth has infinite radius and that the index of refraction of air is uniform throughout the atmosphere. The flat Earth model is applicable over short distances and is used in applications like communications, automotive radar, and synthetic aperture radar (SAR).

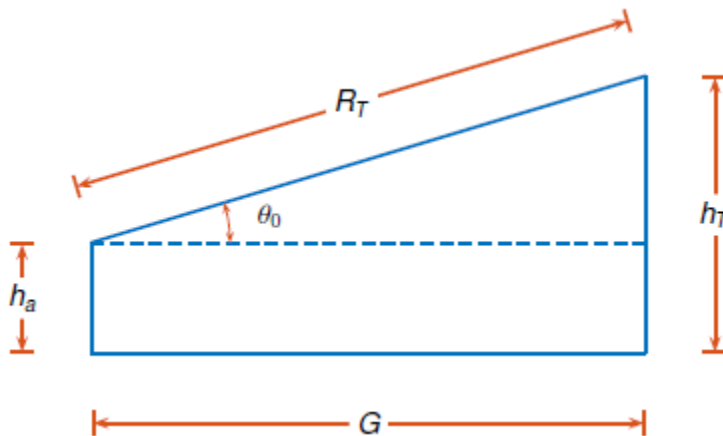
Given the antenna height  $h_a$  and the initial elevation angle  $\theta_0$ , the model relates the target height  $h_T$  and the slant range  $R_T$  by

$$h_T = h_a + R_T \sin \theta_0 \quad \Leftrightarrow \quad R_T = (h_T - h_a) \csc \theta_0,$$

so knowing one of those magnitudes enables you to compute the other. The actual range  $R$  is equal to the slant range. The true elevation angle  $\theta_T$  is equal to the initial elevation angle.

To compute the ground range  $G$ , use

$$G = (h_T - h_a) \cot \theta_0.$$



### Curved Earth Model

The fact that the index of refraction of air depends on height can be treated approximately by using an effective Earth's radius larger than the actual value.

Given the effective Earth's radius  $R_0$ , the antenna height  $h_a$ , and the initial elevation angle  $\theta_0$ , the model relates the target height  $h_T$  and the slant range  $R_T$  by

$$(R_0 + h_T)^2 = (R_0 + h_a)^2 + R_T^2 + 2R_T(R_0 + h_a) \sin \theta_0,$$

so knowing one of those magnitudes enables you to compute the other. In particular,

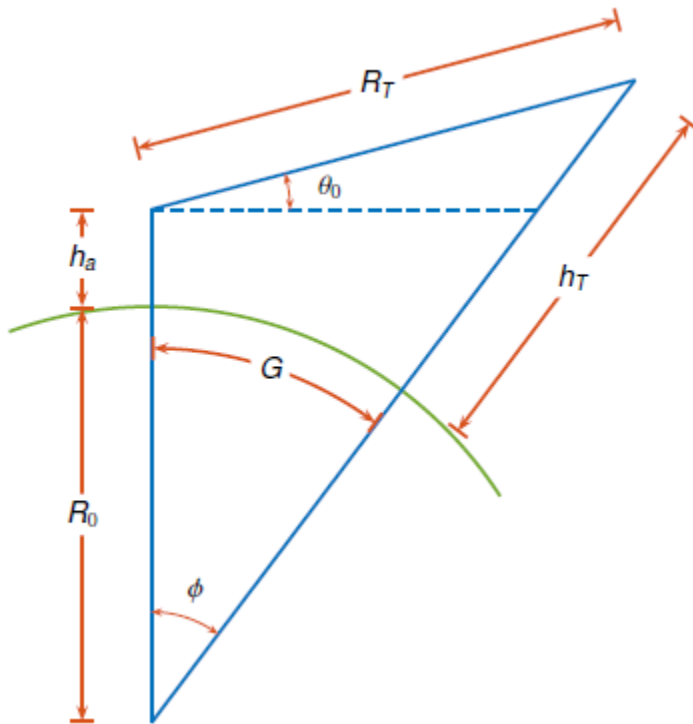
$$h_T = \sqrt{(R_0 + h_a)^2 + R_T^2 + 2R_T(R_0 + h_a) \sin \theta_0} - R_0.$$



The actual range  $R$  is equal to the slant range. The true elevation angle  $\theta_T$  is equal to the initial elevation angle.

To compute the ground range  $G$ , use

$$G = R_0\phi = R_0\arcsin\frac{R_T\cos\theta_0}{R_0 + h_T}.$$



A standard propagation model uses an effective Earth's radius that is 4/3 times the actual value. This model has two major limitations:

- 1 The model implies a value for the index of refraction near the Earth's surface that is valid only for certain areas and at certain times of the year. To mitigate this limitation, use an effective Earth's radius based on the near-surface refractivity value.
- 2 The model implies a value for the gradient of the index of refraction that is unrealistically low at heights of around 8 km. To partially mitigate this limitation, use an effective Earth's radius based on the platform altitudes.

For more information, see `effearthradius`.

### CRPL Exponential Reference Atmosphere Model

Atmospheric refraction evidences itself as a deviation in an electromagnetic ray from a straight line due to variation in air density as a function of height. The Central Radio Propagation Laboratory (CRPL) exponential reference atmosphere model treats refraction effects by assuming that the index of refraction  $n(h)$  and the refractivity  $N$  decay exponentially with height. The model defines

$$N = (n(h) - 1) \times 10^6 = N_s e^{-R_{\text{exp}}h},$$

where  $N_s$  is the atmospheric refractivity value (in units of  $10^{-6}$ ) at the surface of the earth,  $R_{\text{exp}}$  is the decay constant, and  $h$  is the height above the surface in kilometers. Thus

$$n(h) = 1 + (N_s \times 10^{-6})e^{-R_{\text{exp}}h}.$$

The default value of  $N_s$  is 313 N-units and can be modified using the `SurfaceRefractivity` name-value argument in functions that accept it. The default value of  $R_{\text{exp}}$  is  $0.143859 \text{ km}^{-1}$  and can be modified using the `RefractionExponent` name-value argument in functions that accept it.

### CRPL Model Geometry

When the refractivity of air is incorporated into the curved Earth model, the ray paths do not follow a straight line but curve downward. (This statement assumes standard atmospheric propagation and nonnegative elevation angles.) The true elevation angle  $\theta_T$  is different from the initial  $\theta_0$ . The actual range  $R$ , which is the distance along the curved path  $R'$ , is different from the slant range  $R_T$ .

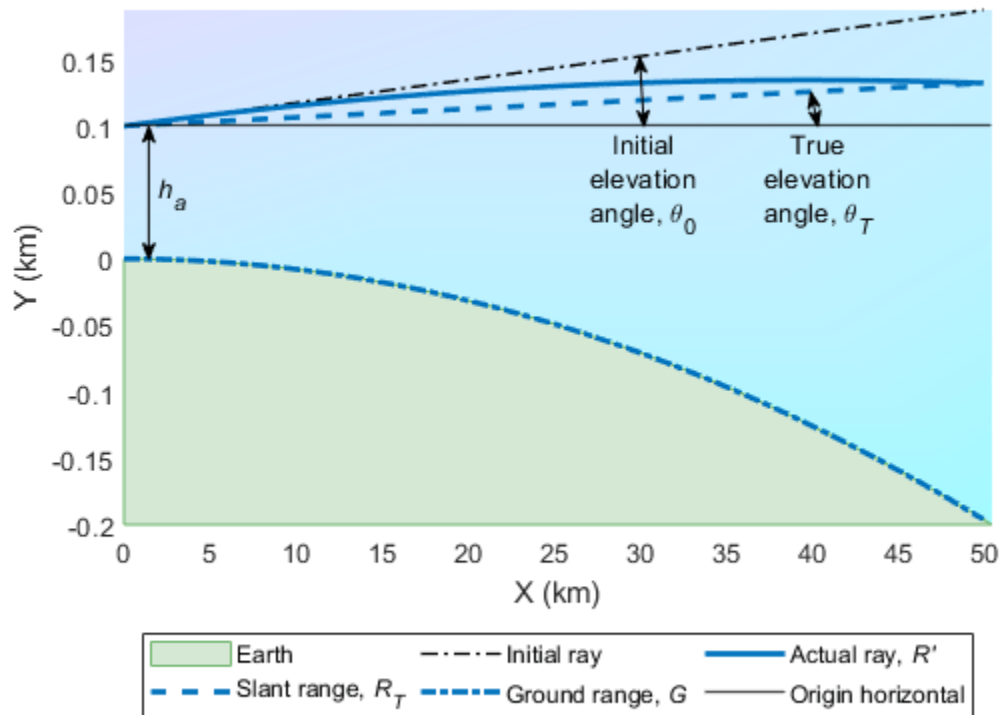
Given the Earth's radius  $R_0$ , the antenna height  $h_a$ , the initial elevation angle  $\theta_0$ , and the height-dependent index of refraction  $n(h)$  with value  $n_0$  at  $h = 0$ , the modified model relates the target height  $h_T$  and the actual range  $R$  by

$$R = \int_0^{h_T-h_a} n(h) dh \left( 1 - \left( \frac{n_0 \cos \theta_0}{n(h) \left( 1 + \frac{h}{R_0 + h_a} \right)} \right)^2 \right)^{-1/2}.$$

When `Method` is specified as "CRPL", the integral is solved using  $n(h)$  from "CRPL Exponential Reference Atmosphere Model" on page 1-93.

To compute the ground range  $G$ , use

$$G = \int_0^{h_T-h_a} \frac{dh}{1 + \frac{h}{R_0 + h_a}} \left( \left( \frac{n(h) \left( 1 + \frac{h}{R_0 + h_a} \right)}{n_0 \cos \theta_0} \right)^2 - 1 \right)^{-1/2}.$$



## References

- [1] Barton, David K. *Radar Equations for Modern Radar*. Norwood, MA: Artech House, 2013.
- [2] Bean, B.R., and G.D. Thayer. "Central Radio Propagation Laboratory Exponential Reference Atmosphere." *Journal of Research of the National Bureau of Standards, Section D: Radio Propagation* 63D, no. 3 (November 1959): 315. <https://doi.org/10.6028/jres.063D.031>.
- [3] Blake, Lamont V. "Ray Height Computation for a Continuous Nonlinear Atmospheric Refractive-Index Profile." *Radio Science* 3, no. 1 (January 1968): 85-92. <https://doi.org/10.1002/rds19683185>.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Apps

Radar Designer

**Functions**

blakechart | el2height | height2el | height2range | height2grndrange | radarvcd |  
refractionexp

**Topics**

“Radar Vertical Coverage over Terrain”

“Modeling Target Position Errors due to Refraction”

**Introduced in R2021b**

# rcscylinder

Radar cross section of cylinder

## Syntax

```
rcspat = rcscylinder(r1,r2,height,c,fc)
rcspat = rcscylinder(r1,r2,height,c,fc,az,el)
[rcspat,azout,elout] = rcscylinder( ___ )
```

## Description

`rcspat = rcscylinder(r1,r2,height,c,fc)` returns the radar cross section pattern of an elliptical cylinder having a semi-major axis, `r1`, a semi-minor axis, `r2`, and a height, `height`. The radar cross section is a function of signal frequency, `fc`, and signal propagation speed, `c`. The bottom of the cylinder lies on the `xy`-plane. The height of the cylinder points along the positive `z`-axis.

`rcspat = rcscylinder(r1,r2,height,c,fc,az,el)` also specifies the azimuth angles, `az`, and elevation angles, `el`, at which to compute the radar cross section.

`[rcspat,azout,elout] = rcscylinder( ___ )` also returns the azimuth angles, `azout`, and elevation angles, `elout`, at which the radar cross sections are computed. You can use these output arguments with any of the previous syntaxes.

## Examples

### Radar Cross Section of Elliptical Cylinder

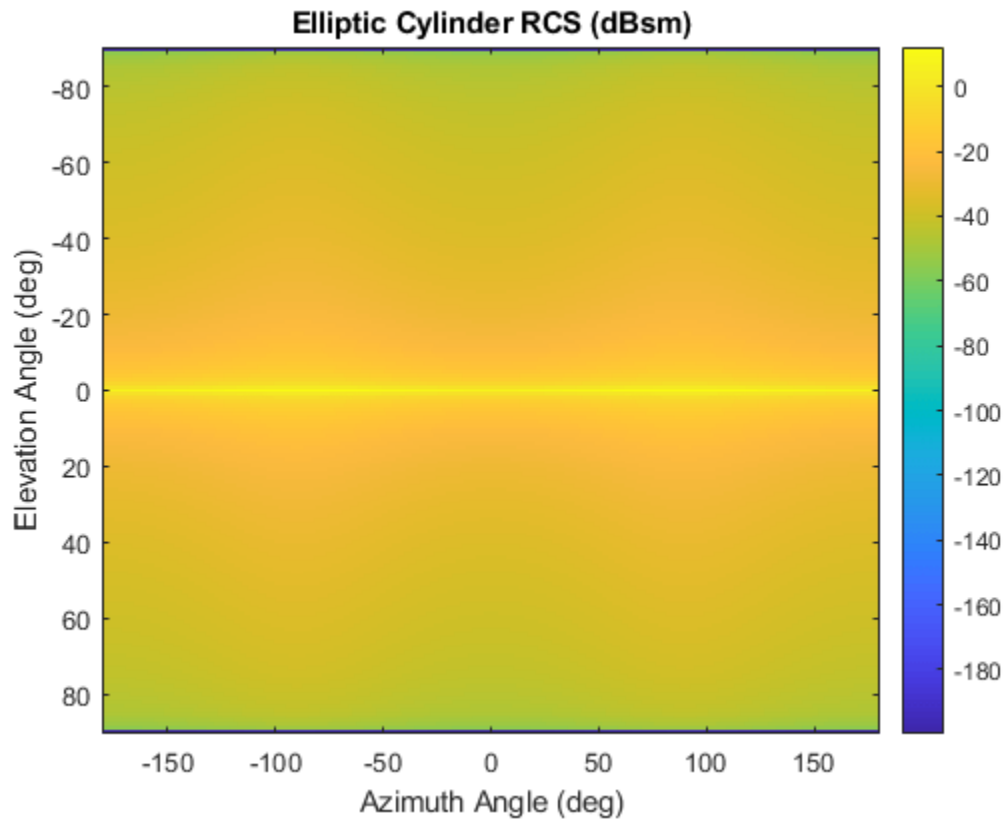
Display the radar cross section (RCS) pattern as a function of azimuth and elevation for an elliptical cylinder whose semi-major axis is 12.5 cm and whose semi-minor axis is 9 cm. The cylinder height is 1 m. The operating frequency is 4.5 GHz.

Specify the cylinder geometry and signal parameters.

```
c = physconst('Lightspeed');
fc = 4.5e9;
rada = 0.125;
radb = 0.090;
hgt = 1;
```

Compute the RCS for all directions using the default direction values.

```
[rcspat,azresp,elresp] = rcscylinder(rada,radb,hgt,c,fc);
imagesc(azresp,elresp,pow2db(rcspat))
colorbar
xlabel('Azimuth Angle (deg)')
ylabel('Elevation Angle (deg)')
title('Elliptic Cylinder RCS (dBsm)')
```



### Radar Cross Section of Elliptical Cylinder as Function of Elevation

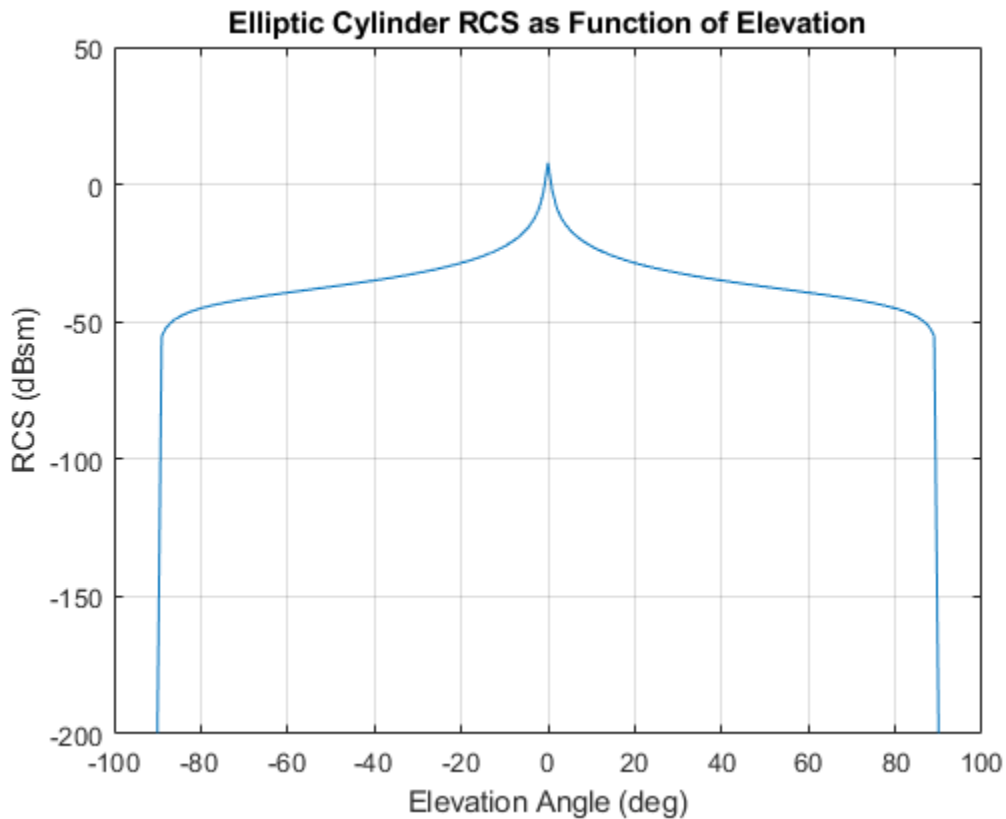
Plot the radar cross section (RCS) pattern of an elliptical cylinder as a function of elevation at a constant azimuth angle of  $5^\circ$ . The cylinder has a semi-major axis of 12.5 cm and a semi-minor axis of 9 cm. The cylinder height is 1 m. The operating frequency is 4.5 GHz.

Specify the cylinder geometry and signal parameters.

```
c = physconst('Lightspeed');
fc = 4.5e9;
rada = 0.125;
radb = 0.090;
hgt = 1;
```

Compute the RCS for all elevation angles at a fixed azimuth angle of  $5^\circ$ .

```
el = -90:90;
az = 5;
[rcspat,azresp,elresp] = rcscylinder(rada,radb,hgt,c,fc,az,el);
plot(elresp,pow2db(rcspat))
xlabel('Elevation Angle (deg)')
ylabel('RCS (dBsm)')
title('Elliptic Cylinder RCS as Function of Elevation')
grid on
```



### Radar Cross Section of Elliptical Cylinder as Function of Frequency

Plot the radar cross section (RCS) of an elliptical cylinder as a function of frequency for a fixed direction. The cylinder has a semi-major axis of 12.5 cm and a semi-minor axis of 9 cm. The cylinder height is 1 m.

Specify the cylinder geometry and signal parameters.

```
c = physconst('Lightspeed');
rada = 0.125;
radb = 0.090;
hgt = 1;
```

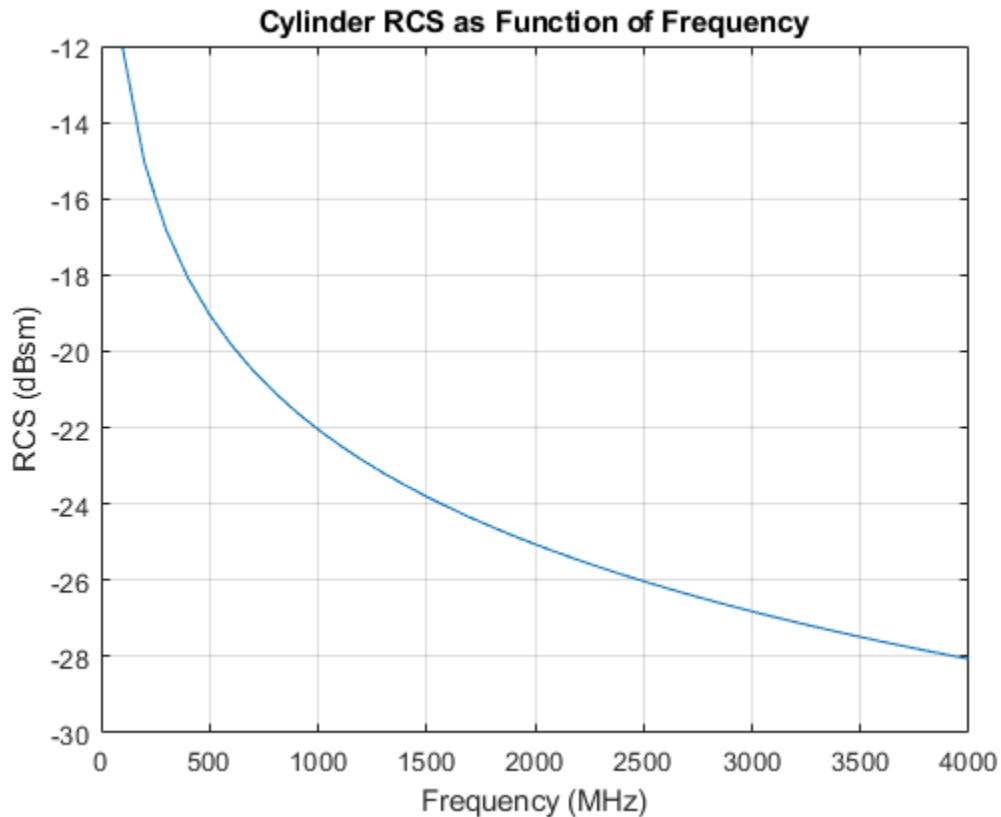
Compute radar cross sections as a function of frequency for a fixed azimuth and elevation.

```
az = 5.0;
el = 20.0;
fc = (100:100:4000)*1e6;
[rcspat,azpat,elpat] = rcscylinder(rada,radb,hgt,c,fc,az,el);
disp([azpat,elpat])
```

```
5    20
```

```
plot(fc/1e6,pow2db(squeeze(rcspat)))
xlabel('Frequency (MHz)')
```

```
ylabel('RCS (dBsm)')  
title('Cylinder RCS as Function of Frequency')  
grid on
```



## Input Arguments

### **r1** — Length of semi-major axis of cylinder

positive scalar

Length of semi-major axis of cylinder, specified as a positive scalar. Units are in meters.

Example: 5.5

Data Types: double

### **r2** — Length of semi-minor axis of cylinder

positive scalar

Length of semi-minor axis of cylinder, specified as a positive scalar. Units are in meters.

Example: 3.0

Data Types: double

### **height** — Height of cylinder

positive scalar



Height of cylinder, specified as a positive scalar. Units are in meters.

Example: 3.0

Data Types: double

### **c** – Signal propagation speed

positive scalar

Signal propagation speed, specified as a positive scalar. Units are in meters per second. For the SI value of the speed of light, use `physconst('LightSpeed')`.

Example: 3e8

Data Types: double

### **fc** – Frequency for computing radar cross section

positive scalar | positive, real-valued, 1-by- $L$  row vector

Frequency for computing radar cross section, specified as a positive scalar or positive, real-valued, 1-by- $L$  row vector. Frequency units are in Hz.

Example: [100e6 200e6]

Data Types: double

### **az** – Azimuth angles

-180:180 (default) | 1-by- $M$  real-valued row vector

Azimuth angles for computing directivity and pattern, specified as a real-valued 1-by- $M$  row vector where  $M$  is the number of azimuth angles. Angle units are in degrees. Azimuth angles must lie between  $-180^\circ$  and  $180^\circ$ , inclusive.

The azimuth angle is the angle between the  $x$ -axis and the projection of a direction vector onto the  $xy$ -plane. The azimuth angle is positive when measured from the  $x$ -axis toward the  $y$ -axis.

Example: -45:2:45

Data Types: double

### **e1** – Elevation angles

-90:90 (default) | 1-by- $N$  real-valued row vector

Elevation angles for computing directivity and pattern, specified as a real-valued, 1-by- $N$  row vector where  $N$  is the number of desired elevation directions. Angle units are in degrees. Elevation angles must lie between  $-90^\circ$  and  $90^\circ$ , inclusive.

The elevation angle is the angle between a direction vector and  $xy$ -plane. The elevation angle is positive when measured towards the  $z$ -axis.

Example: -75:1:70

Data Types: double

---

**Tip** To construct a circular cylinder, set `r2` equal to `r1`.

---

## Output Arguments

### **rcspat** — Radar cross section pattern

real-valued  $N$ -by- $M$ -by- $L$  array

Radar cross section pattern, returned as a real-valued  $N$ -by- $M$ -by- $L$  array.  $N$  is the length of the vector returned in the `elout` argument.  $M$  is the length of the vector returned in the `azout` argument.  $L$  is the length of the `fc` vector. Units are in meters-squared.

Data Types: `double`

### **azout** — Azimuth angles

real-valued 1-by- $M$  row vector

Azimuth angles for computing directivity and pattern, returned as a real-valued 1-by- $M$  row vector where  $M$  is the number of azimuth angles specified by the `az` input argument. Angle units are in degrees.

The azimuth angle is the angle between the  $x$ -axis and the projection of the direction vector onto the  $xy$ -plane. The azimuth angle is positive when measured from the  $x$ -axis toward the  $y$ -axis.

Data Types: `double`

### **elout** — Elevation angles

real-valued 1-by- $N$  row vector

Elevation angles for computing directivity and pattern, returned as a real-valued 1-by- $N$  row vector where  $N$  is the number of elevation angles specified in `el` output argument. Angle units are in degrees.

The elevation angle is the angle between the direction vector and  $xy$ -plane. The elevation angle is positive when measured towards the  $z$ -axis.

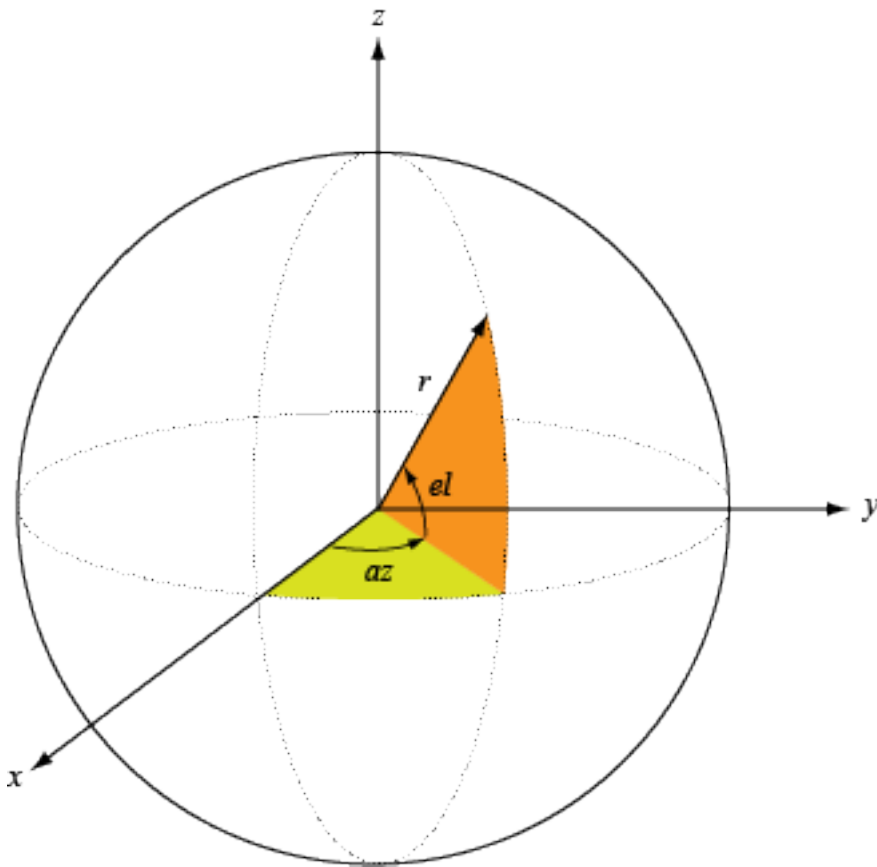
Data Types: `double`

## More About

### **Azimuth and Elevation**

This section describes the convention used to define azimuth and elevation angles.

The azimuth angle of a vector is the angle between the  $x$ -axis and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going from the  $x$ -axis toward the  $y$ -axis. Azimuth angles lie between  $-180^\circ$  and  $180^\circ$  degrees, inclusive. The elevation angle is the angle between the vector and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going toward the positive  $z$ -axis from the  $xy$ -plane. Elevation angles lie between  $-90^\circ$  and  $90^\circ$  degrees, inclusive.



## References

- [1] Mahafza, Bassem. *Radar Systems Analysis and Design Using MATLAB, 2nd Ed.* Boca Raton, FL: Chapman & Hall/CRC, 2005.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`rcsdisc` | `rcsphere` | `rcstruncone` | `phased.BackscatterRadarTarget` | `phased.RadarTarget`

**Introduced in R2021a**

## rscdisc

Radar cross section of flat circular plate

### Syntax

```
rcspat = rscdisc(r,c,fc)
rcspat = rscdisc(r,c,fc,az,el)
[rcspat,azout,elout] = rscdisc( ___ )
```

### Description

`rcspat = rscdisc(r,c,fc)` returns the radar cross section pattern of a flat circular plate of radius  $r$ . The radar cross section is a function of signal frequency,  $fc$ , and signal propagation speed,  $c$ . The plate is assumed to lie on the  $xy$ -plane. The center of the plate is located at the origin of the local coordinate system.

`rcspat = rscdisc(r,c,fc,az,el)` also specifies the azimuth angles,  $az$ , and elevation angles,  $el$ , at which to compute the radar cross section.

`[rcspat,azout,elout] = rscdisc( ___ )` also returns the azimuth angles,  $azout$ , and elevation angles,  $elout$ , at which the radar cross sections are computed. You can use these output arguments with any of the previous syntaxes.

### Examples

#### Radar Cross Section of Circular Plate

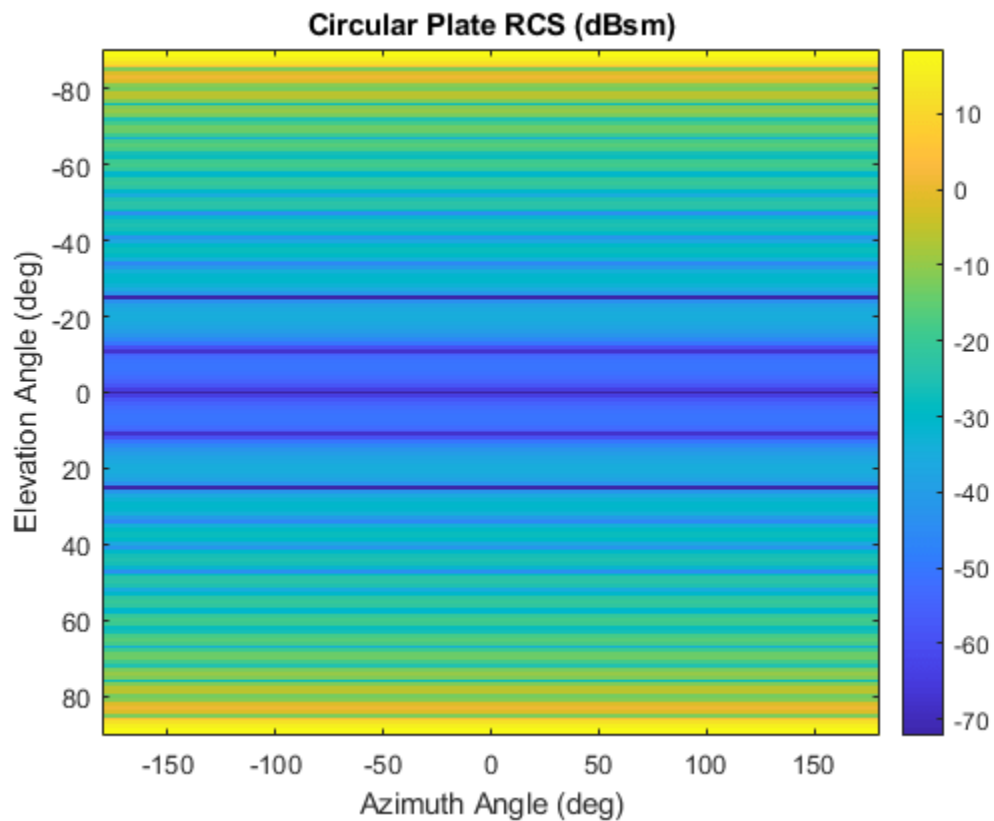
Display the radar cross section (RCS) pattern of a circular plate as a function of azimuth and elevation. The plate radius is 22.5 cm. The operating frequency is 4.5 GHz.

Specify the plate geometry and signal parameters.

```
c = physconst('Lightspeed');
fc = 4.5e9;
platerad = 0.225;
```

Compute the RCS for all directions using the default direction values.

```
[rcspat,azresp,elresp] = rscdisc(platerad,c,fc);
imagesc(azresp,elresp,pow2db(rcspat))
colorbar
xlabel('Azimuth Angle (deg)')
ylabel('Elevation Angle (deg)')
title('Circular Plate RCS (dBsm)')
```



### Radar Cross Section of Circular Plate as Function of Elevation

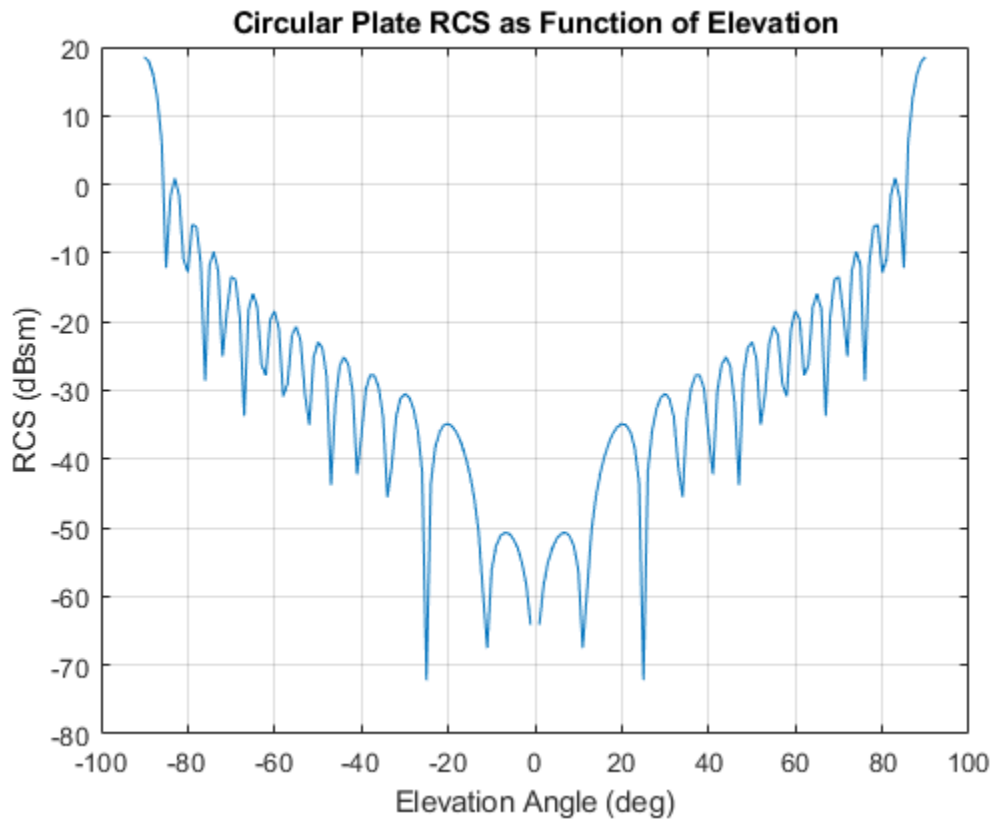
Plot the radar cross section (RCS) pattern of a circular plate as a function of elevation angle for a fixed azimuth angle of  $5^\circ$ . The plate radius is 22.5 cm. The operating frequency is 4.5 GHz.

Define the plate radius and signal parameters.

```
c = physconst('Lightspeed');
fc = 4.5e9;
platerad = 0.225;
```

Compute the RCS as a function of elevation.

```
az = 5;
el = -90:90;
[rcspat,azresp,elresp] = rcsdisc(platerad,c,fc,az,el);
plot(elresp,pow2db(rcspat))
xlabel('Elevation Angle (deg)')
ylabel('RCS (dBsm)')
title('Circular Plate RCS as Function of Elevation')
grid on
```



### Radar Cross Section of Circular Plate as Function of Frequency

Plot the radar cross section (RCS) pattern of a circular plate as a function of frequency for a single azimuth and elevation. The plate radius 22.5 cm.

Define the plate radius and signal parameters.

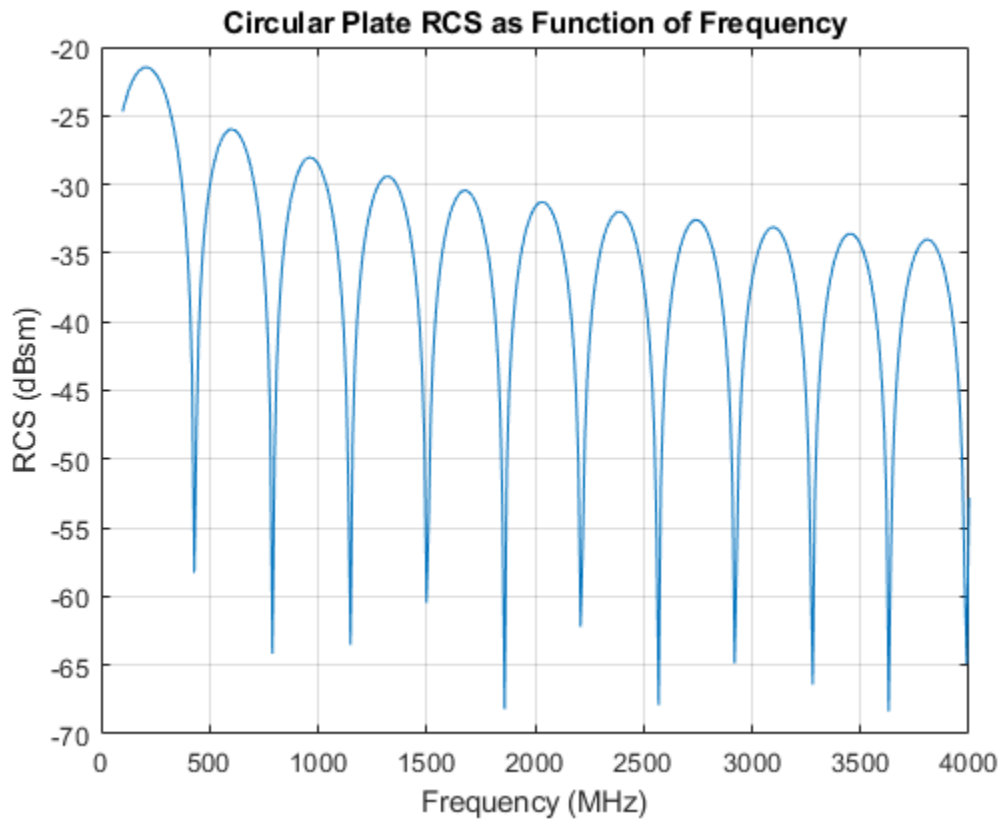
```
c = physconst('Lightspeed');
platerad = 0.225;
```

Compute the RCS over a range of frequencies for a single direction.

```
az = 5.0;
el = 20.0;
fc = (100:10:4000)*1e6;
[rcspat,azpat,elpat] = rcsdisc(platerad,c,fc,az,el);
disp([azpat,elpat])
```

```
5    20
```

```
plot(fc/1e6,pow2db(squeeze(rcspat)))
xlabel('Frequency (MHz)')
ylabel('RCS (dBsm)')
title('Circular Plate RCS as Function of Frequency')
grid on
```



## Input Arguments

### **r** — Radius of circular plate

positive scalar

Radius of circular plate, specified as a positive scalar. Units are in meters.

Example: 5.5

Data Types: double

### **c** — Signal propagation speed

positive scalar

Signal propagation speed, specified as a positive scalar. Units are in meters per second. For the SI value of the speed of light, use `physconst('LightSpeed')`.

Example: 3e8

Data Types: double

### **fc** — Frequency for computing radar cross section

positive scalar | positive, real-valued, 1-by-*L* row vector

Frequency for computing radar cross section, specified as a positive scalar or positive, real-valued, 1-by-*L* row vector. Frequency units are in Hz.

Example: [100e6 200e6]

Data Types: double

### **az** — Azimuth angles

-180:180 (default) | 1-by- $M$  real-valued row vector

Azimuth angles for computing directivity and pattern, specified as a real-valued 1-by- $M$  row vector where  $M$  is the number of azimuth angles. Angle units are in degrees. Azimuth angles must lie between  $-180^\circ$  and  $180^\circ$ , inclusive.

The azimuth angle is the angle between the  $x$ -axis and the projection of a direction vector onto the  $xy$ -plane. The azimuth angle is positive when measured from the  $x$ -axis toward the  $y$ -axis.

Example: -45:2:45

Data Types: double

### **e1** — Elevation angles

-90:90 (default) | 1-by- $N$  real-valued row vector

Elevation angles for computing directivity and pattern, specified as a real-valued, 1-by- $N$  row vector where  $N$  is the number of desired elevation directions. Angle units are in degrees. Elevation angles must lie between  $-90^\circ$  and  $90^\circ$ , inclusive.

The elevation angle is the angle between a direction vector and  $xy$ -plane. The elevation angle is positive when measured towards the  $z$ -axis.

Example: -75:1:70

Data Types: double

## **Output Arguments**

### **rcspat** — Radar cross section pattern

real-valued  $N$ -by- $M$ -by- $L$  array

Radar cross section pattern, returned as a real-valued  $N$ -by- $M$ -by- $L$  array.  $N$  is the length of the vector returned in the `e1out` argument.  $M$  is the length of the vector returned in the `azout` argument.  $L$  is the length of the `fc` vector. Units are in meters-squared.

Data Types: double

### **azout** — Azimuth angles

real-valued 1-by- $M$  row vector

Azimuth angles for computing directivity and pattern, returned as a real-valued 1-by- $M$  row vector where  $M$  is the number of azimuth angles specified by the `az` input argument. Angle units are in degrees.

The azimuth angle is the angle between the  $x$ -axis and the projection of the direction vector onto the  $xy$ -plane. The azimuth angle is positive when measured from the  $x$ -axis toward the  $y$ -axis.

Data Types: double

### **e1out** — Elevation angles

real-valued 1-by- $N$  row vector



Elevation angles for computing directivity and pattern, returned as a real-valued 1-by- $N$  row vector where  $N$  is the number of elevation angles specified in `el` output argument. Angle units are in degrees.

The elevation angle is the angle between the direction vector and  $xy$ -plane. The elevation angle is positive when measured towards the  $z$ -axis.

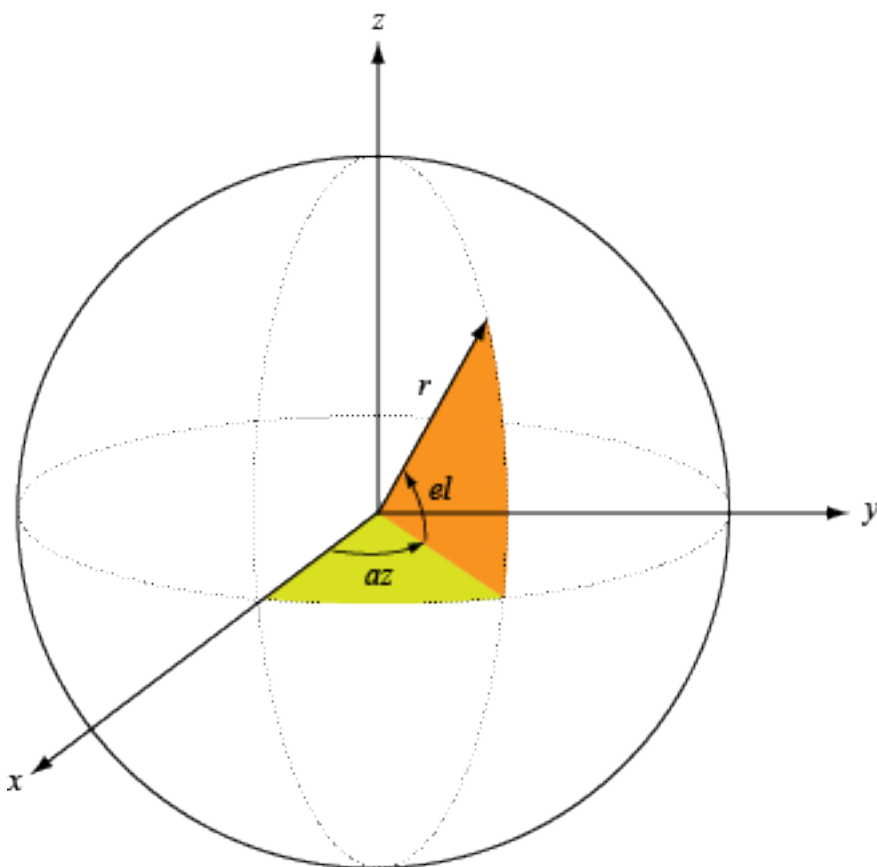
Data Types: `double`

## More About

### Azimuth and Elevation

This section describes the convention used to define azimuth and elevation angles.

The azimuth angle of a vector is the angle between the  $x$ -axis and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going from the  $x$ -axis toward the  $y$ -axis. Azimuth angles lie between  $-180^\circ$  and  $180^\circ$  degrees, inclusive. The elevation angle is the angle between the vector and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going toward the positive  $z$ -axis from the  $xy$ -plane. Elevation angles lie between  $-90^\circ$  and  $90^\circ$  degrees, inclusive.



## References

- [1] Mahafza, Bassem. *Radar Systems Analysis and Design Using MATLAB, 2nd Ed.* Boca Raton, FL: Chapman & Hall/CRC, 2005.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`rcscylinder` | `rcsphere` | `rcstruncone` | `phased.BackscatterRadarTarget` | `phased.RadarTarget`

**Introduced in R2021a**

# rcssphere

Radar cross section of sphere

## Syntax

```
rcspat = rcssphere(r,c,fc)
rcspat = rcssphere(r,c,fc,az,el)
[rcspat,azout,elout] = rcssphere( ___ )
```

## Description

`rcspat = rcssphere(r,c,fc)` returns the radar cross section pattern of a sphere of radius `r` as a function of signal frequency, `fc`, and signal propagation speed, `c`. The center of the sphere is assumed to be located at the origin of the local coordinate system.

`rcspat = rcssphere(r,c,fc,az,el)` also specifies the azimuth angles, `az`, and elevation angles, `el`, at which to compute the radar cross section.

`[rcspat,azout,elout] = rcssphere( ___ )` also returns the azimuth angles, `azout`, and elevation angles, `elout`, at which the radar cross sections are computed. You can use these output arguments with any of the previous syntaxes.

## Examples

### Radar Cross Section of Sphere

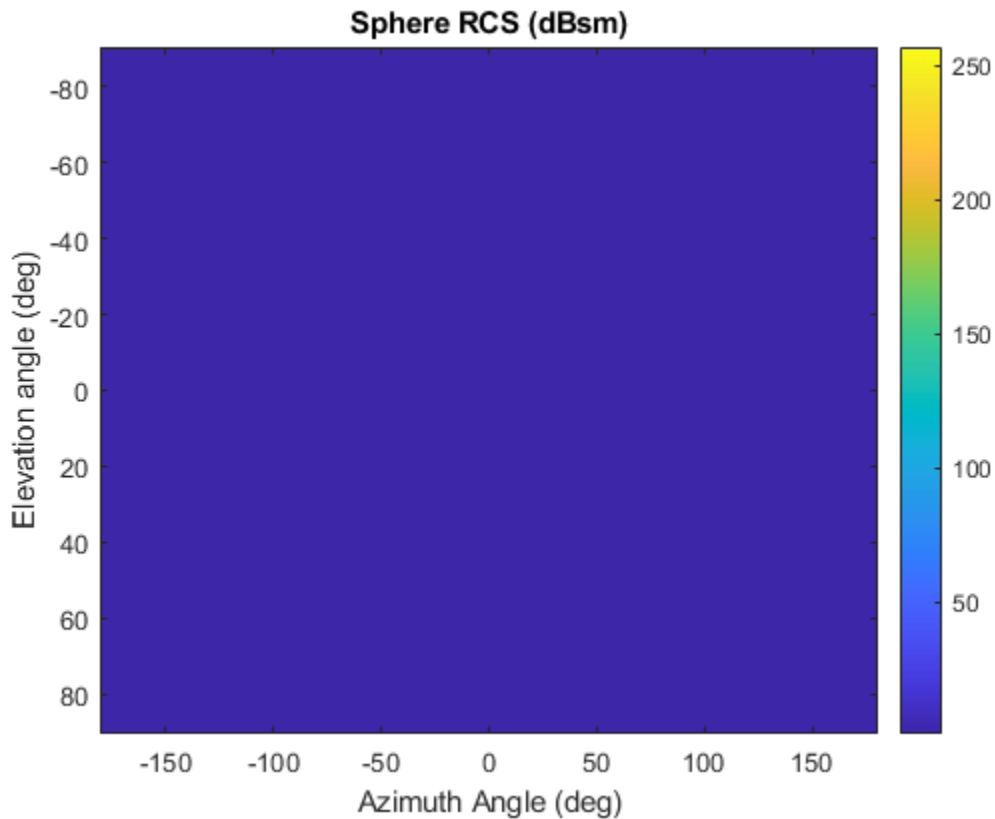
Display the radar cross section (RCS) pattern of a sphere as a function of azimuth and elevation. The sphere radius is 20.0 cm. The operating frequency is 4.5 GHz.

Define the sphere radius and signal parameters.

```
c = physconst('Lightspeed');
fc = 4.5e9;
rad = 0.20;
```

Compute the RCS over all angles. The image shows that the RCS is constant over all directions.

```
[rcspat,azresp,elresp] = rcssphere(rad,c,fc);
image(azresp,elresp,pow2db(rcspat))
colorbar
ylabel('Elevation angle (deg)')
xlabel('Azimuth Angle (deg)')
title('Sphere RCS (dBsm)')
```



### Radar Cross Section of Sphere as Function of Elevation

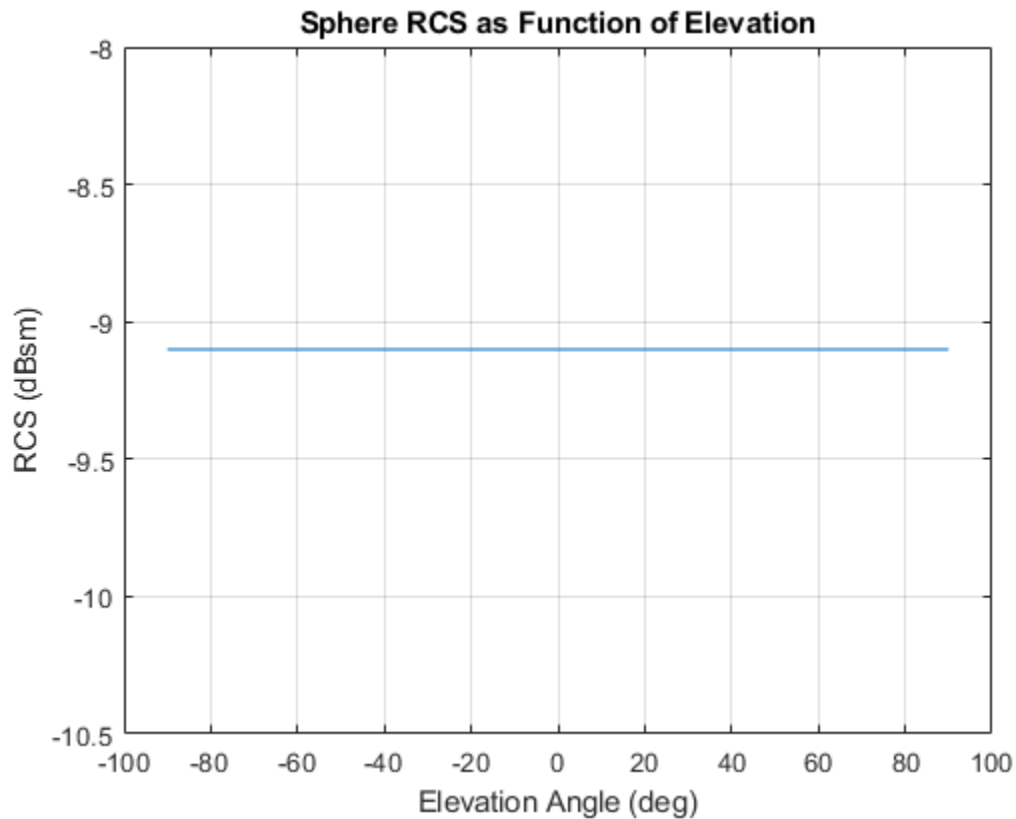
Plot the radar cross section (RCS) pattern of a sphere as a function of elevation angle for a fixed azimuth angle of 5 degrees. The sphere radius is 20.0 cm. The operating frequency is 4.5 GHz.

Specify the sphere radius and signal parameters.

```
c = physconst('LightSpeed');
rad = 0.20;
fc = 4.5e9;
```

Compute the RCS over a constant azimuth slice. The plot shows that the RCS is constant.

```
az = 5.0;
el = -90:90;
[rcspat,azresp,elresp] = rcssphere(rad,c,fc,az,el);
plot(elresp,pow2db(rcspat))
xlabel('Elevation Angle (deg)')
ylabel('RCS (dBsm)')
title('Sphere RCS as Function of Elevation')
grid on
```



### Radar Cross Section of Sphere as Function of Frequency

Plot the radar cross section (RCS) pattern of a sphere as a function of frequency for a single azimuth and elevation. The radius of the sphere is 20 cm

Define the sphere radius and signal parameters.

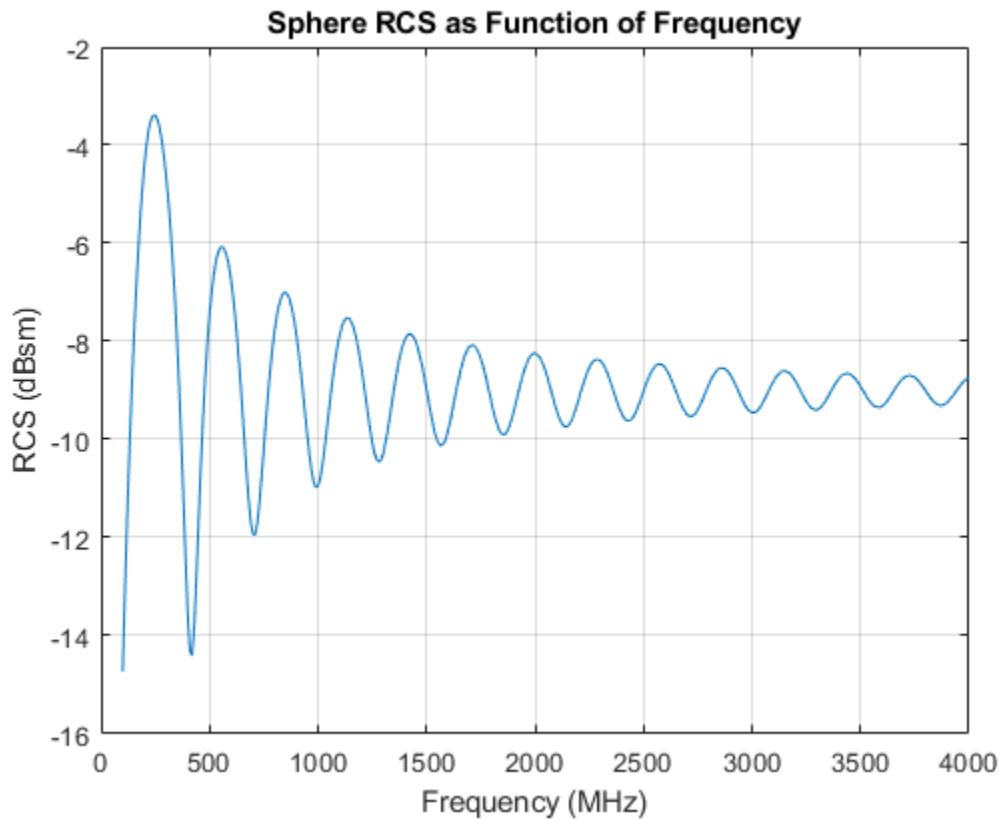
```
c = physconst('Lightspeed');
rad = 0.20;
```

Compute the RCS over a range of frequencies for a single direction.

```
az = 5.0;
el = 20.0;
fc = (100:10:4000)*1e6;
[rcspat,azpat,elpat] = rcsphere(rad,c,fc,az,el);
disp([azpat,elpat])
```

```
5    20
```

```
plot(fc/1e6,pow2db(squeeze(rcspat)))
xlabel('Frequency (MHz)')
ylabel('RCS (dBsm)')
title('Sphere RCS as Function of Frequency')
grid on
```



## Input Arguments

### **r** — Radius of sphere

positive scalar

Radius of sphere, specified as a positive scalar. Units are in meters.

Example: 5.5

Data Types: double

### **c** — Signal propagation speed

positive scalar

Signal propagation speed, specified as a positive scalar. Units are in meters per second. For the SI value of the speed of light, use `physconst('LightSpeed')`.

Example: 3e8

Data Types: double

### **fc** — Frequency for computing radar cross section

positive scalar | positive, real-valued, 1-by-*L* row vector

Frequency for computing radar cross section, specified as a positive scalar or positive, real-valued, 1-by-*L* row vector. Frequency units are in Hz.

Example: [100e6 200e6]

Data Types: double

### **az** — Azimuth angles

-180:180 (default) | 1-by- $M$  real-valued row vector

Azimuth angles for computing directivity and pattern, specified as a real-valued 1-by- $M$  row vector where  $M$  is the number of azimuth angles. Angle units are in degrees. Azimuth angles must lie between  $-180^\circ$  and  $180^\circ$ , inclusive.

The azimuth angle is the angle between the  $x$ -axis and the projection of a direction vector onto the  $xy$ -plane. The azimuth angle is positive when measured from the  $x$ -axis toward the  $y$ -axis.

Example: -45:2:45

Data Types: double

### **e1** — Elevation angles

-90:90 (default) | 1-by- $N$  real-valued row vector

Elevation angles for computing directivity and pattern, specified as a real-valued, 1-by- $N$  row vector where  $N$  is the number of desired elevation directions. Angle units are in degrees. Elevation angles must lie between  $-90^\circ$  and  $90^\circ$ , inclusive.

The elevation angle is the angle between a direction vector and  $xy$ -plane. The elevation angle is positive when measured towards the  $z$ -axis.

Example: -75:1:70

Data Types: double

## **Output Arguments**

### **rcsspat** — Radar cross section pattern

real-valued  $N$ -by- $M$ -by- $L$  array

Radar cross section pattern, returned as a real-valued  $N$ -by- $M$ -by- $L$  array.  $N$  is the length of the vector returned in the `e1out` argument.  $M$  is the length of the vector returned in the `azout` argument.  $L$  is the length of the `fc` vector. Units are in meters-squared.

Data Types: double

### **azout** — Azimuth angles

real-valued 1-by- $M$  row vector

Azimuth angles for computing directivity and pattern, returned as a real-valued 1-by- $M$  row vector where  $M$  is the number of azimuth angles specified by the `az` input argument. Angle units are in degrees.

The azimuth angle is the angle between the  $x$ -axis and the projection of the direction vector onto the  $xy$ -plane. The azimuth angle is positive when measured from the  $x$ -axis toward the  $y$ -axis.

Data Types: double

### **e1out** — Elevation angles

real-valued 1-by- $N$  row vector

Elevation angles for computing directivity and pattern, returned as a real-valued 1-by- $N$  row vector where  $N$  is the number of elevation angles specified in `el` output argument. Angle units are in degrees.

The elevation angle is the angle between the direction vector and  $xy$ -plane. The elevation angle is positive when measured towards the  $z$ -axis.

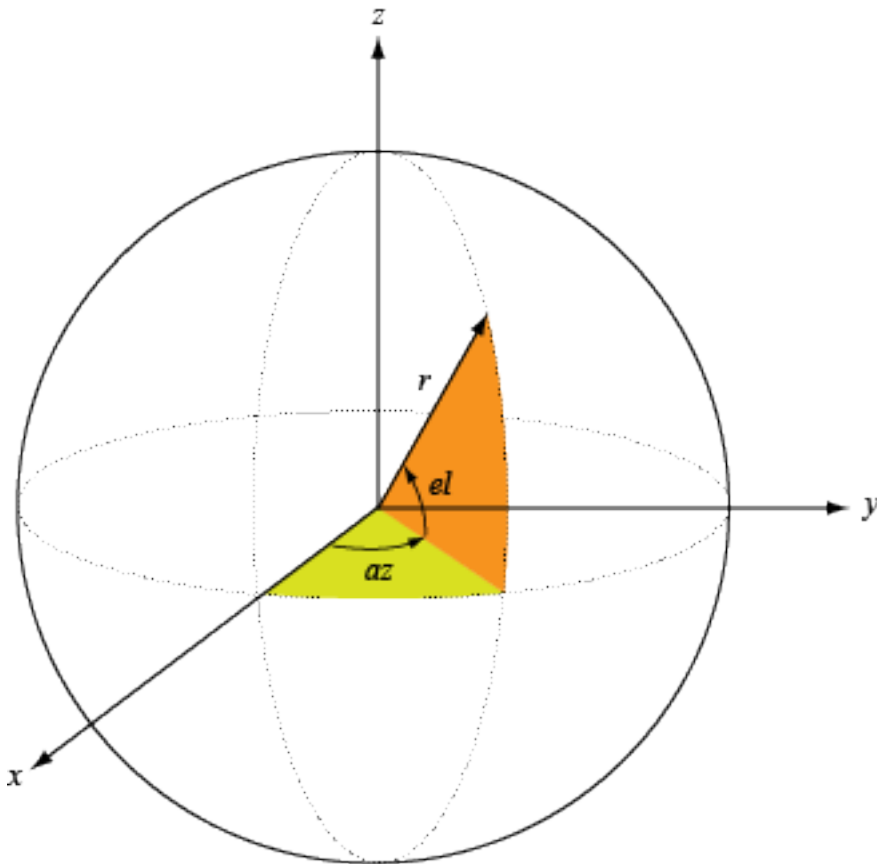
Data Types: `double`

## More About

### Azimuth and Elevation

This section describes the convention used to define azimuth and elevation angles.

The azimuth angle of a vector is the angle between the  $x$ -axis and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going from the  $x$ -axis toward the  $y$ -axis. Azimuth angles lie between  $-180^\circ$  and  $180^\circ$  degrees, inclusive. The elevation angle is the angle between the vector and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going toward the positive  $z$ -axis from the  $xy$ -plane. Elevation angles lie between  $-90^\circ$  and  $90^\circ$  degrees, inclusive.



## References

- [1] Mahafza, Bassem. *Radar Systems Analysis and Design Using MATLAB, 2nd Ed.* Boca Raton, FL: Chapman & Hall/CRC, 2005.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`rcscylinder` | `rcsdisc` | `rcstruncone` | `phased.BackscatterRadarTarget` | `phased.RadarTarget`

### Introduced in R2021a

## rcstruncone

Radar cross section of truncated cone

### Syntax

```
rcspat = rcstruncone(r1,r2,height,c,fc)
rcspat = rcstruncone(r1,r2,height,c,fc,az,el)
[rcspat,azout,elout] = rcstruncone(____)
```

### Description

`rcspat = rcstruncone(r1,r2,height,c,fc)` returns the radar cross section pattern of a truncated cone. `r1` is the radius of the small end of the cone, `r2` is the radius of the large end, and `height` is the cone height. The radar cross section is a function of signal frequency, `fc`, and signal propagation speed, `c`. You can create a non-truncated cone by setting `r1` to zero. The cone points downward towards the `xy`-plane. The origin is located at the apex of a the non-truncated cone constructed by extending the truncated cone to an apex.

`rcspat = rcstruncone(r1,r2,height,c,fc,az,el)` also specifies the azimuth angles, `az`, and elevation angles, `el`, at which to compute the radar cross section.

`[rcspat,azout,elout] = rcstruncone(____)` also returns the azimuth angles, `azout`, and elevation angles, `elout`, at which the radar cross sections are computed. You can use these output arguments with any of the previous syntaxes.

### Examples

#### Radar Cross Section of Truncated Cone

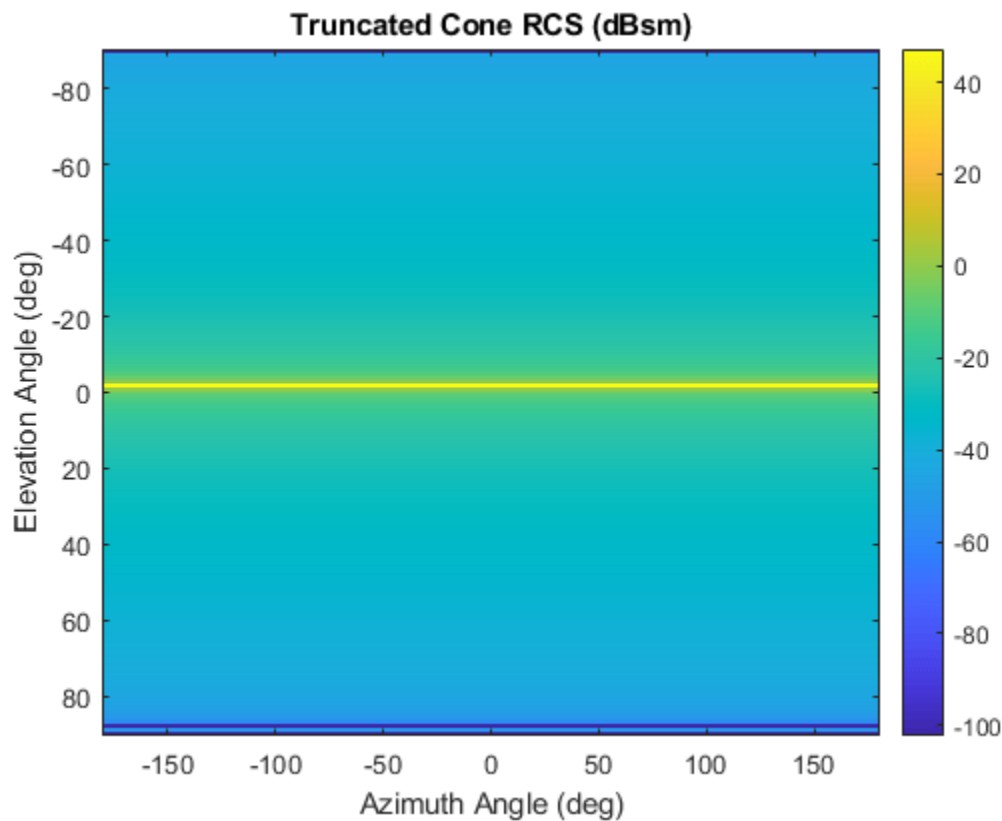
Display the radar cross section (RCS) pattern of a truncated cone as a function of azimuth angle and elevation. The truncated cone has a bottom radius of 9.0 cm and a top radius of 12.5 cm. The cone height is 1 m. The operating frequency is 4.5 GHz.

Define the truncated cone geometry and signal parameters.

```
c = physconst('Lightspeed');
fc = 4.5e9;
radbot = 0.090;
radtop = 0.125;
hgt = 1;
```

Compute the RCS for all directions using the default direction values.

```
[rcspat,azresp,elresp] = rcstruncone(radbot,radtop,hgt,c,fc);
imagesc(azresp,elresp,pow2db(rcspat))
xlabel('Azimuth Angle (deg)')
ylabel('Elevation Angle (deg)')
title('Truncated Cone RCS (dBsm)')
colorbar
```



### Radar Cross Section of Truncated Cone as Function of Elevation

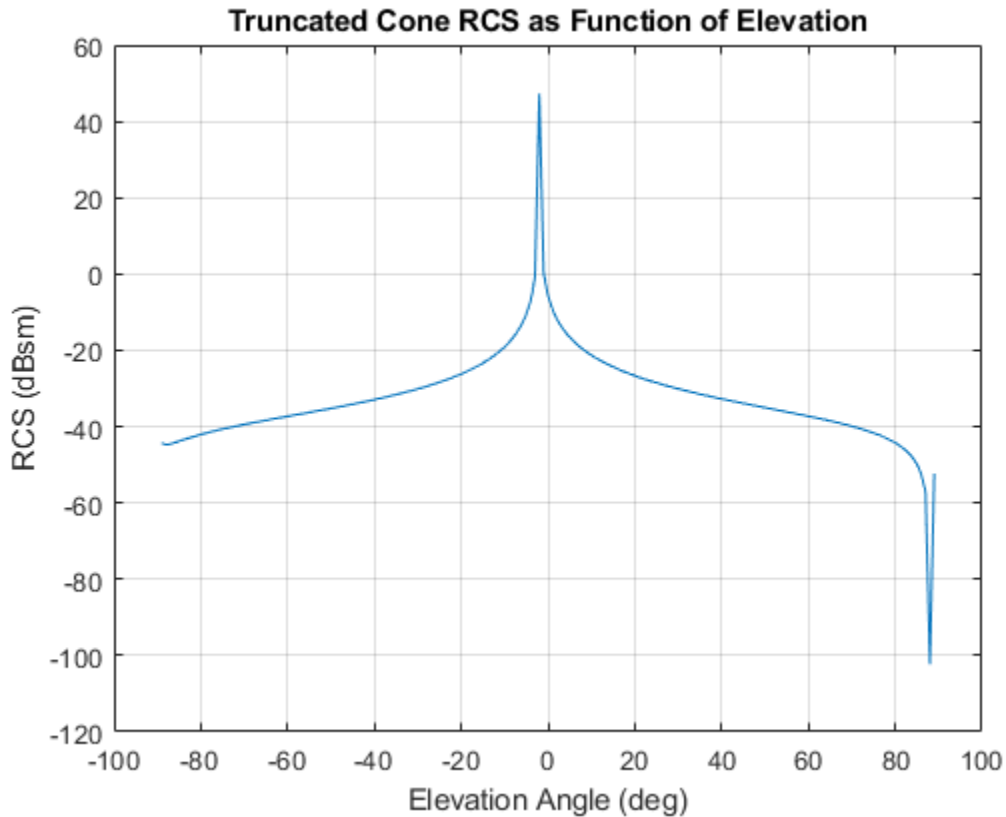
Plot the radar cross section (RCS) pattern of a truncated cone as a function of elevation for a fixed azimuth angle of 5 degrees. The cone has a bottom radius of 9.0 cm and a top radius of 12.5 cm. The truncated cone height is 1 m. The operating frequency is 4.5.

Define the truncated cone geometry and signal parameters.

```
c = physconst('Lightspeed');
fc = 4.5e9;
radbot = 0.090;
radtop = 0.125;
hgt = 1;
```

Compute the RCS at an azimuth angle of 5 degrees.

```
az = 5.0;
el = -90:90;
[rcspat,azresp,elresp] = rcstruncone(radbot,radtop,hgt,c,fc,az,el);
plot(elresp,pow2db(rcspat))
xlabel('Elevation Angle (deg)')
ylabel('RCS (dBsm)')
title('Truncated Cone RCS as Function of Elevation')
grid on
```



### Radar Cross Section of Truncated Cone as Function of Frequency

Plot the radar cross section (RCS) pattern of a truncated cone as a function of frequency for a single direction. The cone has a bottom radius of 9.0 cm and a top radius of 12.5 cm. The truncated cone height is 1 m.

Specify the truncated cone geometry and signal parameters.

```
c = physconst('Lightspeed');
radbot = 0.090;
radtop = 0.125;
hgt = 1;
```

Compute the RCS over a range of frequencies for a single direction.

```
az = 5.0;
el = 20.0;
fc = (100:100:4000)*1e6;
[rcspat,azpat,elpat] = rcstruncone(radbot,radtop,hgt,c,fc,az,el);
disp([azpat,elpat])
```

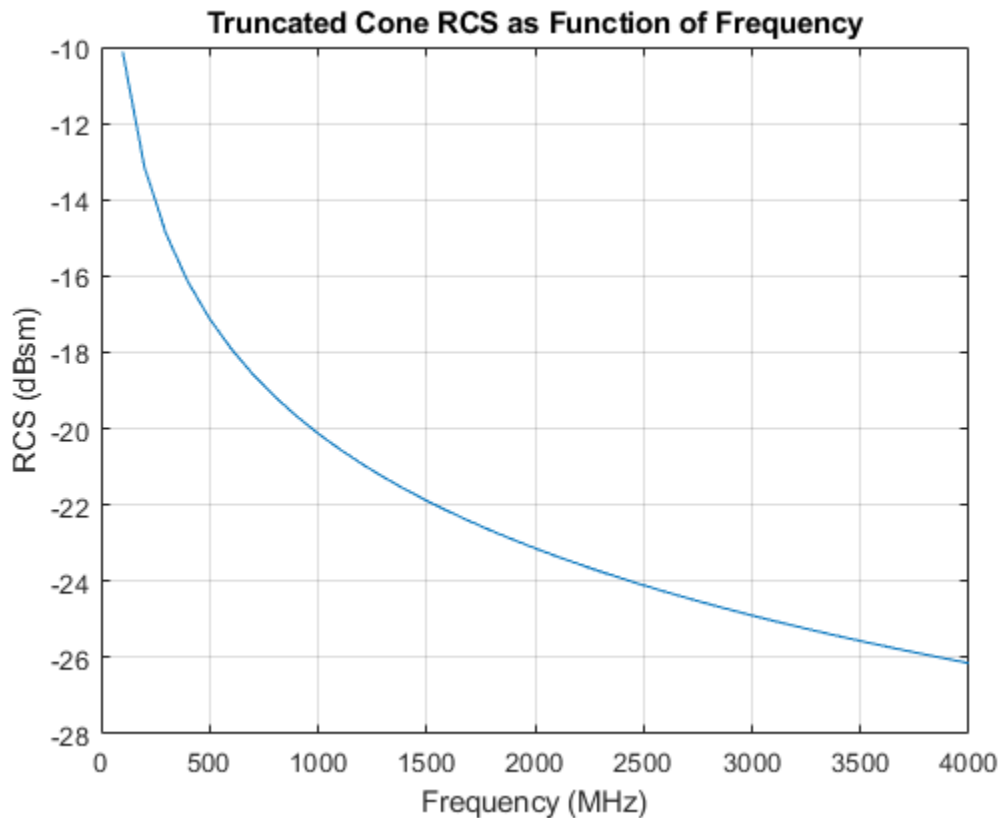
```
5    20
```

```
plot(fc/1e6,pow2db(squeeze(rcspat)))
xlabel('Frequency (MHz)')
```

```

ylabel('RCS (dBsm)')
title('Truncated Cone RCS as Function of Frequency')
grid on

```



### Radar Cross Section of Full Cone as Function of Elevation

Plot the radar cross section (RCS) pattern of a full cone as a function of elevation for a fixed azimuth angle. To define a full cone set the bottom radius to zero. Set the top radius to 20.0 cm and the cone height to 50 cm. Assume the operating frequency is 4.5 GHz and the azimuth angle is 5 degrees.

Define the cone geometry and signal parameters.

```

c = physconst('Lightspeed');
fc = 4.5e9;
radsmall = 0.0;
radlarge = 0.20;
hgt = 0.5;

```

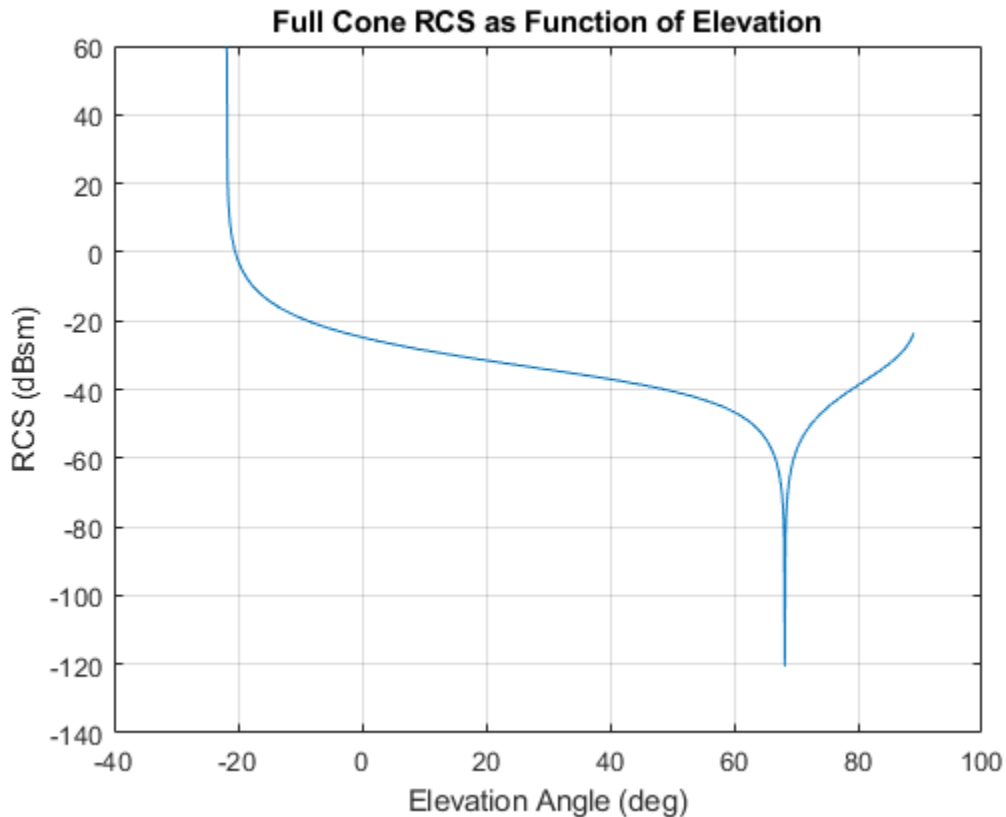
Compute the RCS for a fixed azimuth angle of 5 degrees.

```

az = 5.0;
el = -89:0.1:89;
[rcspat,azresp,elresp] = rcstruncone(radsmall,radlarge,hgt,c,fc,az,el);
plot(elresp,pow2db(rcspat))
xlabel('Elevation Angle (deg)')

```

```
ylabel('RCS (dBsm)')  
title('Full Cone RCS as Function of Elevation')  
grid on
```



## Input Arguments

### **r1** — Radius of small end of truncated cone

nonnegative scalar

Radius of small end of truncated cone, specified as a nonnegative scalar. Units are in meters.

Example: 5.5

Data Types: double

### **r2** — Radius of large end of truncated cone

positive scalar

Radius of large end of truncated cone, specified as a positive scalar. Units are in meters.

Example: 5.5

Data Types: double

### **height** — Height of truncated cone

positive scalar

Height of truncated cone, specified as a positive scalar. Units are in meters.

Example: 3.0

Data Types: double

### **c** — Signal propagation speed

positive scalar

Signal propagation speed, specified as a positive scalar. Units are in meters per second. For the SI value of the speed of light, use `physconst('LightSpeed')`.

Example: 3e8

Data Types: double

### **fc** — Frequency for computing radar cross section

positive scalar | positive, real-valued, 1-by-*L* row vector

Frequency for computing radar cross section, specified as a positive scalar or positive, real-valued, 1-by-*L* row vector. Frequency units are in Hz.

Example: [100e6 200e6]

Data Types: double

### **az** — Azimuth angles

-180:180 (default) | 1-by-*M* real-valued row vector

Azimuth angles for computing directivity and pattern, specified as a real-valued 1-by-*M* row vector where *M* is the number of azimuth angles. Angle units are in degrees. Azimuth angles must lie between -180° and 180°, inclusive.

The azimuth angle is the angle between the *x*-axis and the projection of a direction vector onto the *xy*-plane. The azimuth angle is positive when measured from the *x*-axis toward the *y*-axis.

Example: -45:2:45

Data Types: double

### **e1** — Elevation angles

-90:90 (default) | 1-by-*N* real-valued row vector

Elevation angles for computing directivity and pattern, specified as a real-valued, 1-by-*N* row vector where *N* is the number of desired elevation directions. Angle units are in degrees. Elevation angles must lie between -90° and 90°, inclusive.

The elevation angle is the angle between a direction vector and *xy*-plane. The elevation angle is positive when measured towards the *z*-axis.

Example: -75:1:70

Data Types: double

## **Output Arguments**

### **rcspat** — Radar cross section pattern

real-valued *N*-by-*M*-by-*L* array

Radar cross section pattern, returned as a real-valued  $N$ -by- $M$ -by- $L$  array.  $N$  is the length of the vector returned in the `eIout` argument.  $M$  is the length of the vector returned in the `azout` argument.  $L$  is the length of the `fc` vector. Units are in meters-squared.

Data Types: `double`

### **azout — Azimuth angles**

real-valued 1-by- $M$  row vector

Azimuth angles for computing directivity and pattern, returned as a real-valued 1-by- $M$  row vector where  $M$  is the number of azimuth angles specified by the `az` input argument. Angle units are in degrees.

The azimuth angle is the angle between the  $x$ -axis and the projection of the direction vector onto the  $xy$ -plane. The azimuth angle is positive when measured from the  $x$ -axis toward the  $y$ -axis.

Data Types: `double`

### **eIout — Elevation angles**

real-valued 1-by- $N$  row vector

Elevation angles for computing directivity and pattern, returned as a real-valued 1-by- $N$  row vector where  $N$  is the number of elevation angles specified in `eI` output argument. Angle units are in degrees.

The elevation angle is the angle between the direction vector and  $xy$ -plane. The elevation angle is positive when measured towards the  $z$ -axis.

Data Types: `double`

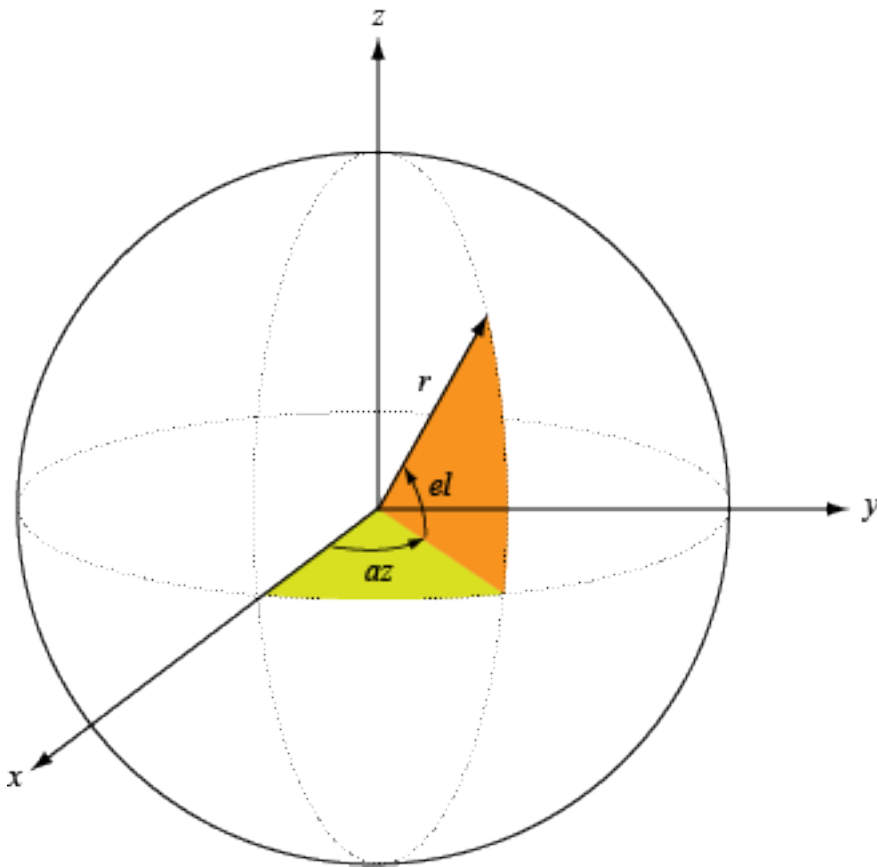
## **More About**

### **Azimuth and Elevation**

This section describes the convention used to define azimuth and elevation angles.

The azimuth angle of a vector is the angle between the  $x$ -axis and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going from the  $x$ -axis toward the  $y$ -axis. Azimuth angles lie between  $-180^\circ$  and  $180^\circ$  degrees, inclusive. The elevation angle is the angle between the vector and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going toward the positive  $z$ -axis from the  $xy$ -plane. Elevation angles lie between  $-90^\circ$  and  $90^\circ$  degrees, inclusive.





## References

- [1] Mahafza, Bassem. *Radar Systems Analysis and Design Using MATLAB, 2nd Ed.* Boca Raton, FL: Chapman & Hall/CRC, 2005.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`rcscylinder` | `rcsdisc` | `rcsphere` | `phased.BackscatterRadarTarget` | `phased.RadarTarget`

**Introduced in R2021a**

## refractionexp

CRPL exponential reference atmosphere refraction exponent

### Syntax

```
rexp = refractionexp(Ns)
```

### Description

`rexp = refractionexp(Ns)` computes the refraction exponent or decay constant of the “CRPL Exponential Reference Atmosphere Model” on page 1-128.

### Examples

#### Refraction Exponent as Function of Surface Refractivity

Compute the refraction exponents for surface refractivities equal to 200 N-units, 313 N-units, and 450 N-units.

```
srfrf = [200 313 450];  
rexp = refractionexp(srfrf)  
rexp = 1×3  
    0.1184    0.1439    0.2233
```

#### Radar Vertical Coverage Pattern

Compute and plot the radar vertical coverage pattern for a sinc antenna pattern. Specify a frequency of 100 MHz, an antenna height of 10 meters, and a range of 100 km. Assume the surface is smooth, the antenna is not tilted, and the transmitted polarization is horizontal.

```
frq = 100e6;  
anht = 10;  
rng = 100;
```

To specify the effective Earth radius, assume a high-latitude atmosphere model and a winter-like seasonal profile. Use the `refractiveidx` function to compute the refractivity gradient in N-units per meter using the Earth's surface and an altitude of 1 km.

```
alt1km = 1e3;  
[nidx,N] = refractiveidx([0 alt1km], ...  
    LatitudeModel="High",Season="Winter");  
RGrad = (nidx(2) - nidx(1))/alt1km;  
Re = effearthradius(RGrad);
```

Compute the vertical coverage pattern using the effective Earth radius and the radar parameters.

```
[vcpKm,vcpangles] = radarvcd(frq, rng, anht, ...
    EffectiveEarthRadius=Re);
```

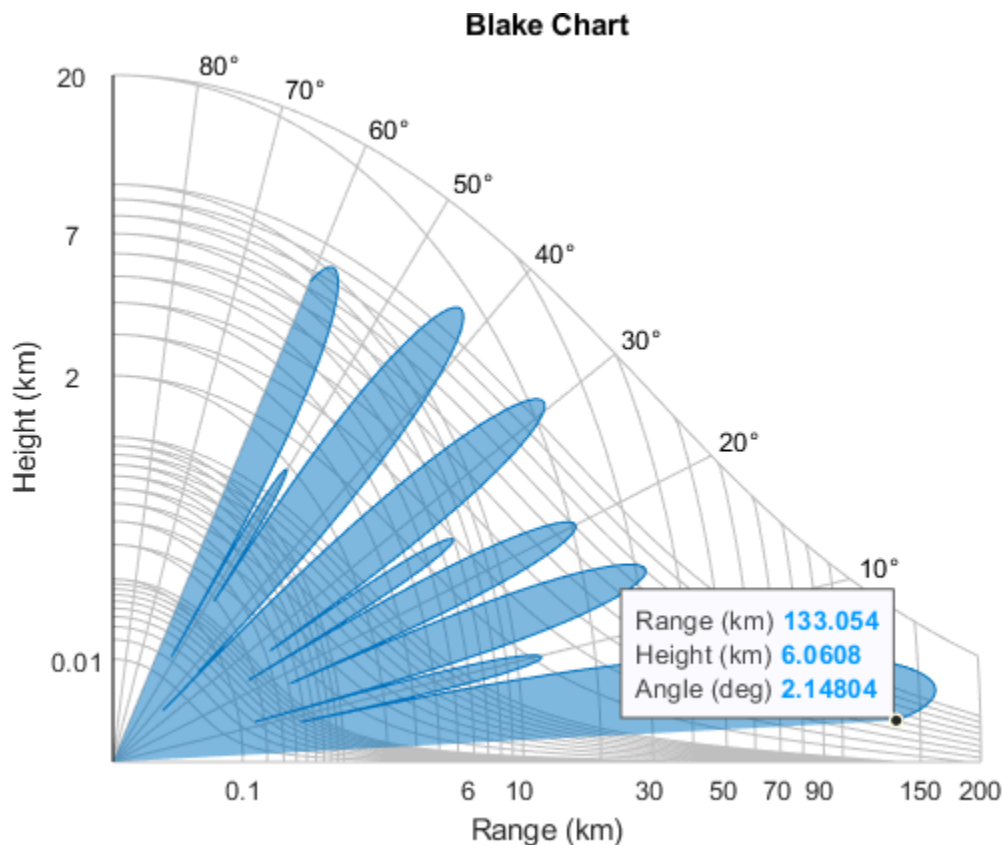
Use the refractivity at the surface in N-units to compute the refraction exponent.

```
Ns = N(1);
rexp = refractionexp(Ns)
```

```
rexp = 0.1438
```

Plot the vertical coverage pattern in the form of a Blake chart.

```
blakechart(vcpKm,vcpangles, ...
    SurfaceRefractivity=Ns,RefractionExponent=rexp)
```



## Input Arguments

### **Ns** — M-length refractivity at the surface

real scalar

M-length refractivity at the surface in N-units, specified as a real scalar.

Example: 313

Data Types: double

## Output Arguments

### rexp — Refraction exponent

nonnegative real scalar

Refraction exponent or decay constant in  $\text{km}^{-1}$ , returned as nonnegative real scalar.

## More About

### CRPL Exponential Reference Atmosphere Model

Atmospheric refraction evidences itself as a deviation in an electromagnetic ray from a straight line due to variation in air density as a function of height. The Central Radio Propagation Laboratory (CRPL) exponential reference atmosphere model treats refraction effects by assuming that the index of refraction  $n(h)$  and the refractivity  $N$  decay exponentially with height. The model defines

$$N = (n(h) - 1) \times 10^6 = N_s e^{-R_{\text{exp}} h},$$

where  $N_s$  is the atmospheric refractivity value (in units of  $10^{-6}$ ) at the surface of the earth,  $R_{\text{exp}}$  is the decay constant, and  $h$  is the height above the surface in kilometers. Thus

$$n(h) = 1 + (N_s \times 10^{-6}) e^{-R_{\text{exp}} h}.$$

The default value of  $N_s$  is 313 N-units and can be modified using the `SurfaceRefractivity` name-value argument in functions that accept it. The default value of  $R_{\text{exp}}$  is  $0.143859 \text{ km}^{-1}$  and can be modified using the `RefractionExponent` name-value argument in functions that accept it.

## References

- [1] Bean, B.R., and G.D. Thayer. "Central Radio Propagation Laboratory Exponential Reference Atmosphere." *Journal of Research of the National Bureau of Standards, Section D: Radio Propagation* 63D, no. 3 (November 1959): 315. <https://doi.org/10.6028/jres.063D.031>.
- [2] Dutton, E. J., and G. D. Thayer. *Techniques for Computing Refraction of Radio Waves in the Troposphere*. National Bureau of Standards Technical Note 97. United States National Bureau of Standards, 1961, revised 1964.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Apps

**Radar Designer**

### Functions

blakechart | el2height | height2el | height2range | height2grndrange | radarvcd | range2height

**Topics**

“Modeling Target Position Errors due to Refraction”

**Introduced in R2021b**

## probgrid

Nonuniformly spaced probabilities

### Syntax

```
p = probgrid(p1,p2)
p = probgrid(p1,p2,n)
```

### Description

`p = probgrid(p1,p2)` returns a nonuniformly spaced array of 100 probabilities between `p1` and `p2` that correspond to the values of the normal cumulative distribution function (CDF) evaluated over a set of points uniformly spaced in the domain of the normal distribution.

`p = probgrid(p1,p2,n)` returns an array of `n` probabilities.

### Examples

#### Normal CDF Samples

Evaluate the standard normal cumulative distribution function (CDF) on a 10-point grid between 0.2 and 0.95. Determine the points that correspond to the probabilities by evaluating the inverse normal CDF, also known as the *probit* function.

```
pmin = 0.2;
pmax = 0.95;
N = 10;
```

```
pd = probgrid(pmin,pmax,N);
```

```
xd = sqrt(2)*erfinv(2*pd-1);
```

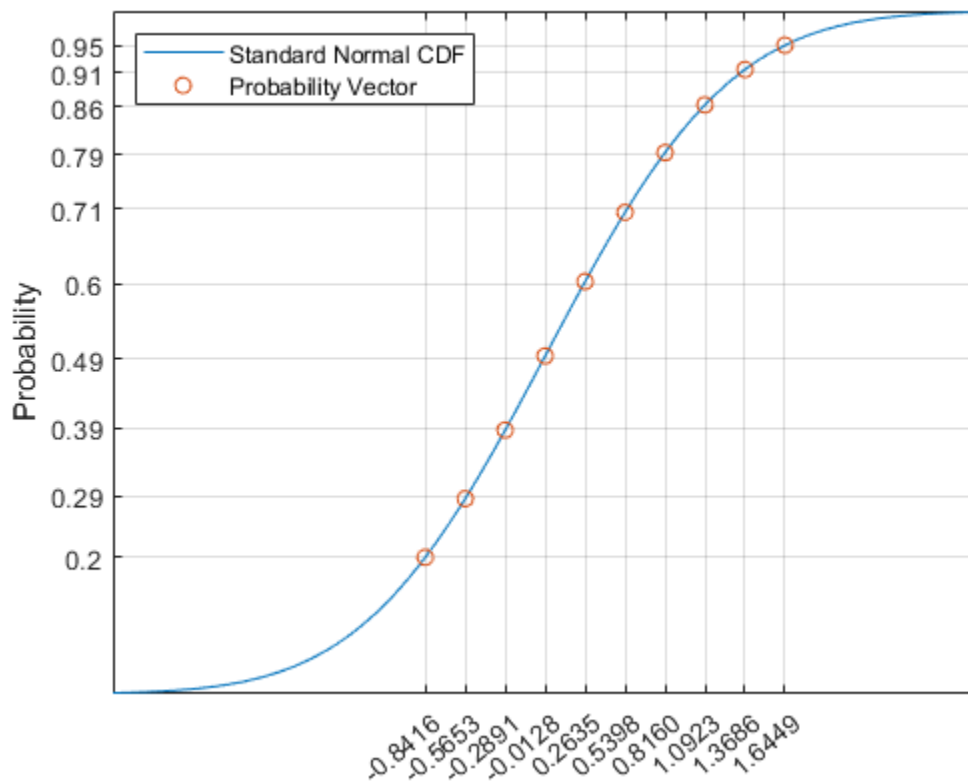
Plot the standard normal CDF and overlay the points generated by `probgrid`.

```
x = -3:0.01:3;
snCDF = (1+erf(x/sqrt(2)))/2;
```

```
plot(x,snCDF)
```

```
hold on
plot(xd,pd,'o')
hold off
```

```
legend({'Standard Normal CDF','Probability Vector'}, ...
       'Location','Northwest')
xticks(xd)
xtickangle(40)
yticks(round(100*pd)/100)
ylabel('Probability')
grid on
```



## Input Arguments

### **p1, p2 — Interval endpoints**

scalars from the interval  $[0, 1]$

Interval endpoints, specified as scalars from the interval  $[0, 1]$ .  $p1$  and  $p2$  must obey  $p1 < p2$ .

Data Types: double

### **n — Number of samples in probability grid**

100 (default) | positive integer scalar

Number of samples in probability grid, specified as a positive integer scalar.

Data Types: double

## Output Arguments

### **p — Array of probabilities**

row vector

Array of probabilities, returned as a row vector.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`detectability` | `rocinterp`

**Introduced in R2021a**



# rocinterp

ROC curve interpolation

## Syntax

```
ipd = rocinterp(snr,pd,snrq,'snr-pd')
isnr = rocinterp(pd,snr,pdq,'pd-snr')

ipd = rocinterp(pfa,pd,pfaq,'pfa-pd')
ipfa = rocinterp(pd,pfa,pdq,'pd-pfa')
```

## Description

`ipd = rocinterp(snr,pd,snrq,'snr-pd')` returns the probability of detection ( $P_d$ ) computed by interpolating a  $P_d$  vs. signal-to-noise ratio (SNR) receiver operating characteristic (ROC) curve. If `pd` is a matrix, the function interpolates each column independently. In this and the next syntax, `rocinterp` performs linear interpolation after transforming the  $P_d$ -axis of the ROC curve using the normal probability scale.

`isnr = rocinterp(pd,snr,pdq,'pd-snr')` returns the SNR computed by interpolating a  $P_d$  vs. SNR ROC curve. If `snr` is a matrix, the function interpolates each column independently.

`ipd = rocinterp(pfa,pd,pfaq,'pfa-pd')` returns the  $P_d$  computed by interpolating a  $P_d$  vs. probability of false alarm ( $P_{fa}$ ) ROC curve. If `pd` is a matrix, the function interpolates each column independently. In this and the next syntax, `rocinterp` performs linear interpolation after transforming both axes of the ROC curve using a logarithmic scale.

`ipfa = rocinterp(pd,pfa,pdq,'pd-pfa')` returns the  $P_{fa}$  computed by interpolating a  $P_d$  vs.  $P_{fa}$  ROC curve. If `pfa` is a matrix, the function interpolates each column independently.

## Examples

### Interpolate Probability of Detection vs. SNR ROC Curve

Compute the probability of detection ( $P_d$ ) for a Swerling 1 case target given a set of signal-to-noise ratio (SNR) and probability of false alarm values. Express the SNR values in decibels.

```
SNR = [13.5 14.5];
pfa = [1e-9 1e-6 1e-3];
```

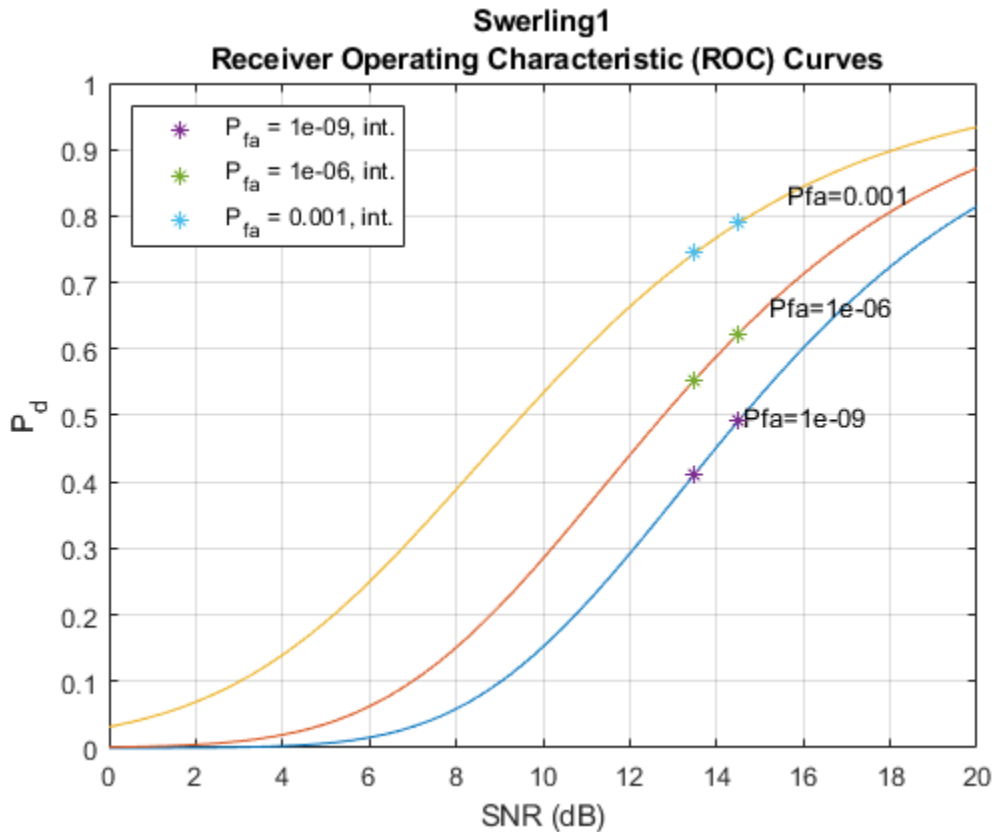
Compute the  $P_d$  vs. SNR ROC curves and interpolate them at the SNR values of interest.

```
[pd,snr] = rocpfa(pfa,'SignalType','Swerling1');
ipd = rocinterp(snr,pd,SNR,'snr-pd');
```

Plot the ROC curves and overlay the interpolated values.

```
rocpfa(pfa,'SignalType','Swerling1')
hold on
```

```
q = plot(SNR,ipd,'*');
hold off
legend(q,append("P_{fa} = ",string(pfa)," , int."),'Location','northwest')
```



## Input Arguments

### snr — Signal-to-noise ratio

vector | matrix

Signal-to-noise ratio in decibels (dB), specified as a vector or matrix. If `snr` is a vector, its values must be unique. If `snr` is a matrix, then each of its columns must contain unique values.

Data Types: `double`

### snrq — Signal-to-noise ratio query points

vector

Signal-to-noise ratio query points, specified as a vector. All values of `snrq` must be expressed in dB.

Data Types: `double`

### pd — Probability of detection

vector | matrix

Probability of detection, specified as a vector or matrix. All values of `pd` must be between 0 and 1. If `pd` is a vector, its values must be unique. If `pd` is a matrix, then each of its columns must contain unique values.

Data Types: `double`

### **pdq — Probability of detection query points**

vector

Probability of detection query points, specified as a vector. All values of `pdq` must be between 0 and 1.

Data Types: `double`

### **pfa — Probability of false alarm**

vector | matrix

Probability of false alarm, specified as a vector or matrix. All values of `pfa` must be between 0 and 1. If `pfa` is a vector, its values must be unique. If `pfa` is a matrix, then each of its columns must contain unique values.

Data Types: `double`

### **pfaq — Probability of false alarm query points**

vector

Probability of false alarm query points, specified as a vector. All values of `pfaq` must be between 0 and 1.

Data Types: `double`

## **Output Arguments**

### **ipd — Interpolated probability of detection**

vector | matrix

Interpolated probability of detection, returned as a vector or matrix.

### **isnr — Interpolated signal-to-noise ratio**

vector | matrix

Interpolated signal-to-noise ratio, returned as a vector or matrix.

### **ipfa — Interpolated probability of false alarm**

vector | matrix

Interpolated probability of false alarm, returned as a vector or matrix.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`detectability` | `rocdfa` | `rocsnr`

**Introduced in R2021a**

# gaspl

RF signal attenuation due to atmospheric gases

## Syntax

`L = gaspl(range, freq, T, P, den)`

## Description

`L = gaspl(range, freq, T, P, den)` returns the attenuation, `L`, of signals propagating through the atmosphere.

- `range` represents the signal path length.
- `freq` represents the signal carrier frequency.
- `T` represents the ambient temperature.
- `P` represents the atmospheric pressure.
- `den` represents the atmospheric water vapor density.

The `gaspl` function applies the International Telecommunication Union (ITU) atmospheric gas attenuation model [1] to calculate path loss for signals primarily due to oxygen and water vapor. The model computes attenuation as a function of ambient temperature, pressure, water vapor density, and signal frequency.

The function requires that the signal path is contained entirely in a homogeneous environment - temperature `T`, atmospheric pressure `P`, and water vapor density `den` do not vary along the signal path. You can account for the variation of atmospheric parameters with height using the `tropopl` and `atmositu` functions in the Radar Toolbox.

The attenuation model applies only for frequencies at 1-1000 GHz.

## Examples

### Atmospheric Gas Attenuation Spectrum

Compute the attenuation spectrum from 1 to 1000 GHz for an atmospheric pressure of 101.300 kPa and a temperature of 15°C. Plot the spectrum for a water vapor density of  $7.5 \text{ g/m}^3$  and then plot the spectrum for dry air (zero water vapor density).

Set the attenuation frequencies.

```
freq = [1:1000]*1e9;
```

Assume a 1 km path distance.

```
R = 1000.0;
```

Compute the attenuation for air containing water vapor.

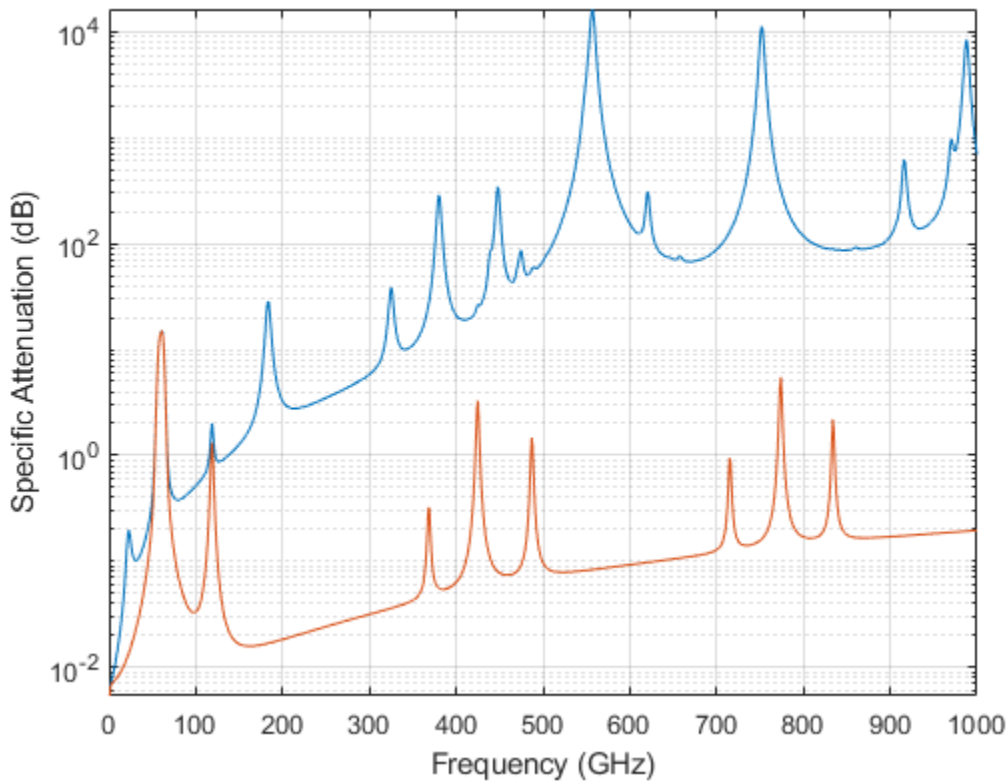
```
T = 15;
P = 101300.0;
W = 7.5;
L = gaspl(R, freq, T, P, W);
```

Compute the attenuation for dry air.

```
L0 = gaspl(R, freq, T, P, 0.0);
```

Plot the attenuations.

```
semilogy(freq/1e9, L)
hold on
semilogy(freq/1e9, L0)
grid
xlabel('Frequency (GHz)')
ylabel('Specific Attenuation (dB)')
hold off
```



### Plot Attenuation Due to Atmospheric Gases and Free Space

First, plot the specific attenuation of atmospheric gases for frequencies from 1 GHz to 1000 GHz. Assume a sea-level dry air pressure of  $101.325 \times 10^3$  kPa and a water vapor density of  $7.5 \text{ g/m}^3$ . The air temperature is  $20^\circ\text{C}$ . Specific attenuation is defined as dB loss per kilometer. Then, plot the actual attenuation at 10 GHz for a span of ranges.

### Plot Specific Atmospheric Gas Attenuation

Set the atmosphere temperature, pressure, water vapor density.

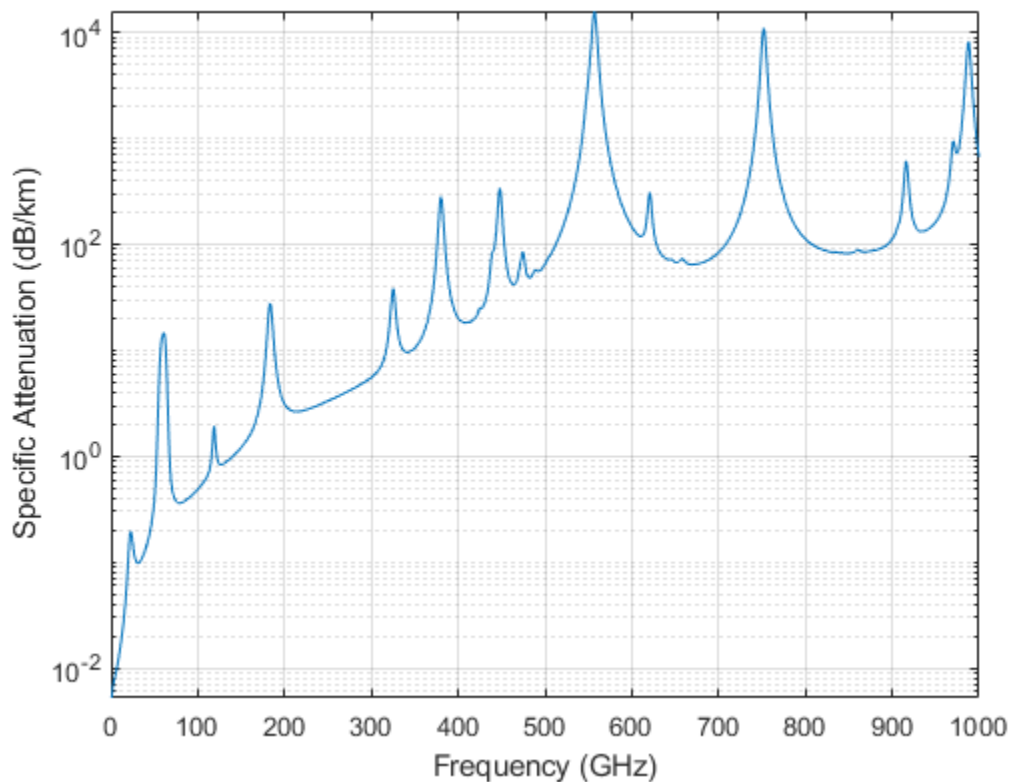
```
T = 20.0;
Patm = 101.325e3;
rho_wv = 7.5;
```

Set the propagation distance, speed of light, and frequencies.

```
km = 1000.0;
c = physconst('LightSpeed');
freqs = [1:1000]*1e9;
```

Compute and plot the atmospheric gas loss.

```
loss = gaspl(km, freqs, T, Patm, rho_wv);
semilogy(freqs/1e9, loss)
grid on
xlabel('Frequency (GHz)')
ylabel('Specific Attenuation (dB/km)')
```



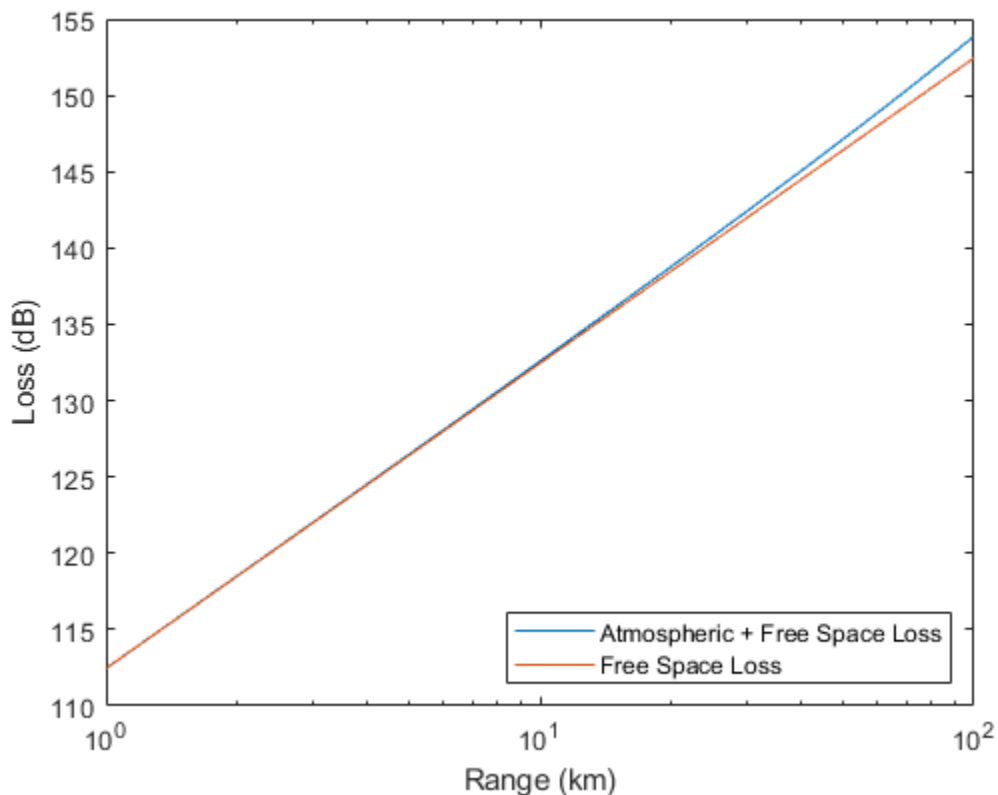
### Plot Actual Atmospheric and Free Space Attenuation

Compute both free space loss and atmospheric gas loss at 10 GHz for ranges from 1 to 100 km. The frequency corresponds to an X-band radar. Then, plot the free space loss and the total (atmospheric + free space) loss.

```

ranges = [1:100]*1000;
freq_xband = 10e9;
loss_gas = gaspl(ranges, freq_xband, T, Patm, rho_wv);
lambda = c/freq_xband;
loss_fsp = fspl(ranges, lambda);
semilogx(ranges/1000, loss_gas + loss_fsp.', ranges/1000, loss_fsp)
legend('Atmospheric + Free Space Loss', 'Free Space Loss', 'Location', 'SouthEast')
xlabel('Range (km)')
ylabel('Loss (dB)')

```



## Input Arguments

### range — Signal path length

nonnegative real-valued scalar |  $M$ -by-1 nonnegative real-valued column vector | 1-by- $M$  nonnegative real-valued row vector

Signal path length used to compute attenuation, specified as a nonnegative real-valued scalar or vector. You can specify multiple path lengths simultaneously. Units are in meters.

Example: [13000.0, 14000.0]

### freq — Signal frequency

positive real-valued scalar |  $N$ -by-1 nonnegative real-valued column vector | 1-by- $N$  nonnegative real-valued row vector



Signal frequency, specified as a positive real-valued scalar, or as an  $N$ -by-1 nonnegative real-valued vector or 1-by- $N$  nonnegative real-valued vector. You can specify multiple frequencies simultaneously. Frequencies must lie in the range 1-1000 GHz. Units are in hertz.

Example: [1.4e9, 2.0e9]

### T – Ambient temperature

real-valued scalar

Ambient temperature, specified as a real-valued scalar. Units are in degrees Celsius.

---

**Note** The `atmositu`, `gaspl`, and `tropopl` functions use different units for pressure and temperature.

### Pressure and Temperature Units

Function	Pressure Units	Temperature Units
<code>atmositu</code>	hectoPascals (hPa)	kelvin (K)
<code>tropopl</code>	hectoPascals (hPa)	kelvin (K)
<code>gaspl</code>	Pascals (Pa)	Celsius (C)

One hPa equals 100 Pa and  $K = C + 273.15$ . Use caution when combining the use of these three functions.

---

Example: -10.0

### P – Dry air pressure

positive real-valued scalar

Dry air pressure, specified as a positive real-valued scalar. Units are in Pa. One standard atmosphere at sea level is 101325 Pa.

Example: 101300.0

### den – Water vapor density

nonnegative real-valued scalar

Water vapor density or absolute humidity, specified as a nonnegative real-valued scalar. Units are  $\text{g/m}^3$ . The maximum water vapor density of air at 30° C is approximately 30.0  $\text{g/m}^3$ . The maximum water vapor density of air at 0°C is approximately 5.0  $\text{g/m}^3$ .

Example: 4.0

## Output Arguments

### L – Signal attenuation

real-valued  $M$ -by- $N$  matrix

Signal attenuation, returned as a real-valued  $M$ -by- $N$  matrix. Each matrix row represents a different path where  $M$  is the number of paths. Each column represents a different frequency where  $N$  is the number of frequencies. Units are in dB.

## More About

### Atmospheric Gas Attenuation Model

This model calculates the attenuation of signals that propagate through atmospheric gases.

Electromagnetic signals attenuate when they propagate through the atmosphere. This effect is due primarily to the absorption resonance lines of oxygen and water vapor, with smaller contributions coming from nitrogen gas. The model also includes a continuous absorption spectrum below 10 GHz. The ITU model *Recommendation ITU-R P.676-10: Attenuation by atmospheric gases* is used. The model computes the specific attenuation (attenuation per kilometer) as a function of temperature, pressure, water vapor density, and signal frequency. The atmospheric gas model is valid for frequencies from 1-1000 GHz and applies to polarized and nonpolarized fields.

The formula for specific attenuation at each frequency is

$$\gamma = \gamma_o(f) + \gamma_w(f) = 0.1820fN''(f).$$

The quantity  $N''()$  is the imaginary part of the complex atmospheric refractivity and consists of a spectral line component and a continuous component:

$$N''(f) = \sum_i S_i F_i + N''_D(f)$$

The spectral component consists of a sum of discrete spectrum terms composed of a localized frequency bandwidth function,  $F(f)_i$ , multiplied by a spectral line strength,  $S_i$ . For atmospheric oxygen, each spectral line strength is

$$S_i = a_1 \times 10^{-7} \left( \frac{300}{T} \right)^3 \exp \left[ a_2 \left( 1 - \left( \frac{300}{T} \right) \right) \right] P.$$

For atmospheric water vapor, each spectral line strength is

$$S_i = b_1 \times 10^{-1} \left( \frac{300}{T} \right)^{3.5} \exp \left[ b_2 \left( 1 - \left( \frac{300}{T} \right) \right) \right] W.$$

$P$  is the dry air pressure,  $W$  is the water vapor partial pressure, and  $T$  is the ambient temperature. Pressure units are in hectoPascals (hPa) and temperature is in degrees Kelvin. The water vapor partial pressure,  $W$ , is related to the water vapor density,  $\rho$ , by

$$W = \frac{\rho T}{216.7}.$$

The total atmospheric pressure is  $P + W$ .

For each oxygen line,  $S_i$  depends on two parameters,  $a_1$  and  $a_2$ . Similarly, each water vapor line depends on two parameters,  $b_1$  and  $b_2$ . The ITU documentation cited at the end of this section contains tabulations of these parameters as functions of frequency.

The localized frequency bandwidth functions  $F_i(f)$  are complicated functions of frequency described in the ITU references cited below. The functions depend on empirical model parameters that are also tabulated in the reference.

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the path length,  $R$ . Then, the total attenuation is  $L_g = R(\gamma_o + \gamma_w)$ .

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands, and apply attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

## References

- [1] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.676-10: Attenuation by atmospheric gases* 2013.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

## See Also

`fspl` | `fogpl` | `tropopl` | `phased.LOSChannel` | `phased.WidebandLOSChannel`

**Introduced in R2020b**

## el2height

Convert target elevation angle to height

### Syntax

```
tgtht = el2height(el,anht,R)
tgtht = el2height( ____,model)
tgtht = el2height( ____,re)
```

### Description

`tgtht = el2height(el,anht,R)` returns the target height in meters. This function assumes that heights are referenced to the ground.

`tgtht = el2height( ____,model)` specifies the Earth model used to compute the target height. Specify `model` as 'Curved' or 'Flat'.

`tgtht = el2height( ____,re)` specifies the effective Earth radius in meters as a positive scalar `re`.

### Examples

#### Determine Target Height

Determine the target height in meters given an elevation angle of 0.5 degrees, a sensor height of 10 m, and a range of 300 km. Convert the range to meters.

```
el = 0.5;
anht = 10;
R = 300e3;

tgtht = el2height(el,anht,R)

tgtht = 7.9325e+03
```

### Input Arguments

**el — Elevation angle**  
scalar | *M*-length vector

Elevation angle to target, specified as a scalar or *M*-length vector. Units are in degrees.

Data Types: `double`

**anht — Sensor height**  
scalar | *M*-length vector

Sensor height, specified as a scalar or *M*-length vector. Units are in meters.

Data Types: double

**R — Range**

scalar |  $M$ -length vector

Range between target and sensor, specified as a scalar or  $M$ -length vector. Units are in meters.

Data Types: double

**model — Earth model**

'Curved' (default) | 'Flat'

Earth model used to compute target height, specified as 'Curved' or 'Flat'. By default, the `el2height` function assumes a curved Earth model.

**re — Effective Earth radius**

positive scalar

Effective Earth radius, specified as a positive scalar. By default, `re` is 4/3 of the Earth radius. Units are in meters. The function ignores this input when `model` is set to 'Flat'.

Data Types: double

**Output Arguments****tgtht — Target height**

scalar |  $M$ -length vector

Target height, returned as a scalar or  $M$ -length vector. Units are in meters.

**References**

[1] Barton, David K. *Radar Equations for Modern Radar*. Artech House Radar Series. Norwood, Mass: Artech House, 2013.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also****Functions**

`height2el` | `horizonrange` | `depressionang` | `grazingang` | `effearthradius`

**Introduced in R2021a**

## height2el

Convert target height to elevation angle

### Syntax

```
el = height2el(tgtht,anht,R)
el = height2el( ____,model)
el = height2el( ____,re)
```

### Description

`el = height2el(tgtht,anht,R)` returns the target elevation angle in degrees. This function assumes that heights are referenced to the ground.

`el = height2el( ____,model)` specifies the Earth model used to compute the target elevation. Specify `model` as 'Curved' or 'Flat'.

`el = height2el( ____,re)` specifies the effective Earth radius in meters as a positive scalar `re`.

### Examples

#### Determine Elevation Angle of Target

Determine the elevation angle of a target given a target height of 8 km, sensor height of 10 m, and range of 300 km. Convert the target height and range to meters.

```
tgtht = 8e3;
anht = 10;
R = 300e3;

el = height2el(tgtht,anht,R)

el = 0.5129
```

### Input Arguments

#### **tgtht** — Target height

scalar | *M*-length vector

Target height, specified as a scalar or *M*-length vector. Units are in meters.

Data Types: double

#### **anht** — Sensor height

scalar | *M*-length vector

Sensor height, specified as a scalar or *M*-length vector. Units are in meters.

Data Types: double

**R — Range**scalar |  $M$ -length vector

Range between target and sensor, specified as a scalar or  $M$ -length vector. Units are in meters.

Data Types: double

**model — Earth model**

'Curved' (default) | 'Flat'

Earth model used to compute target elevation angle, specified as 'Curved' or 'Flat'. By default, the height2el function assumes a curved Earth model.

**re — Effective Earth radius**

positive scalar

Effective Earth radius, specified as a positive scalar. By default, re is 4/3 of the Earth radius. Units are in meters. The function ignores this input when model is set to 'Flat'.

Data Types: double

**Output Arguments****e1 — Target elevation angle**scalar |  $M$ -length vector

Target elevation angle, returned as a scalar or  $M$ -length vector. Units are in degrees.

**References**

[1] Barton, David K. *Radar Equations for Modern Radar*. Artech House Radar Series. Norwood, Mass: Artech House, 2013.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also****Functions**

depressionang | effearthradius | el2height | grazingang | horizonrange

**Introduced in R2021a**

## clutterSurfaceRCS

Surface clutter radar cross section

### Syntax

```
rsc = clutterSurfaceRCS(nrsc,range,azimuth,elevation,graz,tau)
rsc = clutterSurfaceRCS( ____,C)
rsc = clutterSurfaceRCS( ____, 'BeamLoss',Lp)
```

### Description

`rsc = clutterSurfaceRCS(nrsc,range,azimuth,elevation,graz,tau)` returns the radar cross section, `rsc`, of the surface clutter patch as an  $M$ -length row vector in meters squared.

`rsc = clutterSurfaceRCS( ____,C)` returns the surface clutter radar cross-section with the propagation speed  $C$ .

`rsc = clutterSurfaceRCS( ____, 'BeamLoss',Lp)` returns the surface clutter radar cross section using the beamshape loss.

### Examples

#### Calculate Radar Cross Section

Calculate the radar cross section of a clutter patch and estimate the clutter-to-noise ratio at the receiver. Assume that the patch is 1000 meters away from the radar system and the azimuth and elevation beamwidths are 1 degree and 3 degrees, respectively. Also assume that the grazing angle is 20 degrees, the pulse width is 10 microseconds, and the radar is operated at a wavelength of 1 cm with a peak power of 5 kw.

```
rng    = 1000;
bwAz   = 1;
bwEl   = 3;
graz   = 20;
tau    = 10e-6;
lambda = 0.01;
ppow   = 5000;
```

Calculate the NRCS.

```
nrsc = landreflectivity('Mountains',graz)
```

```
nrsc = 0.1082
```

Calculate clutter RCS using the calculated NRCS.

```
rsc = clutterSurfaceRCS(nrsc,rng,bwAz,bwEl,graz,tau)
```

```
rsc = 288.9855
```

Calculate clutter-to-noise ratio using the calculated RCS.



```
cnr = radareqsnr(lambda, rng, ppow, tau, 'rsc', rcs)
cnr = 62.5973
```

## Input Arguments

### **nrcs** — Normalized radar cross section

nonnegative scalar |  $M$ -length vector of nonnegative values

The normalized radar cross section (NRCS) of a clutter patch is specified as either a nonnegative scalar or an  $M$ -length vector of nonnegative values in meters squared. The NRCS is also known as the reflectivity or  $\sigma^0$ .

Example: `nrcs = 1`

### **range** — Clutter patch range

nonnegative scalar |  $M$ -length vector of nonnegative values

The clutter patch range, specified as either a nonnegative scalar or an  $M$ -length vector of nonnegative values in meters.

Example: `range = 1000;`

### **azimuth** — Azimuth beamwidth

positive scalar | [`azimuth_Tx`, `azimuth_Rx`]

The azimuth beamwidth of the radar, specified as a positive scalar or a 1-by-2 vector in degrees. Use with the `elevation` argument.

- When the transmit and receive beamwidths are the same, specify `azimuth` as a positive scalar .
- When the transmit and receive azimuth beamwidths are not the same, specify `azimuth` as a 1-by-2 positive vector [`azimuth_Tx`, `azimuth_Rx`], where the first element is the transmit azimuth beamwidth in degrees and the second element is the receive azimuth beamwidth in degrees.

The function uses these two beamwidths to create an effective azimuth beamwidth. See “Effective Beamwidth” on page 1-150.

Example: `bwAz = 1`

### **elevation** — Elevation beamwidth

positive scalar | [`elevation_Tx`, `elevation_Rx`]

The elevation beamwidth of the radar, specified as a positive scalar or a 1-by-2 vector in degrees. Use with the `azimuth` argument.

- When the transmit and receive beamwidths are the same, specify `elevation` as a positive scalar .
- When the transmit and receive elevation beamwidths are not the same, specify `elevation` as a 1-by-2 positive vector [`elevation_Tx`, `elevation_Rx`], where the first element is the transmit azimuth beamwidth in degrees and the second element is the receive azimuth beamwidth in degrees.

The function uses these two beamwidths to create an effective elevation beamwidth. See “Effective Beamwidth” on page 1-150.

Example: `bwEl = 3`

**graz — Grazing angle**nonnegative scalar |  $N$ -length vector of grazing values

Grazing angle, specified as a nonnegative scalar or an  $N$ -length row vector of nonnegative values. This argument specifies the grazing angles of the clutter patch relative to the radar. Units are in degrees. See `grazingang`.

**tau — Pulse width**

nonnegative scalar

Pulse width of the transmitted signal, specified as a nonnegative scalar in seconds.

Example: `tau = 10e-6`**C — Propagation speed**

speed of light (default) | positive scalar

The propagation speed specified as a positive scalar in meters per second.

**Lp — Beamshape loss**

0 dB (default) | nonnegative scalar

The beamshape loss, specified as a nonnegative scalar in decibels. The beamshape loss accounts for the reduced two-way antenna gain of off-axis scatterers.

Use this property when the elevation beamwidth (`elevation`) for the transmitter and receiver are not the same.

Example: `loss = 0`

## Output Arguments

**rscs — Radar cross section** $M$ -length vector

The radar cross section of a surface cluster patch, returned as an  $M$ -length vector in meters squared.

## Algorithms

**Effective Beamwidth**

The effective beamwidth is used for the effective azimuth  $\theta_{azimutheff}$  and effective elevation  $\theta_{elevationeff}$  calculation when the transmitter and receiver beamwidths are not equal.

$$\theta_{azimutheff} = \frac{\sqrt{2\theta_{at}\theta_{ar}}}{\sqrt{\theta_{at}^2 + \theta_{ar}^2}}$$
$$\theta_{elevationeff} = \frac{\sqrt{2\theta_{et}\theta_{er}}}{\sqrt{\theta_{et}^2 + \theta_{er}^2}}$$

- $at$  is the azimuth transmitter elevation beamwidth in degrees.
- $ar$  is the azimuth receiver elevation beamwidth in degrees.
- $et$  is the elevation transmitter elevation beamwidth in degrees.

- $\theta_r$  is the elevation receiver elevation beamwidth in degrees.

## References

- [1] Barton, David K. *Radar Equations for Modern Radar*. Norwood, MA: Artech House, 2013.
- [2] Long, Maurice W. *Radar Reflectivity of Land and Sea*. Boston: Artech House, 2001.
- [3] Nathanson, Fred E., J. Patrick Reilly, and Marvin N. Cohen. *Radar Design Principles*. Mendham, NJ: SciTech Publishing, 1999.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[landreflectivity](#) | [seareflectivity](#) | [radareqsnr](#) | [surfacegamma](#) | [grazingang](#)

**Introduced in R2021a**

# landreflectivity

Reflectivity of land surface

## Syntax

```
nrcs = landreflectivity(landtype,ang)
nrcs = landreflectivity(landtype,ang,freq)
nrcs = landreflectivity( ____,Model = model)
nrcs = landreflectivity( ____,Polarization = pol)
nrca = landreflectivity( ____,SurfaceHeightStandardDeviation = hgtsd)
[nrcs,hgtsd,beta0,vegtype] = landreflectivity( ____)
```

## Description

`nrcs = landreflectivity(landtype,ang)` returns the surface radar reflectivity `nrcs` for the land type `landtype` at the grazing angle `ang`. Radar reflectivity is also called the normalized radar cross section (NRCS). This syntax assumes that the radar operates at 10 GHz and also assumes that the land model is the Barton reflectivity model. For a table of land models and land types, see “Land Reflectivity Models and Land Types” on page 4-222.

`nrcs = landreflectivity(landtype,ang,freq)` also specifies the transmitted frequency of the radar.

`nrcs = landreflectivity( ____,Model = model)` also specifies the reflectivity model.

`nrcs = landreflectivity( ____,Polarization = pol)` also specifies the polarization `pol` of the transmitted wave. To use this syntax, set the `model` argument to 'UlabyDobson'.

`nrca = landreflectivity( ____,SurfaceHeightStandardDeviation = hgtsd)` specifies the scalar standard deviation of the surface height `hgtsd`. To use this syntax, set the `model` argument to 'GIT'.

`[nrcs,hgtsd,beta0,vegtype] = landreflectivity( ____)` returns

- `hgtsd` — the standard deviation of the surface height.
- `beta0` — the slope of the land type.
- `vegtype` — the vegetation type.

To enable this syntax, set the `model` argument to 'Barton'.

## Examples

### NRCS of Urban Patch

Calculate NRCS, surface height standard deviation, land slope, and vegetation type. Specify an urban land type and a grazing angle of 20 degrees.

```
graz = 20;
[nrcs,hgtsd,beta0,vegtype] = landreflectivity("Urban",graz)
```

```
nracs = 0.1082
hgtsd = 10
beta0 = 5.7296
vegtype =
'None'
```

## Input Arguments

### landtype — Surface land type

char | string

Surface land type, specified as a character array or string. The land type is depends on the value of the model. For the acceptable land types for different models, see the table “Land Models and Land Types” on page 1-155.

### ang — Grazing or depression angle

nonnegative scalar |  $M$ -length vector of nonnegative values

Grazing or depression angle of a surface relative to the radar, specified as a scalar or an  $M$ -length row vector of nonnegative values. When the land model is set to 'Billingsley', the angle is interpreted as a depression angle between -90 and 90 degrees. For all other models, the angle is interpreted as a grazing angle ranging from 0 to 90 degrees. Units are in degrees.

### freq — Transmitted frequencies

10e9 (default) | positive scalar |  $N$ -length vector of positive values

Transmitted frequencies, specified as a positive scalar or  $N$ -length vector of positive values. Units are in Hz.

Example: `freq = 7*10e9`

### model — Land reflectivity model

'Barton' (default) | string | char

Land reflectivity model, specified as a string or char. See the “Land Models and Land Types” on page 1-155 table for all acceptable land reflectivity models.

### pol — Polarization of reflectivity model

'H' (default) | 'V' | 'HV'

Polarization of reflectivity model, specified as 'H' for horizontal polarization, 'V' for vertical polarization, or 'HV' which indicates horizontal transmit with vertical receive.

### Dependencies

To enable this argument, set the model argument to 'UlabyDobson'.

Data Types: char | string

### hgtsd — Surface height standard deviation

0 (default) | scalar

Standard deviation of the surface height, specified as a scalar. Units are in meters.

**Dependencies**

To enable this argument, set the `model` argument to 'GIT'.

**Output Arguments**

**nrcs — Normalized surface reflectivity**

real-valued  $N$ -length row vector | real-valued  $M$ -by- $N$  matrix

Normalized surface reflectivity, returned as either a real-valued  $N$ -length row vector or a real-valued  $M$ -by- $N$  matrix. Normalized reflectivity is also called normalized radar cross section.  $M$  is the length of the grazing angle or depression angle vector `graz` and  $N$  is the length of the frequency vector `freq`. `nrcs` is dimensionless but often expressed as  $m^2/m^2$ .

**hgtstd — Standard deviation of surface height**

scalar

Standard deviation of the surface height, returned as a scalar. Units are in meters.

**Dependencies**

To enable this argument, set the `model` argument to 'Barton'.

**beta0 — Slope of the land type**

scalar

Slope of the land type  $\beta_0$ , returned as a scalar. Note that  $\beta_0$  is 1.4 times the RMS surface slope. Units are in degrees.

**Dependencies**

To enable this argument, set the `model` argument to 'Barton'.

**vegtype — Vegetation type**

character array | string

Vegetation type, returned a character array or string. The vegetation type depends on the land type.

Land Type	Vegetation Type
Rugged Mountains	Trees (dense)
Mountains	Trees (dense)
Woods	Trees (dense)
Wooded Hills	Trees (dense)
Rolling Hills	Brush (dense)
Farm	Grass (thin)
Desert	Grass (thin)
Flatland	Grass (thin)
Metropolitan	None
Urban	None
Smooth	None

## Dependencies

To enable this argument, set the `model` argument to `'Barton'`.

## Limitations

This function assumes a Gaussian clutter model and that the reflectivity of land clutter is mostly independent of wavelength. The Gaussian model may fail to simulate the effects of some natural and most man-made structures, which are generally modeled separately as discrete clutter.

## More About

### Land Models and Land Types

Model	Land Type	Range of Validity
'Barton' - Constant-gamma mathematical model generally applicable over medium grazing angles. 'Barton' is the default model. See [1][2], and [3].	'RuggedMountains'	<ul style="list-style-type: none"> <li>Grazing angle 20 - 60 degrees</li> <li>Frequency 1 - 10 GHz</li> </ul>
	'Mountains'	
	'Metropolitan'	
	'Urban'	
	'WoodedHills'	
	'RollingHills'	
	'Woods'	
	'Farm'	
	'Desert'	
	'Flatland' (default for model)	
	'Smooth'	
'APL' - This model also known as the ADSAM model. Low-fidelity constant-gamma mathematical model that includes specular scattering. See [4].	'Urban'	<ul style="list-style-type: none"> <li>Grazing angle 0 - 90 degrees</li> <li>Frequency 1 - 100 GHz</li> </ul>
	'HighRelief'	
	'LowRelief' (default for model)	

Model	Land Type	Range of Validity
'Billingsley' - High-validity empirical model generally applicable for low depression angles less than 2 degrees. See [5].	'LowReliefRural' (default for model)	<ul style="list-style-type: none"> <li>• Depression angle -0.75 - 2 degrees</li> <li>• Frequency - VHF (0.030 - 0.3), UHF (0.3 - 1), L (1 - 2), S (2 - 4), X (8 - 12) GHz</li> </ul>
	'LowReliefForest'	
	'Farm'	
	'Desert'	
	'Marsh'	
	'Grassland'	
	'HighReliefRural'	
	'HighReliefForest'	
	'Mountains'	
	'Urban'	
'GIT' - Georgia Institute of Technology semi-empirical model takes into account terrain roughness. Generally applicable for medium grazing angles. See [6].	'Soil' (default for Model)	<ul style="list-style-type: none"> <li>• Grazing angle 20 - 65 degrees</li> <li>• Frequency 3 - 15 GHz</li> </ul>
	'Grass'	
	'TallGrass'	
	'Trees'	
	'Urban'	
'Morchin' - Mathematical model generally applicable for high grazing angles for frequencies from UHF to C-band. See [7].	'Desert'	<ul style="list-style-type: none"> <li>• Grazing angle 70 - 90 degrees</li> <li>• Frequencies UHF (0.3 - 1) L (1 - 2) S (2 - 4) C (4 - 8)</li> </ul>
	'Farm' (default for Model)	
	'Woods'	
	'Mountains'	
'Nathanson' - Applicable up to Ka band for low grazing angle surface radars and medium grazing angle airborne radars for low mountains, farmland, and wooded areas. See [3].	'Desert'	<ul style="list-style-type: none"> <li>• Grazing angle 0 - 60 degrees</li> <li>• Frequency L (1 - 2). S (2 - 4), C (4 - 8), X (8 - 12), Ku (12 - 18), Ka (32 - 36) GHz</li> </ul>
	'Farm' (default for Model)	
	'Woods'	
	'Jungle'	
	'RollingHills'	
	'Urban'	
'UlabyDobson' - High-validity semi-empirical model for low to medium grazing angles covering L-band to Ku, taking into account polarization. See [8].	'Soil' (default for Model)	<ul style="list-style-type: none"> <li>• Grazing angle 0 - 60 degrees</li> <li>• Frequency L (1 - 2), S (2 - 4), C (4 - 8), X (8 - 12), Ku (12 - 18) GHz</li> </ul>
	'Grass'	
	'Shrubs'	
	'ShortVegetation'	

## References

[1] Barton, David Knox. *Radar Equations for Modern Radar*. Artech House, 2013.



- [2] Long, Maurice W. *Radar Reflectivity of Land and Sea*. 3rd ed, Artech House, 2001.
- [3] Nathanson, Fred E., et al. *Radar Design Principles: Signal Processing and the Environment*. 2. ed., Repr, Scitech Publ, 2004.
- [4] Reilly, J. P., R. L. McDonald, and G. D. Dockery. "RF-Environment Models for the ADSAM Program." Report No. A1A97U-070, Laurel, MD: Johns Hopkins University Applied Physics Laboratory, August 22, 1997.
- [5] Billingsley, J. Barrie. *Low-Angle Radar Land Clutter: Measurements and Empirical Models*. William Andrew Pub. : SciTech Pub. ; Institution of Electrical Engineers, 2002.
- [6] Richards, M. A., et al., editors. *Principles of Modern Radar*. SciTech Pub, 2010.
- [7] Morchin, Fred E., J. Patrick Reilly, and Marvin Cohen. *Radar Design Principles: Signal Processing and the Environment*. 2nd ed. New York: McGraw-Hill, 1991.
- [8] Ulaby, Fawwaz T., and M. Craig Dobson. *Handbook of Radar Scattering Statistics for Terrain*

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[landroughness](#) | [searoughness](#) | [seareflectivity](#) | [clutterSurfaceRCS](#) | [grazingang](#) | [depressionang](#) | [surfaceReflectivityLand](#)

**Introduced in R2021a**

# landroughness

Surface height standard deviation for land

## Syntax

```
hgtsd = landroughness(landtype)
[hgtsd,beta0,vegtype] = landroughness(landtype)
```

## Description

`hgtsd = landroughness(landtype)` returns the standard deviation of the surface height for the specified land type.

`[hgtsd,beta0,vegtype] = landroughness(landtype)` in addition to `hgtsd` returns:

- `beta0` — the slope of the land type.
- `vegtype` — the vegetation type.

## Examples

### Land Roughness of Urban Patch

Obtain the standard deviation of the surface height for an urban land.

```
hgtsd = landroughness('Urban')
```

```
hgtsd = 10
```

## Input Arguments

### landtype — Surface land type

"Rugged Mountains" | "Mountains" | "Metropolitan" | "Urban" | "Wooded Hills" | "Rolling Hills" | "Woods" | "Farm" | "Desert" | "Flatland" | "Smooth"

Surface land type, specified as "Rugged Mountains", "Mountains", "Metropolitan", "Urban", "Wooded Hills", "Rolling Hills", "Woods", "Farm", "Desert", "Flatland". or "Smooth".

## Output Arguments

### hgtsd — Standard deviation of the surface height

scalar

Standard deviation of the surface height, returned as a scalar in meters.

### beta0 — Slope of the land type

scalar

Slope of the land type  $\beta_0$ , returned as a scalar in degrees.

**vegtype – Vegetation type**

character array

The vegetation type is a character array determined by the `landtype` input.

Land Type	Vegetation Type
Rugged Mountains	Trees (dense)
Mountains	Trees (dense)
Woods	Trees (dense)
Wooded Hills	Trees (dense)
Rolling Hills	Brush (dense)
Farm	Grass (thin)
Desert	Grass (thin)
Flatland	Grass (thin)
Metropolitan	None
Urban	None
Smooth	None

**References**

[1] Barton, David K. *Radar Equations for Modern Radar*. 1st edition. Norwood, MA: Artech House, 2013.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`searoughness` | `landreflectivity` | `seareflectivity` | `clutterSurfaceRCS` | `radarpropfactor` | `radarvcd` | `blakechart`

**Topics**

“Modeling Target Position Errors due to Refraction”

**Introduced in R2021a**

## seareflectivity

Normalized sea surface reflectivity

### Syntax

```
nrcs = seareflectivity(scale,graz,freq)
nrcs = seareflectivity( ___,Polarization = pol)
nrcs = seareflectivity( ___,ScaleType = scaletype)
nrcs = seareflectivity( ___,Model = model)
nrcs = seareflectivity( ___,LookAngle = lookang)
[nrcs,htgsd,beta0,windvelocity] = seareflectivity( ___ )
```

### Description

`nrcs = seareflectivity(scale,graz,freq)` returns the normalized sea surface reflectivity `nrcs` for the sea state `scale` at the grazing angle `graz` with the transmitted frequency `freq`. In this syntax, sea surface reflectivity is calculated using the NRL Sea Clutter Model by Gregers-Hansen and Mittal. The reflectivity is also called the normalized radar cross section (NRCS) and denoted  $\sigma^0$ .

`nrcs = seareflectivity( ___,Polarization = pol)` also specifies the polarization `pol` of the transmitted wave. Polarization can be horizontal or vertical.

`nrcs = seareflectivity( ___,ScaleType = scaletype)` also specifies the scale type `scaletype` which is either sea state 'SeaState' or wind scale 'WindScale'.

`nrcs = seareflectivity( ___,Model = model)` also specifies the reflectivity `model`.

`nrcs = seareflectivity( ___,LookAngle = lookang)` also specifies the look angle `lookang`.

`[nrcs,htgsd,beta0,windvelocity] = seareflectivity( ___ )` returns additional outputs:

- `htgsd` — Standard deviation of the surface height for the specified sea state number.
- `beta0` — Slope of the sea type. `beta0` is 1.4 times the root mean square (RMS) surface slope. The surface  $\sigma^0$  value for sea clutter reflectivity is computed based on the NRL Sea Clutter Model by Gregers-Hansen and Mittal.
- `windvelocity` — Wind velocity.

### Examples

#### NRCS of Sea Clutter Patch

Calculate the NRCS of a sea clutter patch. Assume that the patch is the sea with sea state number equal to 2 and the radar system operates at a frequency of 30 GHz. Also assume the grazing angle is 10 degrees.

```
scale = 2;
graz = 10;
freq = 30e9;
```

Calculate the normalized NRCS for the sea clutter patch.

```
nrcs = seareflectivity(scale,graz,freq)
nrcs = 2.1555e-04
```

You can use the normalized RCS to calculate the total clutter patch RCS.

### Polarized Sea Reflectivity

Calculate and plot the horizontal and vertical reflectivities from the GIT model. The radar operates at an L-band frequency of 1.5 GHz at grazing angles from 0.1 to 10 degrees. Assume sea state 3.

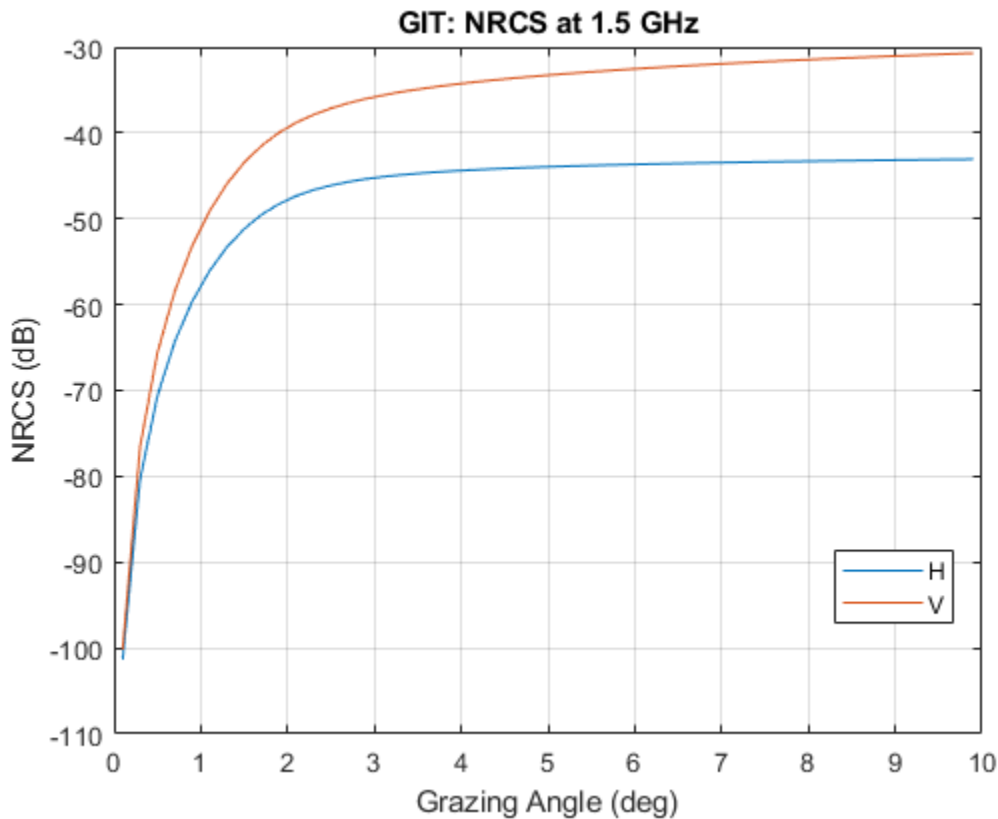
```
seastate = 3;
graz = 0.1:0.2:10;
freq = 1.5e9;
model = 'GIT';
```

Compute the horizontal and vertical polarized reflectivities.

```
reflh = seareflectivity(seastate,graz,freq, ...
    Model = model,Polarization = 'H');
reflv = seareflectivity(seastate,graz,freq, ...
    Model = model,Polarization = 'V');
```

Plot the reflectivities as a function of grazing angle.

```
plot(graz,pow2db(reflh))
hold on
grid on
plot(graz,pow2db(reflv))
legend('H','V','Location','Best')
xlabel('Grazing Angle (deg)')
ylabel('NRCS (dB)')
title('GIT: NRCS at 1.5 GHz')
```



## Input Arguments

### scale — Sea state or wind scale

nonnegative integer

If you set `scaleType` to 'SeaState', `scale` is interpreted as the sea state, specified as a nonnegative scalar between [0,8]. If you set `scaleType` to 'WindScale', `scale` is interpreted as the Beaufort wind scale, specified as a positive scalar between [1,9].

### Dependency

The interpretation of the `scale` argument depends on the value of `scaleType` name-value pair.

### graz — Grazing angle

nonnegative scalar |  $N$ -length vector of grazing values

Grazing angle, specified as a nonnegative scalar or an  $N$ -length row vector of nonnegative values. This argument specifies the grazing angles of the clutter patch relative to the radar. Units are in degrees. See `grazingang`.

### freq — Transmitted frequencies

$10e9$  (default) | positive scalar |  $N$ -length vector of positive values

Transmitted frequencies, specified as a positive scalar or  $N$ -length vector of positive values. Units are in Hz.

Example: `freq = 7*10e9`

### **pol — Polarization of transmitted wave**

'H' (default) | 'V'

Polarization of transmitted wave, specified as 'H' for horizontal polarization or 'V' for vertical polarization.

Example: 'V'

### **scaletype — Scale type**

'SeaState' (default) | 'WindScale'

Scale type, specified as either:

- 'SeaState' — The function uses the Sea State model. When you specify this option, the `scale` input scale must be a nonnegative scalar between [0,8].
- 'WindScale' — The function uses the Beaufort Wind Scale model. When you specify this option, the `scale` input scale must be a positive scalars between [1,9].

Example: 'WindScale'

### **model — Sea reflectivity model**

'NRL' (default) | 'APL' | 'GIT' | 'Hybrid' | 'Masuko' | 'Nathanson' | 'RRE' | 'Sittrop' | 'TSC'

Sea reflectivity model, specified as 'NRL', 'APL', 'GIT', 'Hybrid', 'Masuko', 'Nathanson', 'RRE', 'Sittrop', or 'TSC'. The table “Sea Reflectivity Models” on page 1-165 summarizes the sea surface models available in the radar simulation and their domains of application.

### **lookang — Radar look angle**

0 (default) | nonnegative scalar | 'Upwind' | 'Downwind' | 'Crosswind'

Radar look angle, specified as a nonnegative scalar between 0° and 180° or as:

- 'Upwind' - 0°
- 'Downwind' - 180°
- 'Crosswind' - 90°

Radar look angle is zero when looking upwind.

### **Dependencies**

To enable this argument, set the model name to 'APL', 'GIT', 'Hybrid', 'Masuko', 'Sittrop', or 'TSC'.

Data Types: `double`

## **Output Arguments**

### **nrcs — Normalized surface reflectivity**

real-valued  $N$ -length row vector | real-valued  $M$ -by- $N$  matrix

Normalized surface reflectivity, returned as either a real-valued  $N$ -length row vector or a real-valued  $M$ -by- $N$  matrix. Normalized reflectivity is also called normalized radar cross section.  $M$  is the length

of the grazing angle or depression angle vector  $\mathbf{g}_{\text{raz}}$  and  $N$  is the length of the frequency vector  $\mathbf{f}_{\text{req}}$ .  $n_{\text{rCS}}$  is dimensionless but often expressed as  $\text{m}^2/\text{m}^2$ .

**hgtstd – Standard deviation of surface height**

scalar

Standard deviation of the surface height, returned as a scalar. The model for height deviation, surface slope, and wind velocity is based on a model by Barton. Units are in meters.

**beta0 – Slope of the sea type**

scalar

Slope of the sea type  $\beta_0$ , returned as a scalar. The model for height deviation, surface slope, and wind velocity is based on a model by Barton. Units are in degrees.

**windvelocity – Wind velocity**

scalar

Wind velocity, returned as a scalar. The model for the height deviation, surface slope, and wind velocity is based on a model by Barton. Units are in meters per second.



## More About

### Sea Reflectivity Models

Model	Type	Grazing Angles	Frequency Range	Sea State
<p>'NRL' - Sea Clutter Model due to Gregers-Hansen and Mittal. (Default model)</p> <ul style="list-style-type: none"> <li>• Naval Research Laboratory empirical model for sea reflectivity.</li> <li>• The model does not include variation with azimuth or wind direction.</li> <li>• The model matches experimental results with an absolute deviation of about 2.2 to 2.3 dB for grazing angles from 0.1° to 10°. A deviation of 2.6 dB can be seen for grazing angles above 10° and below 60°.</li> </ul> <p>See [1] and [2].</p>	Empirical model	0.1° - 60°	0.5 - 35 GHz	0 - 6

Model	Type	Grazing Angles	Frequency Range	Sea State
<p>'APL' -</p> <ul style="list-style-type: none"> <li>• John Hopkins Applied Physics Laboratory ADSAM model.</li> <li>• Derived wind velocity from sea state produces less conservative reflectivity values than GIT model at lower sea states.</li> <li>• Takes into account wave height and wave speed.</li> <li>• Differs from the GIT model by deriving wind velocity from sea state.</li> </ul> <p>See [3].</p>	Semi-empirical	0.1 - 10	1 - 100	1 - 6

Model	Type	Grazing Angles	Frequency Range	Sea State
<p>'GIT' -</p> <ul style="list-style-type: none"> <li>• Georgia Institute of Technology</li> <li>• Semi-empirical model based on multipath, wind speed, and wind direction factor.</li> <li>• Takes into account wave height and wave speed.</li> <li>• Derived wind velocity from sea state produces less conservative reflectivity values than GIT at lower sea states.</li> </ul> <p>[5] and [1].</p>	Semi-empirical	0.1 - 10	1 - 100	1 - 6
<p>'Hybrid' -</p> <ul style="list-style-type: none"> <li>• Hybrid model that mixes work by Barton, Nathanson's tables, and GIT semi-empirical models.</li> <li>• May be biased high in the low grazing angle regime.</li> </ul> <p>See [5].</p>	Semi-empirical	0.1 - 30	0.5 - 35	0 - 5

Model	Type	Grazing Angles	Frequency Range	Sea State
<p>'Masuko' --</p> <ul style="list-style-type: none"> <li>Empirical model applicable for medium grazing angles for X and Ka bands.</li> </ul> <p>See [6].</p>	Empirical	30 - 60	X (8 - 12) Ka (26.5 - 40)	1 - 6
<p>'Nathanson' -</p> <ul style="list-style-type: none"> <li>Empirical tables compiled from experimental data that are averages of all wind directions covering UHF to Ka.</li> </ul> <p>See [7].</p>	Empirical	0.1 - 60	UHF (0.3 - 1), L (1 - 2), S(2 - 4), C(4 - 8), X(8 - 12), Ku(12 - 18), Ka(32 - 36)	0 - 6
<p>'RRE' -</p> <ul style="list-style-type: none"> <li>Royal Radar Establishment model</li> <li>Averages over all wind directions.</li> <li>Used extensively in the UK for airborne radar performance assessment.</li> </ul> <p>See [4]</p>	Mathematical	< 10	9 - 10	1 - 6
<p>'Sittrop' -</p> <ul style="list-style-type: none"> <li>Empirical model for lower grazing angles and higher sea states for X-band.</li> </ul> <p>See [4].</p>	Empirical	0.2 - 10	X (8 - 12)	0 - 7

Model	Type	Grazing Angles	Frequency Range	Sea State
'TSC' - <ul style="list-style-type: none"> <li>• Technology Service Corporation Empirical model.</li> <li>• Based on fit to Nathanson tables.</li> <li>• Similar to the GIT model but with values not falling off as rapidly in range</li> <li>• Recommended for conservative performance prediction or when conditions are unknown.</li> </ul> See [5].	Empirical	0.1 - 90	0.5 - 35	0 - 5

## References

- [1] Gregers-Hansen, V. and Mittal, R. "An Improved Empirical Model for Radar Sea Clutter Reflectivity." *NRL/MR/5310-12-9346*, Apr. 27, 2012.
- [2] Barton, David Knox. *Radar Equations for Modern Radar*. Artech House, 2013.
- [3] Reilly, J. P., R. L. McDonald, and G. D. Dockery. "RF-Environment Models for the ADSAM Program." Report No. A1A97U-070, Laurel, MD: Johns Hopkins University Applied Physics Laboratory, August 22, 1997.
- [4] Ward, Keith D., Simon Watts, and Robert J. A. Tough. *Sea Clutter: Scattering, the K-Distribution and Radar Performance*. IET Radar, Sonar, Navigation and Avionics Series 20. London: Institution of Engineering and Technology, 2006.
- [5] Antipov, Irina. "Simulation of Sea Clutter Returns." Department of Defence, June 1998.
- [6] Masuko, Harunobu, Ken'ichi Okamoto, Masanobu Shimada, and Shuntaro Niwa. "Measurement of Microwave Backscattering Signatures of the Ocean Surface Using X Band and K a Band Airborne Scatterometers." *Journal of Geophysical Research* 91, no. C11 (1986): 13065. <https://doi.org/10.1029/JC091iC11p13065>.
- [7] Nathanson, Fred E., et al. *Radar Design Principles: Signal Processing and the Environment*. 2. ed., Repr, Scitech Publ, 2004.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`searoughness` | `landroughness` | `landreflectivity` | `clutterSurfaceRCS` | `surfaceReflectivitySea`

**Introduced in R2021a**

# searoughness

Surface height standard deviation for sea

## Syntax

```
hgtsd = searoughness(scale)
hgtsd = searoughness(scale, 'ScaleType', scaletype)
[hgtsd, beta0, windvelocity] = searoughness( ___ )
```

## Description

`hgtsd = searoughness(scale)` returns the standard deviation of the surface height, `hgtsd`, for the specified sea state number as a scalar in meters.

`hgtsd = searoughness(scale, 'ScaleType', scaletype)` specifies the scale type.

`[hgtsd, beta0, windvelocity] = searoughness( ___ )` returns additional outputs:

- `beta0` — Slope of the sea type in degrees. `beta0` is 1.4 times the root mean square (RMS) surface slope. The surface  $\sigma^0$  value for sea clutter reflectivity is computed based on the NRL Sea Clutter Model by Gregers-Hansen and Mittal
- `windvelocity` — Wind velocity in meters per second

## Examples

### Sea Roughness of Sea State

Obtain the surface height standard deviation in meters assuming a sea state of 2.

```
hgtsd = searoughness(2)
hgtsd = 0.1000
```

## Input Arguments

### scale — Sea state or wind scale

nonnegative integer

If you set `scaletype` to `'SeaState'`, `scale` is interpreted as the sea state, specified as a nonnegative scalar between [0,8]. If you set `scaletype` to `'WindScale'`, `scale` is interpreted as the Beaufort wind scale, specified as a positive scalar between [1,9].

### Dependency

The interpretation of the `scale` argument depends on the value of `scaletype` name-value pair.

### scaletype — Scale type

`'SeaState'` (default) | `'WindScale'`

Scale type, specified as either:

- 'SeaState' — The function uses the Sea State model. When you specify this option, the `scale` input scale must be a nonnegative scalar between [0,8].
- 'WindScale' — The function uses the Beaufort Wind Scale model. When you specify this option, the `scale` input scale must be a positive scalars between [1,9].

Example: 'WindScale'

## Output Arguments

### **hgtstd — Standard deviation of surface height**

scalar

Standard deviation of the surface height, returned as a scalar. The model for height deviation, surface slope, and wind velocity is based on a model by Barton. Units are in meters.

### **beta0 — Slope of the sea type**

scalar

Slope of the sea type  $\beta_0$ , in degrees, returned as a scalar.

### **windvelocity — Wind velocity**

scalar

Wind velocity, returned as a scalar. The model for the height deviation, surface slope, and wind velocity is based on a model by Barton. Units are in meters per second.

## References

[1] Barton, David K. *Radar Equations for Modern Radar*. Norwood, MA: Artech House, 2013.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

landroughness | seareflectivity | landreflectivity | clutterSurfaceRCS | radarpropfactor | radarvcd | blakechart

**Introduced in R2021a**



# atmositu

Use ITU reference atmospheres

## Syntax

```
[T,P,wvden] = atmositu(h)
[T,P,wvden] = atmositu( ____,Name,Value)
atmositu( ____)
```

## Description

`[T,P,wvden] = atmositu(h)` calculates the International Telecommunication Union (ITU) standard atmospheric model known as the Mean Annual Global Reference Atmosphere (MAGRA) and returns the atmospheric temperature *T*, pressure *P*, and water-vapor density *wvden*. MAGRA approximates the U.S. Standard Atmosphere 1976 with insignificant relative error.

`[T,P,wvden] = atmositu( ____,Name,Value)` returns the atmospheric temperature, pressure, and water-vapor density with additional options specified by one or more name-value pairs. For example, `'LatitudeModel','High'` specifies a reference model for latitudes greater than 45°.

`atmositu( ____)` with no output arguments plots:

- Atmospheric temperature *T* versus altitude in linear scale
- Atmospheric pressure *P* versus altitude in logarithmic x-scale
- Atmospheric water-vapor density *wvden* versus altitude in logarithmic x-scale

## Examples

### Find Temperature, Pressure, and Water Vapor Density

Compute the atmospheric temperature, pressure, and water vapor density at a height of 5 km at high latitudes in summer. Use default parameters.

```
h = 5e3;
[T,P,wvden] = atmositu(h)

T = 255.6755
P = 540.4828
wvden = 0.6156
```

### Compute and Visualize Atmospheric Profiles

Compute the atmospheric temperature, pressure, and water-vapor density for a mid-latitude area during winter. Specify an altitude range between 2 km and 88 km.

```
h = (2:88).*1e3;
```

```
[T,P,wvden] = atmositu(h,'LatitudeModel','Mid','Season','Winter');
```

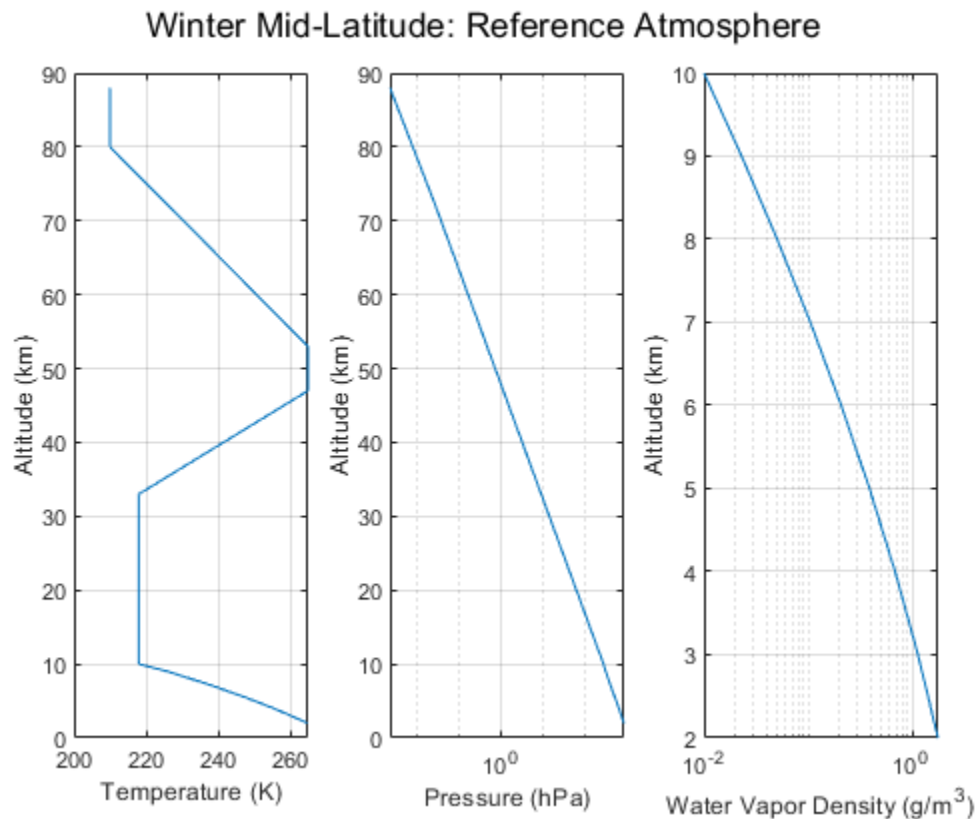
Display the first 6 values of temperature, pressure, and water vapor density.

```
disp([T(1:6)',P(1:6)',wvden(1:6)'])
```

```
264.7771  789.5947  1.7601
260.2759  689.4528  1.1320
255.4229  598.9723  0.6829
250.2181  518.1532  0.3875
244.6615  446.9955  0.2074
238.7531  385.4992  0.1049
```

Plot the atmospheric temperature, pressure, and water-vapor density profiles for the same model.

```
atmositu(h,'Latitude','Mid','Season','Winter')
```



## Input Arguments

### **h** — Geometric heights

*M*-length row vector

Geometric heights corresponding to the altitude above mean sea level (MSL), specified as a row vector. The `atmositu` function returns NaN's for any input value outside of the interval [0,100000]. Units are in meters.

Data Types: double

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `atmositu(h, 'LatitudeModel', 'Mid', 'Season', 'Winter')` specifies the mid-latitude model during winter.

### VaporDensity — Standard ground-level water-vapor density

7.5 (default) | scalar

Standard ground-level water-vapor density in  $\text{g/m}^3$ , specified as a scalar. `VaporDensity` applies only when `LatitudeModel` is set to the default 'Standard' model.

Data Types: double

### ScaleHeight — Scale height

2000 (default) | scalar

Scale height in meters, specified as a scalar. `ScaleHeight` applies only when `LatitudeModel` is set to the default 'Standard' model. For a dry atmosphere, set `ScaleHeight` to 6000.

Data Types: double

### LatitudeModel — Reference latitude model

'Standard' (default) | 'Low' | 'Mid' | 'High'

Reference latitude model, specified as:

- 'Standard' — This is the Mean Annual Global Reference Atmosphere (MAGRA) model that reflects the mean annual temperature and pressure averaged across the world.
- 'Low' — Use this model for latitudes lower than  $22^\circ$ , with little seasonal variation.
- 'Mid' — Use this model for latitudes between  $22^\circ$  and  $45^\circ$  that have seasonal profiles for summer and winter. You can specify a seasonal profile using the `Season` name-value pair.
- 'High' — Use this model for latitudes greater than  $45^\circ$  that have seasonal profiles for summer and winter. You can specify a seasonal profile using the `Season` name-value pair.

### Season — Seasonal profile

'Summer' (default) | 'Winter'

Seasonal profile, specified as 'Summer' or 'Winter'. This argument is valid only when `LatitudeModel` is set to 'Mid' or 'High'.

## Output Arguments

### T — Temperature

$M$ -length row vector

Atmospheric temperature in Kelvin, returned as an  $M$ -length row vector.

---

**Note** The `atmositu`, `gaspl`, and `tropopl` functions use different units for pressure and temperature.

### Pressure and Temperature Units

Function	Pressure Units	Temperature Units
<code>atmositu</code>	hectoPascals (hPa)	kelvin (K)
<code>tropopl</code>	hectoPascals (hPa)	kelvin (K)
<code>gaspl</code>	Pascals (Pa)	Celsius (C)

One hPa equals 100 Pa and  $K = C + 273.15$ . Use caution when combining the use of these three functions.

---

### P — Atmospheric pressure

*M*-length row vector

Atmospheric pressure in hectoPascals, returned as an *M*-length row vector.

### wvden — Water-vapor density

*M*-length row vector

Atmospheric water-vapor density in  $g/m^3$ , returned as an *M*-length row vector.

## References

- [1] International Telecommunication Union (ITU). "Reference Standard Atmospheres". *Recommendation ITU-R P.835-6, P Series, Radiowave Propagation*, December 2017.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`refractiveidx` | `tropopl` | `lenspl`

### Topics

"Modeling Target Position Errors due to Refraction"

### Introduced in R2021a

# lenspl

Calculate loss due to tropospheric lens effect

## Syntax

```
L = lenspl(R,H,EL)
L = lenspl( ____,Name,Value)
```

## Description

`L = lenspl(R,H,EL)` calculates the one-way loss due to the tropospheric lens effect using the International Telecommunication Union (ITU) standard atmospheric model known as the mean annual global reference atmosphere (MAGRA), which approximates the U.S. Standard Atmosphere 1976 with insignificant relative error. The variation in refraction versus altitude makes the atmosphere act like a lens with loss independent of frequency. Rays leaving an antenna are refracted in the troposphere and the energy radiated within some angular extent is distributed over a slightly greater angular sector, thereby reducing the energy density relative to propagation in a vacuum.

`L = lenspl( ____,Name,Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax.

## Examples

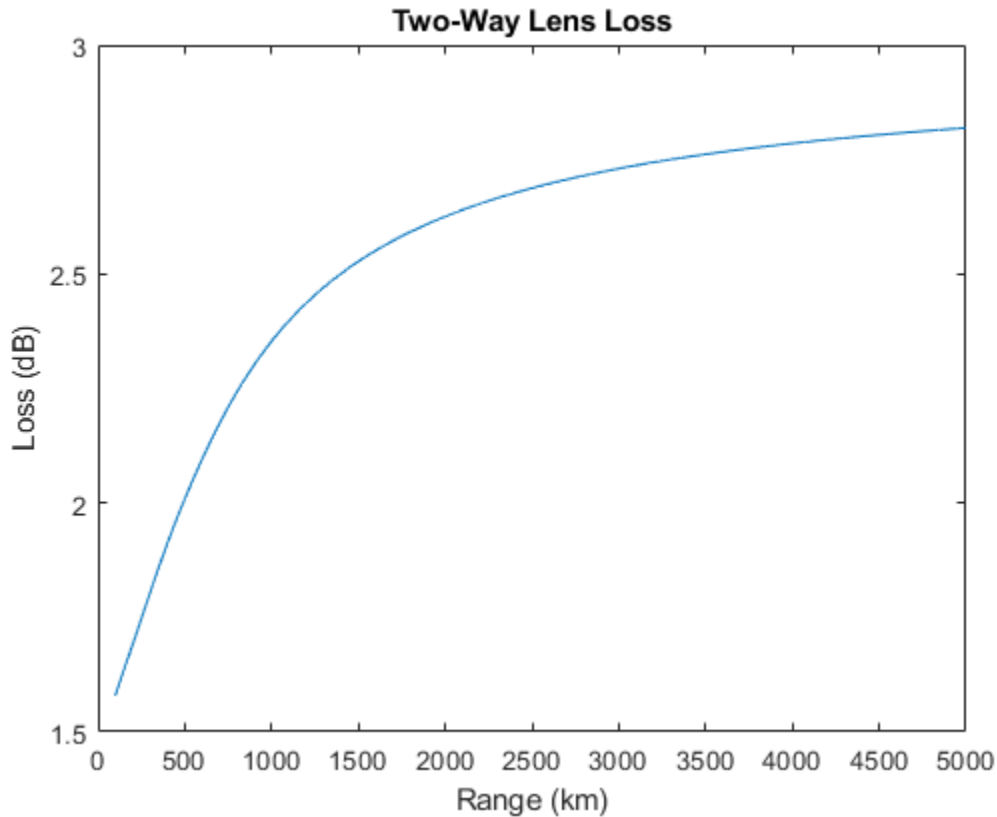
### Plot Two-Way Lens Loss Curve

Calculate the two-way lens loss curve for a radar platform at sea level at an elevation angle of 0.03 deg over a slant range of 0.1 to 5.0 km.

```
h = 0; % m
el = 0.03; % deg
R = (100:5000).*1e3; % m
L = 2*lenspl(R,h,el); % Factor of 2 for two-way propagation
```

Plot the lens loss against the slant range.

```
plot(R.*1e-3,L);
xlabel('Range (km)');
ylabel('Loss (dB)');
title('Two-Way Lens Loss');
```



## Input Arguments

### R — Slant range

positive scalar |  $N$ -length vector

Slant range, specified as a scalar or an  $N$ -length vector. Units are in meters.

Example: 0.5

Data Types: single | double

### H — Altitude of radar platform

scalar in the range [0 100]

Mean sea level (MSL) altitude of the radar platform, specified as a scalar from 0 to 100 km. Values outside the specified range result in NaN output. Units are in meters.

Example: 200e3

Data Types: single | double

### EL — Elevation angle

scalar

Elevation angle of the propagation path, specified as a scalar. Units are in degrees.

Example: 10

Data Types: `single` | `double`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example:

### WaterVaporDensity — Standard ground-level water vapor density

7.5 (default) | positive scalar

Standard ground-level water vapor density, specified as a positive scalar. Applicable only for the default standard model (MAGRA). Units are in grams per meter cubed.

Data Types: `single` | `double`

### ScaleHeight — Altitude above mean sea level

2e3 (default) | positive scalar

Altitude above mean sea level (MSL), specified as a scalar. Applicable only for the default standard model (MAGRA). For dry atmosphere conditions, set to 6e3 m. Units are in meters.

Data Types: `single` | `double`

### LatitudeModel — Reference latitude model

'Standard' (default) | 'Low' | 'Mid' | 'High'

Reference latitude model, specified as one of these.

Model	Description
'Standard' (default)	This model is the mean annual global reference atmosphere (MAGRA) that reflects the mean annual temperature and pressure averaged across the world.
'Low'	This model is for low latitudes less than 22 degrees, where there is little seasonal variation.
'Mid'	This model is for mid latitudes between 22 and 45 degrees with seasonal profiles for 'Summer' and 'Winter', which can be specified using the 'Season' name-value argument.
'High'	This model is for high latitudes greater than 45 degrees with seasonal profiles for 'Summer' and 'Winter', which can be specified using the 'Season' name-value argument.

### Season — Season

'Summer' (default) | 'Winter'

Season for the 'Mid' and 'High' latitude models, specified as 'Summer' or 'Winter'. Other models ignore this input. Defaults to 'Summer'.

**AtmosphereMeasurements — Custom atmospheric measurements***N*-by-4 matrix

Custom atmospheric measurements for the calculation of the refractive index, specified as an *N*-by-4 matrix, where *N* corresponds to the number of altitude measurements. *N* must be greater than or equal to 2. The first column is the atmospheric temperature in kelvins, the second column is the atmospheric pressure in hPa, the third column is the water vapor density in g/m<sup>3</sup>, and the fourth column is the MSL altitude of the measurements in meters. When you use a custom model, all other name-value arguments are ignored and the output refractive index is applicable for the input height.

---

**Note** The model used by `lenspl` assumes geometrical optics conditions, as a result anomalous propagation like ducting and subrefraction cannot be present in provided measurements. If atmospheric measurements evidencing ducting and subrefraction are provided, this function throws an error.

---

Data Types: `single` | `double`**Output Arguments****L — Lens loss**scalar | *M*-length vector

The one-way lens loss, returned as a scalar or *M*-length vector. Units are in decibels.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**`atmositu` | `refractiveidx` | `tropopl` | `gaspl` | `effearthradius`**External Websites**

<https://www.itu.int/rec/R-REC-P.835-6-201712-I/en>

**Introduced in R2021a**



# mergeDetections

Merge detections into clustered detections

## Syntax

```
clusteredDetections = mergeDetections(detections,clusterIndex)
clusteredDetections = mergeDetections( ___,MergingFcn=mergeFcn)
```

## Description

`clusteredDetections = mergeDetections(detections,clusterIndex)` merges detections sharing the same cluster labels. By default, the function merges detections in the same cluster using a Gaussian mixture merging algorithm. The function assumes that all detections in the same cluster share the same `Time`, `SensorIndex`, `ObjectClassID`, `MeasurementParameters`, and `ObjectAttributes` properties or fields.

`clusteredDetections = mergeDetections( ___,MergingFcn=mergeFcn)` specifies the function used to merge the detections in addition to the input arguments from the previous syntax.

## Examples

### Merge Detections to Generate Clustered Detections

Generate two clusters of detections with two false alarms.

```
rng(2021) % For repeatable results
x1 = [5; 5; 0] + randn(3,4); % Four detections in cluster one
x2 = [5; -5; 0] + randn(3,4); % Four detections in cluster two
xFalse = 30*randn(3,2); % Two false alarms
x = [x1 x2 xFalse];
```

Format these detections into a cell array of `objectDetection` objects.

```
detections = repmat({objectDetection(0,[0; 0; 0])},10,1);
for i = 1:10
    detections{i}.Measurement = x(:,i);
end
```

Define the cluster indices according to the previously defined scenario. You can typically obtain the cluster indices by applying a clustering algorithm on the detections.

```
clusterIndex = [1; 1; 1; 1; 2; 2; 2; 2; 3; 4];
```

Use the `mergeDetections` function to merge the detections.

```
clusteredDetections = mergeDetections(detections,clusterIndex);
```

Visualize the results in a theater plot.

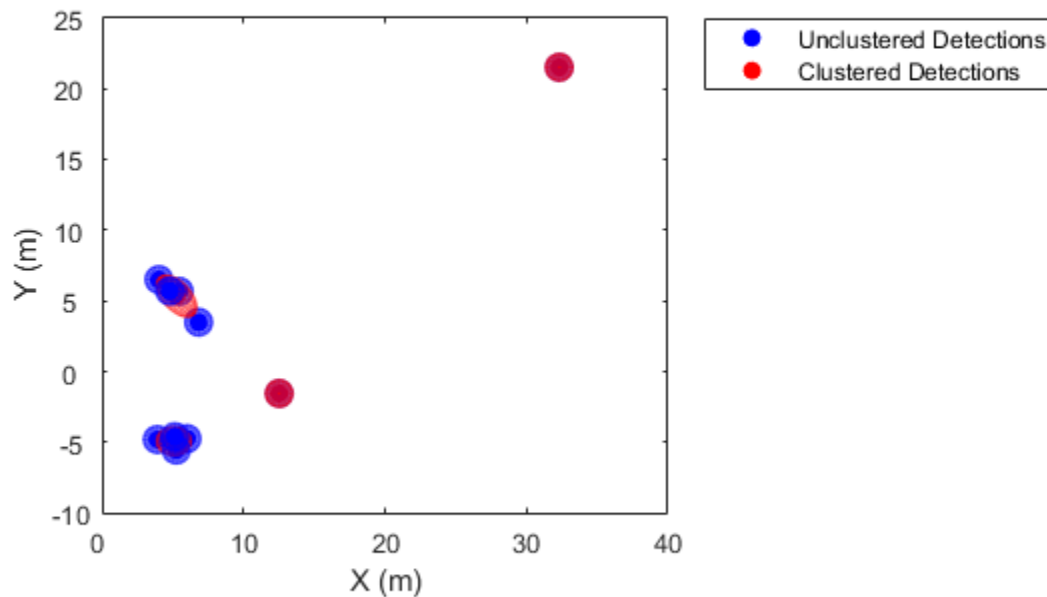
```
% Create a theaterPlot object.
tp = theaterPlot;
```

```
% Create two detection plotters, one for unclustered detections and one for
% clustered detections.
detPlotterUn = detectionPlotter(tp,DisplayName="Unclustered Detections", ...
    MarkerFaceColor="b",MarkerEdgeColor="b");
detPlotterC = detectionPlotter(tp,DisplayName="Clustered Detections", ...
    MarkerFaceColor="r",MarkerEdgeColor="r");

% Concatenate measurements and covariances for unclustered detections
detArray = [detections{:}];
xUn = horzcat(detArray.Measurement)';
PUn = cat(3,detArray.MeasurementNoise);

% Concatenate measurements and covariance for clustered detections
clusteredDetArray = [clusteredDetections{:}];
xC = horzcat(clusteredDetArray.Measurement)';
PC = cat(3,clusteredDetArray.MeasurementNoise);

% Plot all unclustered and clustered detections
plotDetection(detPlotterUn,xUn,PUn);
plotDetection(detPlotterC,xC,PC);
```



## Input Arguments

### **detections** — Object detections

*N*-element array of `objectDetection` objects | *N*-element cell array of `objectDetection` objects | *N*-element array of structures

Object detections, specified as an *N*-element array of `objectDetection` objects, *N*-element cell array of `objectDetection` objects, or an *N*-element array of structures whose field names are the same as the property names of the `objectDetection` object. *N* is the number of detections. You can create `detections` directly, or you can obtain `detections` from the outputs of sensor objects such as `fusionRadarSensor`, `irSensor`, and `sonarSensor`.

### **clusterIndex** — Cluster indices

*N*-element vector of positive integers

Cluster indices, specified as an *N*-element vector of positive integers, where *N* is the number of detections specified in the `detections` input. Each element is the cluster index of the corresponding detection in the `detections` input. For example, if `clusterIndex(i)=k`, then the *i*th detection from the `detections` input belongs to cluster *k*.

### **mergeFcn** — Function to merge detections

function handle

Function to merge detections, specified as a function handle. The function must use this syntax:

```
detectionOut = mergeFcn(detectionsIn)
```

where:

- `detectionsIn` is specified as a cell array of `objectDetection` objects (in the same cluster).
- `detectionOut` is returned as an `objectDetection` object.

Example: `@mergeFcn`

## Output Arguments

### **clusteredDetections** — Clustered detections

*M*-element cell array of `objectDetection` objects

Clustered detections, returned as an *M*-element cell array of `objectDetection` objects, where *M* is the number of unique cluster indices specified in the `clusterIndex` input.

## See Also

`clusterDBSCAN`

**Introduced in R2021b**

## radarpropfactor

One-way radar propagation factor

### Syntax

```
F = radarpropfactor(R, freq, ANHT)
F = radarpropfactor( ___, TGTHT)
F = radarpropfactor( ___, Name, Value)
radarpropfactor( ___)
```

### Description

`F = radarpropfactor(R, freq, ANHT)` calculates the one-way propagation factor assuming a surface target and a sea state of 0. The calculation estimates the complex relative permittivity (dielectric constant) of the reflecting surface using a sea water model described in [1] that is valid from 100 MHz to 10 GHz. The target height is assumed to be the height of significant clutter sources above the average surface height. Specifically, the target height is calculated as 3 times the standard deviation of the surface height. Assuming the paths are the same, the two-way propagation factor is 2F. Atmospheric refraction is taken into account through the use of an `EffectiveEarthRadius` that can be specified. Scattering and ducting are assumed to be negligible.

`F = radarpropfactor( ___, TGTHT)` calculates the target propagation factor assuming a target height of TGTHT.

`F = radarpropfactor( ___, Name, Value)` allows you to specify additional input parameters as Name-Value arguments. You can specify additional name-value pair arguments in any order. This syntax can use any of the input arguments in the previous syntax.

`radarpropfactor( ___)` plots the one-way propagation factor in dB versus range in km. Default range units are km.

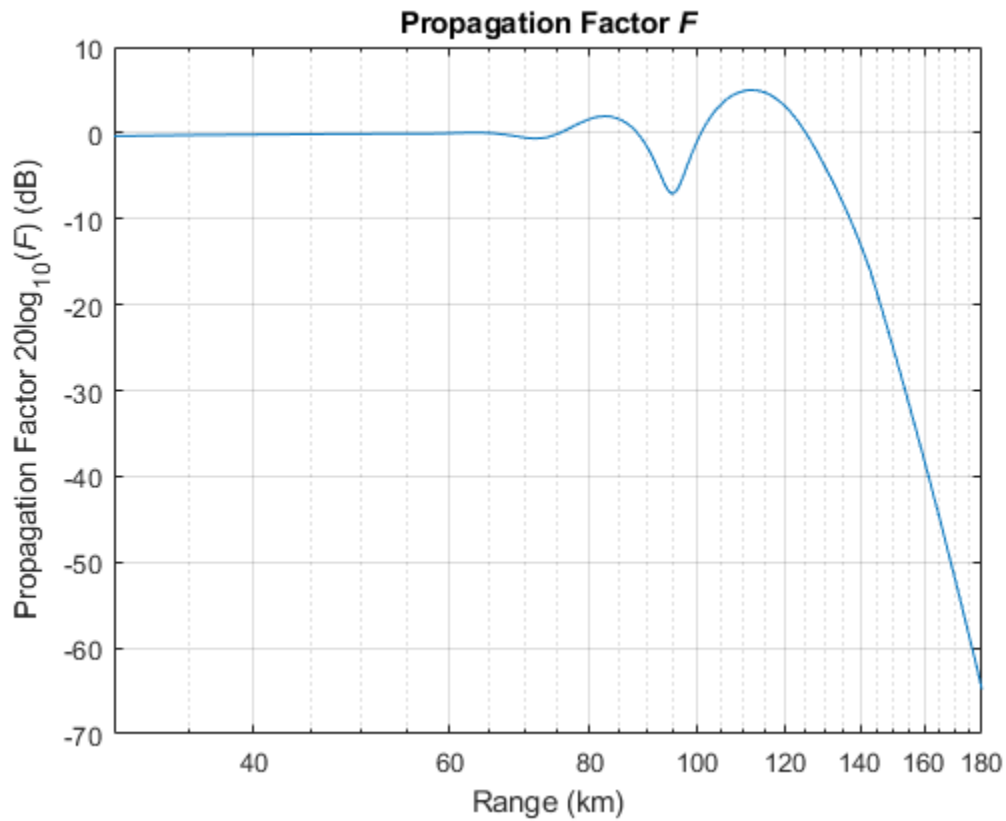
### Examples

#### Plot Propagation Factor for 3 GHz S-band Radar

Plot the propagation factor for a 3 GHz S-band radar assuming an antenna height of 10 m and a target height of 1 km. Assume that the surface has a height standard deviation of 1 m, and the surface slope is 0.05 degrees.

```
R      = (30:0.5:180)*1e3; % Range (m)
freq   = 3e9;             % Frequency (Hz)
anht   = 10;             % Radar height (m)
tgtht  = 1e3;            % Target height (m)
hgtsd  = 1;              % Height standard deviation (m)
beta0  = 0.05;           % Surface slope (deg)
```

```
radarpropfactor(R, freq, anht, tgtht, ...
    'SurfaceHeightStandardDeviation', hgtsd, ...
    'SurfaceSlope', beta0)
```



```
ans = 301x1
```

```
-0.3696
-0.3566
-0.3439
-0.3316
-0.3197
-0.3082
-0.2970
-0.2862
-0.2756
-0.2654
⋮
```

## Input Arguments

**R — Free space range**  
 scalar |  $M$ -length vector

Free space range, specified as a scalar or an  $M$ -length vector. Units are in meters.

Example: 0.5

Data Types: single | double

**freq — Radar frequency**

positive real scalar | vector

Radar frequency in hertz, specified as a positive real scalar or a vector.

Data Types: double

**ANHT — Antenna height**

positive scalar

Antenna height as referenced from the surface, specified as a positive scalar. Units are in meters.

Data Types: double

**TGHT — Target height**

positive scalar

Target height as referenced from the surface, specified as a positive scalar. Units are in meters.

Data Types: double

**Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'SurfaceHeightStandardDeviation', hgtstd, 'SurfaceSlope', beta0`

**Polarization — Polarization of transmitted wave**

'H' (default) | 'V'

Polarization of the transmitted wave, specified as 'H' or 'V'. 'H' indicates horizontal polarization and 'V' indicates vertical polarization.

**SurfaceRelativePermittivity — Complex relative permittivity**

complex scalar

Complex relative permittivity (dielectric constant) of the reflecting surface, specified as a complex scalar. The default value of dielectric constant depends on the value of the `freq` argument. The function uses a sea water model in [1] that is valid up to 10 GHz.

Data Types: single | double

Complex Number Support: Yes

**SurfaceHeightStandardDeviation — Standard deviation of surface height**

0.01 (default) | positive scalar

Standard deviation of the surface height in meters, specified as positive scalar. The default value of 0.01 m indicates a sea state of 0. Units are in meters.

Data Types: single | double

**SurfaceSlope — Surface slope**

nonnegative scalar

Surface slope, specified as a nonnegative scalar. This value is expected to be 1.4 times the RMS surface slope. Given the condition that

$$2 \times \text{GRAZ}/\beta_0 < 1,$$

where GRAZ is the grazing angle of the geometry specified in degrees and  $\beta_0$  is the surface slope, the effective surface height standard deviation in meters is calculated as

$$\text{Effective HGTSD} = \text{HGTSD} \times (2 \times \text{GRAZ}/\beta_0)^{1/5}.$$

This calculation better accounts for shadowing. Otherwise, the effective height standard deviation is equal to HGTSD. This argument defaults to the surface slope value output by the `searoughness` function for a sea state of 0. Units are in degrees.

Data Types: `double`

### **VegetationType — Vegetation type**

'None' (default) | 'Trees' | 'Brush' | 'Weeds' | 'Grass'

Surface vegetation type, specified as 'Trees', 'Weeds', and 'Brush' are assumed to be dense vegetation. 'Grass' is assumed to be thin grass. Use this argument when using the function on surfaces different from the sea.

### **ElevationBeamwidth — Half-power elevation beamwidth**

10 (default) | scalar between 0° and 90°

Half-power elevation beamwidth in degrees, specified as a scalar between 0° and 90°. The elevation beamwidth is used in the calculation of a sinc antenna pattern. The default antenna pattern is symmetric with respect to the beam maximum and is of the form  $\sin(u)/u$ . The parameter  $u$  is given by  $u = k \sin(\theta)$ , where  $\theta$  is the elevation angle in radians and  $k$  is given by  $k = x_0 / \sin(\pi \times \text{ELBW}/360)$ , where ELBW is the half-power elevation beamwidth and  $x_0 \approx 1.3915573$  is a solution of  $\sin(x) = x/\sqrt{2}$ .

Data Types: `double`

### **AntennaPattern — Antenna elevation pattern**

real-valued column vector

Antenna elevation pattern, specified as a real-valued column vector. Values for 'AntennaPattern' must be specified together with values for 'PatternAngles'. Both vectors must have the same size. If both an antenna pattern and an elevation beamwidth are specified, `radarpropfactor` uses the antenna pattern and ignores the elevation beamwidth value. This argument defaults to a sinc antenna pattern.

Example: `cosd([-90:90])`

Data Types: `double`

### **PatternAngles — Antenna pattern elevation angles**

real-valued column vector

Antenna pattern elevation angles specified as a real-valued column vector. The size of the vector specified by `PatternAngles` must be the same as that specified by `AntennaPattern`. Angle units are expressed in degrees and must lie between -90° and 90°. In general, the antenna pattern should fill the whole range from -90° to 90° for the coverage to be computed properly.

Example: `[-90:90]`

Data Types: `double`

### **TiltAngle — Antenna tilt angle**

0 (default) | real-valued scalar

Antenna tilt angle, specified as a real-valued scalar. The tilt angle is the elevation angle of the antenna with respect to the surface. Angle units are expressed in degrees.

Example: 10

Data Types: double

### **EffectiveEarthRadius — Effective Earth radius**

positive scalar

Effective Earth radius in meters, specified as a positive scalar. The effective Earth radius is an approximation used for modeling refraction effects in the troposphere. The default value calculates the effective Earth radius using a refraction gradient of  $-39\text{e-}9$ , which results in approximately 4/3 of the real Earth radius.

Data Types: double

### **RefractiveIndex — Refractive index at surface**

1.000318 (default) | scalar greater than 1

Refractive index at the surface, specified as a nonnegative scalar. Defaults to approximately 1.000318, which is the output of the `refractiveidx` function at an altitude of 0 meters.

Data Types: double

## **Output Arguments**

### **F — One-way propagation factor**

scalar |  $M$ -length vector

The one-way propagation factor, returned as a scalar or  $M$ -length column vector. Units are in decibels.

## **References**

[1] Blake, L.V. "Machine Plotting of Radar Vertical-Plane Coverage Diagrams." Naval Research Laboratory, 1970 (NRL Report 7098).

[2] Barton, David K. *Radar Equations for Modern Radar*. Norwood, MA: Artech House, 2013.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Apps**

**Radar Designer**

### **Functions**

`blakechart` | `el2height` | `height2el` | `height2range` | `height2grndrange` | `landroughness` | `radarvcd` | `range2height` | `refractionexp` | `searoughness`



**Introduced in R2021a**

## refractiveidx

Calculates the refractive index

### Syntax

```
ridx = refractiveidx(h)
ridx = refractiveidx( ___,Name,Value)
[ridx,N] = refractiveidx( ___ )
refractiveidx( ___ )
```

### Description

`ridx = refractiveidx(h)` calculates the refractive index `ridx` at height `h` above mean sea level (MSL) using the International Telecommunication Union (ITU) standard atmospheric model.

`ridx = refractiveidx( ___,Name,Value)` calculates the refractive index with additional options specified by one or more name-value pairs.

`[ridx,N] = refractiveidx( ___ )` additionally outputs the refractivity `N` as a row vector.

`refractiveidx( ___ )` with no output arguments plots the refractive index `n` as a function of altitude in kilometers.

### Examples

#### Compute Refractive Index and Refractivity

Compute the refractive index and refractivity at a height of 20 km using the mid-latitude model during winter.

```
h = 20e3;
```

```
[ridx,N] = refractiveidx(h,'LatitudeModel','Mid','Season','Winter')
```

```
ridx = 1.0000
```

```
N = 21.1961
```

### Input Arguments

#### **h** — Geometric heights

row vector

Geometric heights corresponding to the altitude above MSL in meters, specified as a row vector.

Data Types: double

## Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `refractiveidx(h,'LatitudeModel','Mid','Season','Winter')` specifies the mid-latitude model during winter.

## VaporDensity — Standard ground-level water-vapor density

7.5 (default) | scalar

Standard ground-level water-vapor density in  $\text{g/m}^3$ , specified as a scalar. `VaporDensity` applies only when `LatitudeModel` is set to the default 'Standard' model.

Data Types: double

## ScaleHeight — Scale height

2000 (default) | scalar

Scale height in meters, specified as a scalar. `ScaleHeight` applies only when `LatitudeModel` is set to the default 'Standard' model. For a dry atmosphere, set `ScaleHeight` to 6000.

Data Types: double

## LatitudeModel — Reference latitude model

'Standard' (default) | string

Reference latitude model, specified as a string vector. Specify `LatitudeModel` as:

- 'Standard'

This model is the Mean Annual Global Reference Atmosphere (MAGRA) that reflects the mean annual temperature and pressure averaged across the world.

- 'Low'

Use this option for low latitudes less than  $22^\circ$ , where there exists little seasonal variation.

- 'Mid'

Use this option for mid-latitudes between  $22^\circ$  and  $45^\circ$  that have seasonal profiles for summer and winter, which can be specified using the `Season` name-value pair.

- 'High'

Use this option for high latitudes greater than  $45^\circ$  that have seasonal profiles for summer and winter, which can be specified using the `Season` name-value pair.

## Season — Seasonal profile

'Summer' (default) | 'Winter'

Seasonal profile, specified as 'Summer' or 'Winter'. This argument is valid only when `LatitudeModel` is set to 'Mid' or 'High'.

## AtmosphereMeasurements — Custom atmospheric measurements

$N$ -by-4 matrix

Custom atmospheric measurements for the calculation of `ridx`, specified as an  $N$ -by-4 matrix where  $N$  corresponds to the number of altitude measurements. The first column in  $N$  is the atmospheric temperature in Kelvin, the second column is the atmospheric pressure in hectopascals, the third column is the atmospheric water-vapor density in  $\text{g/m}^3$ , and the fourth column is the altitude above MSL of the measurements in increasing order and specified in meters. When `AtmosphereMeasurements` is specified, all other name-value pair options are ignored and `ridx` is applicable for the input height  $h$ .

## Output Arguments

### **ridx** — Refractive index

row vector

Refractive index, returned as a row vector.

### **N** — Refractivity

row vector

Refractivity, returned as a row vector.

## References

- [1] International Telecommunication Union (ITU). "The Radio Refractive Index: Its Formula and Refractivity Data". *Recommendation ITU-R P.453-11*, P Series, Radiowave Propagation, July 2015.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

`atmositu` | `tropopl` | `lenspl` | `effearthradius`

### **Topics**

"Radar Vertical Coverage over Terrain"  
"Modeling Target Position Errors due to Refraction"

### **Introduced in R2021a**

# snowpl

Path loss due to wet snow

## Syntax

```
l = snowpl(r,f,rs)
l = snowpl( ____,Name,Value)
```

## Description

`l = snowpl(r,f,rs)` returns the one-way path loss `l` due to snow using the Gunn-East model for RF frequencies.

`l = snowpl( ____,Name,Value)` returns the one-way path loss with additional options specified by one or more name-value pairs. For example, 'Type','Dry' specifies dry snow.

## Examples

### Calculate Path Loss Due to Snow

Calculate the one-way path loss due to snow for an RF transmission of 77 GHz at a range of 10 km. The snow equivalent precipitation rate is 0.75 mm/h.

```
r = 10e3;
f = 77e9;
rs = 0.75;

l = snowpl(r,f,rs)

l = 1.0017
```

## Input Arguments

### **r** — Propagation distances

*M*-length vector

Propagation distances in meters, specified as an *M*-length vector.

Data Types: double

### **f** — Signal carrier frequency

*N*-length vector

Signal carrier frequency in hertz, specified as an *N*-length vector.

Data Types: double

### **rs** — Equivalent liquid water content

scalar

Equivalent liquid water content, specified as a scalar expressed in mm/h.

Data Types: `double`

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `snowpl(r,f,rs,'SnowModel','ITU','Type','Dry')` specifies the ITU snow model with dry snow.

### **SnowModel – Snow model**

'GunnEast' (default) | 'ITU'

Snow model, specified as 'GunnEast' or 'ITU'. Use the 'GunnEast' model for RF frequencies and the 'ITU' model for optical frequencies.

### **Type – Type of snow**

'Wet' (default) | 'Dry'

Type of snow, specified as 'Wet' or 'Dry'. 'Type' applies only when `SnowModel` is set to 'ITU'. The function ignores this input when `SnowModel` is set to 'GunnEast'.

## **Output Arguments**

### **L – Path loss**

*M*-by-*N* matrix

Path loss of each propagation path under the corresponding frequency, returned as an *M*-by-*N* matrix.

## **References**

- [1] Gunn, K. L. S., and T. W. R. East. "The Microwave Properties of Precipitation Particles." *Quarterly Journal of the Royal Meteorological Society* 80, no. 346 (October 1954): 522–45. <https://doi.org/10.1002/qj.49708034603>.
- [2] International Telecommunication Union (ITU). "Propagation Data Required for the Design of Terrestrial Free-Space Optical Links". *Recommendation ITU-R P.1817-1, P Series, Radiowave Propagation*, February 2012.
- [3] Nakaya, Ukitiro, and Tôiti Jr Terada. "Simultaneous Observations of the Mass, Falling Velocity and Form of Individual Snow Crystals." *Journal of the Faculty of Science*, vol.1, no. 7 (January 30, 1935): 191–200.
- [4] Richards, M. A., Jim Scheer, William A. Holm, and William L. Melvin, eds. *Principles of Modern Radar*. Raleigh, NC: SciTech Pub, 2010.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

#### Functions

`cranerainpl` | `fogpl` | `fspl` | `gaspl` | `lenspl` | `rainpl` | `tropopl`

**Introduced in R2021a**

## tropopl

Slant-path loss due to atmosphere gaseous absorption

### Syntax

```
Lgas = tropopl(R,F,H,EL)
Lgas = tropopl( ____,Name,Value)
[Lgas,Llens] = tropopl( ____)
```

### Description

`Lgas = tropopl(R,F,H,EL)` calculates the path loss due to tropospheric refraction using the International Telecommunication Union (ITU) standard atmospheric model known as the mean annual global reference atmosphere (MAGRA), which approximates the U.S. Standard Atmosphere 1976 with insignificant relative error.

`Lgas = tropopl( ____,Name,Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax.

`[Lgas,Llens] = tropopl( ____)` calculates the corresponding lens loss. The variation in refractivity versus altitude makes the atmosphere act like a lens with loss independent of frequency. Rays leaving an antenna are refracted in the troposphere and the energy radiated within some angular extent is distributed over a slightly greater angular sector, thereby reducing the energy density relative to propagation in a vacuum.

### Examples

#### Plot Attenuation Versus Range for 100 GHz Radar Frequency

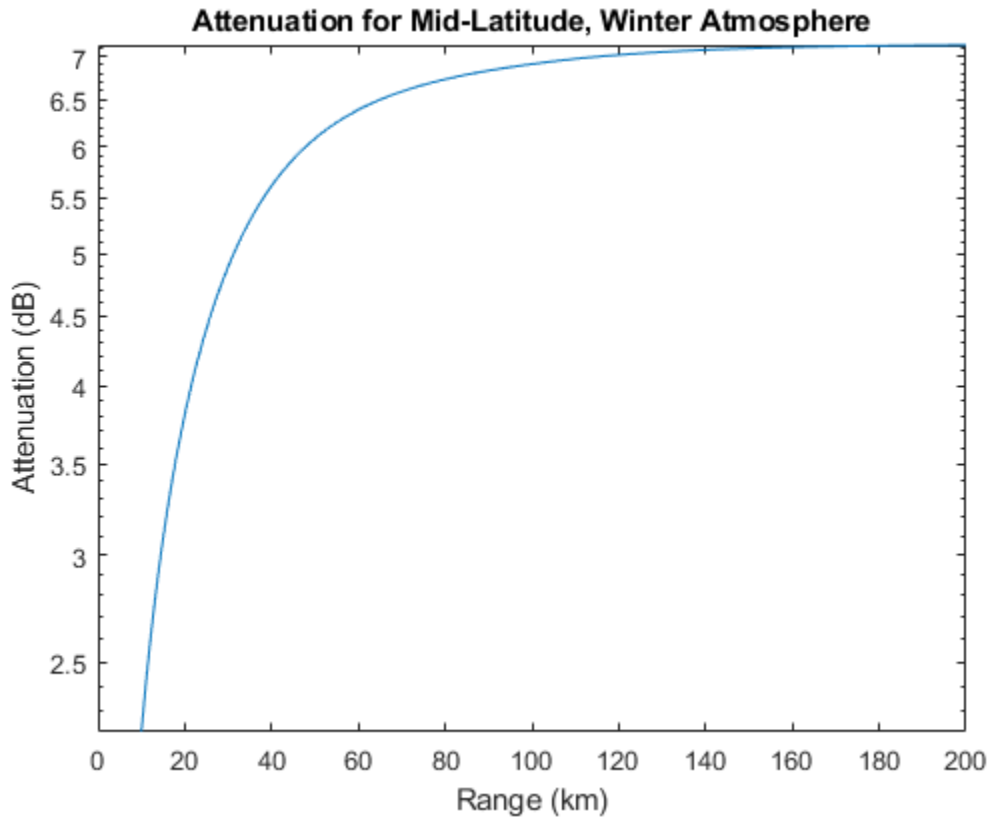
Calculate the attenuation versus range for a frequency of 100 GHz with an elevation of 5 degrees using the mid-latitude, winter atmospheric model.

```
R = (10:200)*1e3;      % m
f = 100e9;            % Hz
ht = 0;               % m
el = 5;               % deg
Lgas = tropopl(R,f,ht,el,'LatitudeModel','Mid','Season','Winter');
```

Plot the results.

```
semilogy(R.*1e-3,Lgas);
xlabel('Range (km)');
ylabel('Attenuation (dB)');
title('Attenuation for Mid-Latitude, Winter Atmosphere');
```





## Input Arguments

### **R — Slant range**

positive scalar |  $M$ -length vector

Slant range, specified as a positive scalar or an  $M$ -length column vector. Units are in meters.

Data Types: single | double

### **F — Radar frequency**

positive scalar |  $N$ -length vector

Radar frequency, specified as a positive real scalar or  $N$ -length row vector. Units are in Hz.

Data Types: single | double

### **H — Altitude of radar platform**

positive scalar

Mean sea level (MSL) altitude of the radar platform, specified as a positive scalar from 0 to 100 km. Values outside the specified range result in NaN output. Units are in meters.

Example: 200e3

Data Types: single | double

**EL — Elevation angle**

scalar

Elevation angle of the propagation path, specified as a scalar. Units are in degrees.

Example: 10

Data Types: single | double

**Name-Value Pair Arguments**

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'LatitudeModel', 'Mid', 'Season', 'Winter'

**WaterVaporDensity — Standard ground-level water vapor density**

7.5 (default) | positive scalar

Standard ground-level water vapor density, specified as a positive scalar in g/m<sup>3</sup>. Applicable only for the default standard model (MAGRA). Defaults to 7.5 g/m<sup>3</sup>.

Data Types: double

**ScaleHeight — Scale height above mean sea level**

2e3 (default) | positive scalar

Altitude above mean sea level (MSL), specified as a positive scalar in meters. Applicable only for the default standard model (MAGRA). Defaults to 2e3 meters. For a dry atmospheric conditions, set scale height to 6e3 meters.

Data Types: double

**LatitudeModel — Reference latitude model**

'Standard' (default) | 'Low' | 'Mid' | 'High'

Reference latitude model, specified as one of these.

Model	Description
'Standard' (default)	This model is the mean annual global reference atmosphere (MAGRA) that reflects the mean annual temperature and pressure averaged across the world.
'Low'	This model is for low latitudes less than 22 degrees, where there is little seasonal variation.
'Mid'	This model is for mid latitudes between 22 and 45 degrees with seasonal profiles for 'Summer' and 'Winter', which can be specified using the 'Season' name-value argument.

Model	Description
'High'	This model is for high latitudes greater than 45 degrees with seasonal profiles for 'Summer' and 'Winter', which can be specified using the 'Season' name-value argument.

### Season — Season

'Summer' (default) | 'Winter'

Season for the 'Mid' and 'High' latitude models, specified as 'Summer' or 'Winter'. Other models ignore this input. Defaults to 'Summer'.

### AtmosphereMeasurements — Custom atmosphere measurements

*N*-by-4 matrix

Custom atmospheric measurements for the calculation of the refractive index, specified as an *N*-by-4 matrix, where *N* corresponds to the number of altitude measurements. *N* must be greater than or equal to 2. The first column is the atmospheric temperature in kelvins, the second column is the atmospheric pressure in hPa, the third column is the water vapor density in g/m<sup>3</sup>, and the fourth column is the MSL altitude of the measurements in meters. When you use a custom model, all other name-value arguments are ignored and the output refractive index is applicable for the input height.

---

**Note** The model used by `lenspl` assumes geometrical optics conditions, as a result anomalous propagation like ducting and sub-refraction cannot be present in provided measurements. If atmospheric measurements evidencing ducting and sub-refraction are provided, this function throws an error.

---



---

**Note** The `atmositu`, `gaspl`, and `tropopl` functions use different units for pressure and temperature.

---

### Pressure and Temperature Units

Function	Pressure Units	Temperature Units
<code>atmositu</code>	hectoPascals (hPa)	kelvin (K)
<code>tropopl</code>	hectoPascals (hPa)	kelvin (K)
<code>gaspl</code>	Pascals (Pa)	Celsius (C)

One hPa equals 100 Pa and  $K = C + 273.15$ . Use caution when combining the use of these three functions.

---

Data Types: `single` | `double`

## Output Arguments

### Lgas — Path loss

*M*-by-*N* matrix

Path loss due to tropospheric refraction, specified as an *M*-by-*N* matrix. *M* and *N* are defined by the slant range, *R*, and frequency, *F*, arguments, respectively. Units are in decibels (dB).

**Llens — One-way lens loss***M*-by-*N* matrix

One-way lens loss, specified as an *M*-by-*N* matrix for elevation angles less than 50 deg. *M* and *N* are defined by the slant range, *R*, and frequency, *F*, arguments, respectively. Units are in decibels (dB).

Data Types: double

**More About****Layered Atmosphere**

The atmosphere is a used in tropopl is a layered model with temperature, pressure, and water vapor density dependent on altitude.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

atmositu | refractiveidx | lenspl | gaspl | effearthradius

**External Websites**

<https://www.itu.int/rec/R-REC-P.835-6-201712-I/en>

**Introduced in R2021a**

# coverageConfig

Sensor and emitter coverage configuration

## Syntax

```
configs = coverageConfig(sc)
configs = coverageConfig(sensors)
configs = coverageConfig(sensors,positions,orientations)
```

## Description

`configs = coverageConfig(sc)` returns sensor coverage configuration structures in a radar scenario `sc`.

`configs = coverageConfig(sensors)` returns sensor coverage configuration structures from a list of sensors and emitters.

`configs = coverageConfig(sensors,positions,orientations)` allows you to specify the position and orientation of the platform on which each sensor or emitter is mounted.

## Examples

### Obtain Coverage Configuration

Create a radar sensor and a radar emitter.

```
radar = radarDataGenerator(1,'Rotator');
emitter = radarEmitter(2);
```

Obtain coverage configurations based on the sensor's position information.

```
cfgs = coverageConfig({radar,emitter})
```

*cfgs=2x1 struct array with fields:*

```
Index
LookAngle
FieldOfView
ScanLimits
Range
Position
Orientation
```

```
cfgs2 = coverageConfig({radar, emitter},[1000 0 0 ; 0 1000 0])
```

*cfgs2=2x1 struct array with fields:*

```
Index
LookAngle
FieldOfView
ScanLimits
Range
```

```
Position  
Orientation
```

Create a radar scenario and add the radar sensor and the radar emitter to the scenario.

```
sc = radarScenario;  
plat = platform(sc);  
plat.Sensors = {radar};  
plat.Emitters = {emitter};
```

Obtain all coverage configurations in the scenario.

```
cfgScenario = coverageConfig(sc)
```

```
cfgScenario=2×1 struct array with fields:  
Index  
LookAngle  
FieldOfView  
ScanLimits  
Range  
Position  
Orientation
```

## Input Arguments

### **sc — Radar scenario**

radarScenario object

Radar scenario, specified as a radarScenario object.

### **sensors — Sensors or emitters**

sensor or emitter object |  $N$ -element cell array of sensor or emitter objects

Sensors or emitters, specified as a sensor or emitter object, or an  $N$ -element cell array of sensor or emitter objects, where  $N$  is the number of sensor or emitter objects. The applicable sensor or emitter objects include radarDataGenerator and radarEmitter.

### **positions — Position of sensor or emitter's platform**

$N$ -by-3 matrix of scalars

Position of sensor or emitter's platform, specified as an  $N$ -by-3 matrix of scalars in meters. The  $i$ th row of the matrix is the  $[x, y, z]$  Cartesian coordinates of the  $i$ th sensor or emitter's platform.

### **orientations — Orientation of sensor or emitter's platform**

$N$ -by-1 vector of quaternions

Orientation of sensor or emitter's platform, specified as an  $N$ -by-1 vector of quaternions. The  $i$ th quaternion in the vector represents the rotation from the global or scenario frame to the  $i$ th sensor or emitter's platform frame.

## Output Arguments

### **configs** — Sensor or emitter coverage configurations

*N*-element array of configuration structures

Sensor or emitter coverage configurations, returned as an *N*-element array of configuration structures. *N* is the number of sensor or emitter objects specified in the `sensors` input or the number of sensors or emitters contained in the `radarScenario` object `sc`. Each configuration structure contains seven fields:

#### Fields of configurations

Field	Description
Index	A unique integer to distinguish sensors or emitters.
LookAngle	The current boresight angles of the sensor or emitter, specified as: <ul style="list-style-type: none"> <li>• A scalar in degrees if scanning only in the azimuth direction.</li> <li>• A two-element vector [azimuth; elevation] in degrees if scanning both in the azimuth and elevation directions.</li> </ul>
FieldOfView	The field of view of the sensor or emitter, specified as a two-element vector [azimuth; elevation] in degrees.
ScanLimits	The minimum and maximum angles the sensor or emitter can scan from its Orientation. <ul style="list-style-type: none"> <li>• If the sensor or emitter can only scan in the azimuth direction, specify the limits as a 1-by-2 row vector [minAz, maxAz] in degrees.</li> <li>• If the sensor or emitter can also scan in the elevation direction, specify the limits as a 2-by-2 matrix [minAz, maxAz; minEl, maxEl] in degrees.</li> </ul>
Range	The range of the beam and coverage area of the sensor or emitter in meters.
Position	The origin position of the sensor or emitter, specified as a three-element vector [X, Y, Z] on the theater plot's axes.
Orientation	The rotation transformation from the scenario or global frame to the sensor or emitter mounting frame, specified as a rotation matrix, a quaternion, or three Euler angles in ZYX sequence.

You can use `configs` to plot the sensor coverage in a `theaterPlot` using its `plotCoverage` object function.

---

**Note** The `Index` field is returned as a positive integer if the input is a sensor object, such as a `radarDataGenerator` object. The `Index` field is returned as a negative integer if the input is an emitter object, such as a `radarEmitter` object.

---

## See Also

`coveragePlotter` | `plotCoverage`

**Introduced in R2021a**



# emissionsInBody

Transform emissions to platform body frame

## Syntax

```
EMBODY = emissionsInBody(EMSCENE, BODYFRAME)
```

## Description

EMBODY = emissionsInBody(EMSCENE, BODYFRAME) returns radar emissions converted to the body frame of the platform.

## Examples

### Convert Reflected Emission to Radar Body Frame

Convert a reflected radar emission back to radar body frame. Create a radar emitter.

```
emitter = radarEmitter(1);
```

Assume the radar is mounted on a platform located at [100 0 -10].

```
platTxRx = struct('PlatformID', 1, ...
    'Position', [100 0 -10], ...
    'Orientation', quaternion([0 0 0], 'eulerd', 'zyx', 'frame'));
```

Create a target.

```
platTgt = struct('PlatformID', 2, ...
    'Position', [20e3 0 -500], ...
    'Orientation', quaternion([45 0 0], 'eulerd', 'zyx', 'frame'), ...
    'Signatures', {rcsSignature});
```

Emit the signal. The emitted signal is in scenario frame.

```
simulationTime = 0;
emTx = step(emitter, platTxRx, simulationTime);
```

Reflect the emission off the target.

```
emProp = radarChannel(emTx, platTgt);
```

Convert the emission back to the body frame of the radar.

```
emRx = emissionsInBody(emProp, platTxRx)
```

```
emRx =
    radarEmission with properties:
```

```
    PlatformID: 1
    EmitterIndex: 1
    OriginPosition: [0 0 0]
```

```

OriginVelocity: [0 0 0]
Orientation: [1x1 quaternion]
FieldOfView: [1 5]
CenterFrequency: 300000000
Bandwidth: 3000000
WaveformType: 0
ProcessingGain: 0
PropagationRange: 0
PropagationRangeRate: 0
EIRP: 100
RCS: 0
    
```

## Input Arguments

### EMSCENE — Radar emission in scenario coordinates

radarEmission object

Emissions in scenario coordinates, specified as a cell array of radarEmission objects.

Data Types: cell

### BODYFRAME — Body frame of platform

body frame structure

Body frame of the platform, specified as a structure. The body frame structure must have the following fields.

Fieldname	Description	Default
Position	A 3-element vector specifying the position of the local reference frame's origin relative to its global frame in meters.	[0 0 0]
Velocity	A 3-element vector specifying the velocity of the local reference frame's origin relative to its global frame in meters per second.	[0 0 0]
Orientation	A scalar quaternion or a 3-by-3 real-valued orthonormal rotation matrix specifying the orientation of the local reference frame relative to its global frame.	eye(3)

Any structure that defines the fields above can be used to define a platform's body frame. For example, the structures returned by the platformPoses method on radarScenario object can be used.

Data Types: struct

## Output Arguments

### **EMBODY — Emissions in body coordinates**

radarEmission object

Emissions in body coordinates, returned as a cell array of radarEmission objects

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

radarEmission | radarEmitter | radarChannel

**Introduced in R2021a**

## quanttemp

Quantization temperature

### Syntax

```
qtemp = quanttemp(Ts,B)
qtemp = quanttemp(____,Name,Value)
[qtemp,qnf] = quanttemp(____)
```

### Description

`qtemp = quanttemp(Ts,B)` returns the quantization temperature in Kelvin based on the system temperature `Ts` and the number of bits `B`.

`qtemp = quanttemp(____,Name,Value)` returns the quantization temperature with additional options specified by one or more name-value pairs. For example, `'ReferenceTemperature',275` specifies a reference temperature of 275 K.

`[qtemp,qnf] = quanttemp(____)` also outputs the quantization noise figure `qnf` in decibels.

### Examples

#### Calculate Quantization Temperature for Radar

Calculate the quantization temperature for a radar with a system temperature of 1000 K and number of bits equal to 10.

```
Ts = 1000;
B = 10;
```

```
qtemp = quanttemp(Ts,B)
```

```
qtemp = 41.7656
```

#### Calculate Quantization Temperature with Specified Dynamic Range

Calculate the quantization temperature for a radar with a system temperature of 1000 K and number of bits equal to 10. Assume a dynamic range of 45 dB.

```
Ts = 1000;
B = 10;
```

```
qtemp = quanttemp(Ts,B,'DynamicRange',45)
```

```
qtemp = 20.1052
```

## Input Arguments

### Ts — System temperature

positive scalar

System temperature, specified as a positive scalar expressed in Kelvin.

Data Types: `double`

### B — Number of bits

vector

Number of bits, specified as a vector of positive integers. `B` and `DynamicRange` have the same length.

Data Types: `double`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `quanttemp(Ts,B,'DynamicRange',45)` specifies a dynamic range of 45 dB.

### DynamicRange — Dynamic range

vector

Dynamic range corresponding to the number of bits in `B`, specified as a vector expressed in decibels. `B` and `DynamicRange` have the same length.

### ReferenceTemperature — Reference temperature

290 (default) | positive scalar

Reference temperature, specified as a positive scalar expressed in Kelvin.

## Output Arguments

### qtemp — Quantization temperature

row vector

Quantization temperature in Kelvin, returned as a row vector.

### qnf — Quantization noise figure

row vector

Quantization noise figure in decibels, returned as a row vector.

## References

- [1] Richards, M. A. *Fundamentals of Radar Signal Processing*. Second edition. New York: McGraw-Hill Education, 2014.

[2] Barton, David K. *Radar Equations for Modern Radar*. Artech House Radar Series. Norwood, Mass: Artech House, 2013.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

systemp

**Introduced in R2021a**

# radarmetricplot

Plot radar performance metric against target range

## Syntax

```
radarmetricplot(range,metric)
radarmetricplot(range,metric,objective)
radarmetricplot(range,metric,objective,threshold)
radarmetricplot(___,Name,Value)
h = radarmetricplot(___)
```

## Description

`radarmetricplot(range,metric)` plots a radar performance metric `metric` as a function of the target range `range`. The input `range` is a length- $J$  vector of target ranges. The input `metric` is a  $J$ -by- $K$  matrix of the performance metric values for  $K$  radar systems computed at the target ranges in `range`.

`radarmetricplot(range,metric,objective)` also plots the objective requirement `objective` on the radar performance metric.

`radarmetricplot(range,metric,objective,threshold)` also plots the threshold requirement `threshold` on the radar performance metric.

`radarmetricplot(___,Name,Value)` specifies additional `Name,Value` arguments.

Example: `'MaxRangeRequirement',125e3,'MetricName','Available SNR'` specifies the maximum range requirement to be 125000 m, and the metric name to be 'Available SNR'

`h = radarmetricplot(___)` returns the handle to the axes in the figure.

## Examples

### Plot Available SNR and Detectability Factor

For a radar system, plot the available SNR and the detectability factor against the target range. Mark the required maximum range. Use the stoplight chart to assess the detection performance of the system at different ranges.

#### Scenario Parameters

Define the scenario parameters.

```
lambda = freq2wavelen(3e9);           % Wavelength (m)
Pt = 5e3;                             % Peak power (W)
tau = 1.2e-5;                         % Pulse width (s)
N = 24;                               % Number of received pulses
SwerlingCase = 'Swerling1';          % Swerling case
G = 40;                               % Antenna gain (dB)
Pfa = 1e-6;                          % Pfa
```

### Requirements

Specify the probability of detection to be 0.9 and the maximum range to be 125000 m.

```
Pd = 0.9; % Required Pd
MaxRangeRq = 125e3; % Maximum range requirement (m)
```

Specify the range points to evaluate the radar equation.

```
R = (1:1e2:200e3).';
```

### Compute Performance Metric and Requirement

Compute the available SNR and the detectability factor.

Compute the available SNR from the radar equation using the `radareqsnr` function.

```
SNRav = radareqsnr(lambda,R,Pt,tau,'Gain',G);
```

Compute the detectability factor using the `detectability` function.

```
DxObj = detectability(Pd,Pfa,N,SwerlingCase)
```

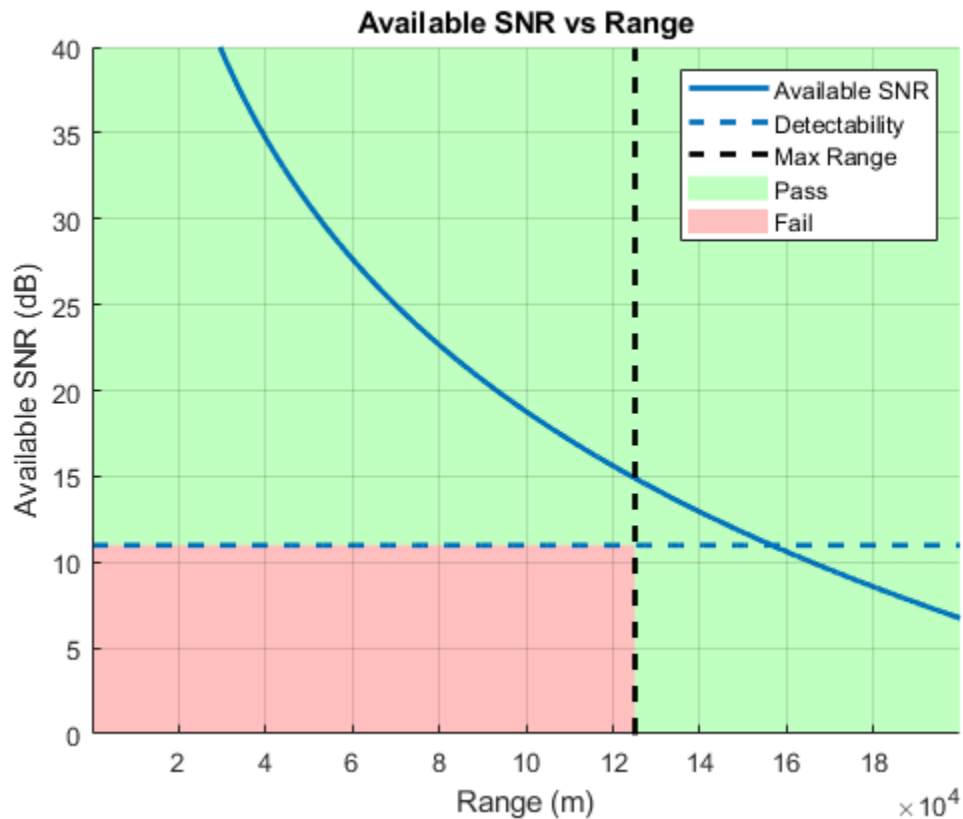
```
DxObj = 10.9850
```

### Plot Performance Metric and Requirement

Plot the available SNR in dB and the detectability factor against the target range using the `radarmetricplot` function. In order to plot, specify the `'MaxRangeRequirement'` to be 125000 m. Set `'ShowStoplight'` to true to show a stoplight chart that color codes the area of the plot according to the specified requirements.

```
radarmetricplot(R,SNRav,DxObj,'MaxRangeRequirement',MaxRangeRq, ...
    'MetricName','Available SNR','MetricUnit','dB',...
    'RequirementName','Detectability','ShowStoplight',true)
ylim([0 40])
```





## Input Arguments

### range — Target ranges

column vector

Target ranges at which the metric is computed, specified as a length- $J$  column vector, where  $J$  is the number of target ranges.

Data Types: double

### metric — Radar performance metric values

matrix

Radar performance metric values, specified as a  $J$ -by- $K$  matrix, where  $J$  is the length of the target range vector `range` and  $K$  is the number of radars.

Data Types: double

### objective — Objective requirement

scalar | vector | matrix

Objective requirement, specified as one of the following:

- scalar -- The objective requirement is assumed to be constant across all ranges in `range` and equal for all  $K$  radars.

- 1-by- $K$  vector -- The objective requirement is specified for each radar and is assumed to be constant for all ranges in `range`.
- $J$ -by-1 vector -- The objective requirement is specified for each range in `range` and is assumed to be equal for all  $K$  radars.
- $J$ -by- $K$  matrix -- The objective requirement is specified for each range in `range` and for each radar.

Data Types: `double`

### **threshold** — Threshold requirement

scalar | vector | matrix

Threshold requirement, specified as one of the following:

- scalar -- The threshold requirement is assumed to be constant across all ranges in `range` and equal for all  $K$  radars.
- 1-by- $K$  vector -- The threshold requirement is specified for each radar and is assumed to be constant for all ranges in `range`.
- $J$ -by-1 vector -- The threshold requirement is specified for each range in `range` and is assumed to be equal for all  $K$  radars.
- $J$ -by- $K$  matrix -- The threshold requirement is specified for each range in `range` and for each radar.

Data Types: `double`

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'MaxRangeRequirement',125e3,'MetricName','Available SNR'` specifies the maximum range requirement to be 125000 m, and the metric name to be `'Available SNR'`

### **MaxRangeRequirement** — Maximum range requirement

scalar | vector

Maximum range requirement, specified as one of the following:

- scalar -- Specifies the objective requirement on the maximum range.
- two-element vector -- Specifies both the objective and the threshold requirements in the `[objective threshold]` format.

Data Types: `double`

### **ShowStoplight** — Show stoplight chart

1 | 0

Specify whether to show the stoplight chart that color codes the area of the plot according to the specified requirements, specified as a logical scalar value.

If you only specify `objective`, the function divides the area of the plot into two colored zones along the metric axis. To satisfy the requirement, the function by default assumes that the metric must be

greater than or equal to the objective. In this case, the area above `objective` is marked `Pass` and is colored green, while the area below `objective` is marked `Fail` and is colored red.

To indicate the opposite case when the metric must be below the objective to satisfy the requirement, specify the `threshold` input explicitly as `Inf`. On the resultant stoplight chart, the objective requirement is satisfied at the ranges where the `metric` curve is in the `Pass` zone. At the ranges where the curve passes through the `Fail` zone, the system violates the objective requirement.

If you specify a finite `threshold`, the area between `objective` and `threshold` is colored yellow and marked `Warn`. At the ranges where the metric passes through the `Warn` zone, the objective requirement is violated, while the threshold requirement is still satisfied. The stoplight chart can be displayed only when the same requirements are specified for all radars (`objective` and `threshold` are scalars or length- $J$  column vectors). Otherwise, this name-value pair is ignored.

The value of the `'MaxRangeRequirement'` name-value pair limits the `Fail` and the `Warn` zones along the range axis. Both the `Fail` and the `Warn` zones extend to the objective value of the maximum range requirement when only the objective is provided. If both the objective and the threshold requirements are specified, the `Fail` zone extends to the threshold requirement while the `Warn` zone extends to the objective.

Data Types: `logical`

### **RadarName — Names of radar systems**

cell array of character vectors | string array

Names of the radar systems, specified as a length- $K$  cell array of character vectors or a string array, where  $K$  is the number of radars. The radar names are used to augment the corresponding legend entries. When not specified, the default name `'Radark'` is used for the  $k$ th radar system.

Data Types: `string` | `char` | `cell`

### **MetricName — Name of radar performance metric**

character vector | string scalar

Name of radar performance metric, specified as a character vector or a string scalar. When not specified, the default name `'Metric'` is used.

Data Types: `char` | `string`

### **RequirementName — Name of requirement**

character vector | string scalar

Name of requirement, specified as a character vector or a string scalar. When not specified, the function uses the default name `'Requirement'`.

Data Types: `char` | `string`

### **RangeUnit — Units for range values**

`'m'` (default) | `'km'` | `'mi'` | `'nmi'` | `'ft'`

Units for range values in vector `range` and for the value of `'MaxRangeRequirement'`, specified as one of the following:

- `'m'` -- Meters
- `'km'` -- Kilometers

- 'mi' -- Miles
- 'nmi' -- Nautical mile
- 'ft' -- Feet

**MetricUnit — Units for metric values**

' ' (default) | character vector | string scalar

Units for metric values, specified as a character vector or a string scalar.

Data Types: char | string

**Parent — Plot axes**

current axes (default) | Axes object

Handle to plot axes, specified as an Axes object. The default value is the current axes, which can be specified using `gca`.

**Output Arguments****h — Handle to axes in figure**

Axes object

Handle to the axes displayed in the figure, returned as an Axes object.

**More About****Stoplight Chart**

A radar system must meet a set of performance requirements that depend on the environment and scenarios in which the system is intended to operate. A number of such requirements can be fairly large and a design that satisfies all of them might be impractical. In this case a tradeoff analysis is applied. A subset of the requirements is satisfied at the expense of accepting lower values for the rest of the metrics. Such tradeoff analysis can be facilitated by specifying multiple requirement values for a single metric.

The requirement for each metric is specified as a pair of values:

- Objective — The desired level of the performance metric
- Threshold — The value of the metric below which the system's performance is considered unsatisfactory

The region between the Threshold and the Objective values is the trade-space. It defines a margin by which a metric can be below the Objective value while the system is still considered to have a satisfactory performance.

A stoplight chart color-codes the status of the performance metric for a radar system based on the specified requirements. The plot is divided into three zones:

- A Pass zone, colored green — At the ranges where the curve is in the Pass zone, the system performance satisfies the Objective value of the requirement.
- A Warn zone, colored yellow — At the ranges where the curve passes through the Warn zone, the system performance violates the Objective value of the specified requirement but still satisfies the Threshold value.

- A Fail zone, colored red — At the ranges where the curve passes through the Fail zone, the system performance violates the Threshold value of the specified requirement.

## References

- [1] Charles S. Wasson. *System engineering analysis, design, and development: Concepts, principles, and practices*. John Wiley & Sons, 2015.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

radareqsnr | detectability | rocinterp

### Apps

Radar Designer

**Introduced in R2021a**

## arrayscanloss

Loss due to electronic scanning off broadside

### Syntax

```
LSS = arrayscanloss(PD,PFA,N)
LSS = arrayscanloss( ____,THETAM)
LSS = arrayscanloss( ____,SW)
LSS = arrayscanloss( ____, 'CosinePower',COSINEPOWER)
```

### Description

`LSS = arrayscanloss(PD,PFA,N)` returns the two-way statistical scan sector loss for a radar with a phased array antenna that electronically scans a sector from  $-60$  to  $+60$  degrees off broadside. The computation assumes a square-law detector and a nonfluctuating target.

`LSS = arrayscanloss( ____,THETAM)` computes the scan sector loss given the scan sector limits specified about the broadside direction.

`LSS = arrayscanloss( ____,SW)` computes the scan sector loss for radar echoes received from a chi-squared distributed target specified by the Swerling case.

`LSS = arrayscanloss( ____, 'CosinePower',COSINEPOWER)` specifies the exponent of the cosine modeling the gain loss of an array scanned off broadside. This exponent takes into account two effects that result in the gain reduction due to array scanning. The first effect is the beam broadening due to the reduced projected array area in the beam direction. The second effect is a reduction of the effective aperture area of the individual array elements at off-broadside angles.

### Examples

#### Plot Statistical Scan Sector Loss Phased Array Radar Antenna

Compute the statistical scan sector loss for a radar with a phased array antenna. The array scans from  $-45$  to  $70$  degrees about the broadside direction. Assume a single pulse is received from a Swerling 1 case target by a square-law detector and the probability of false alarm is set to  $1e-6$ . Plot the computed loss as a function of the desired probability of detection.

```
Pd = 0.1:0.01:0.99;           % Detection probabilities
Pfa = 1e-6;                   % Probability of false alarm
N = 1;                         % Number of received pulses
ThetaM = [-45 70];           % Scan sector limits
Lss = arrayscanloss(Pd,Pfa,N,ThetaM, 'Swerling1');
```

Plot the statistical scan sector loss.

```
plot(Pd,Lss)
xlabel('Probability of Detection')
ylabel('Loss (dB)')
title('Scan Sector Loss vs P_d for Swerling 1 Target')
grid on
```



## Input Arguments

### PD — Desired probability of detection

scalar |  $J$ -length vector

Desired probability of detection, specified as a scalar or  $J$ -length vector between 0.1 and 0.999999.

Data Types: double

### PFA — Probability of false alarm

scalar |  $K$ -length vector

Probability of false alarm, specified as a scalar or  $K$ -length vector between  $1e-15$  and  $1e-3$ .

Data Types: double

### N — Number of received pulses

positive scalar

Number of received pulses, specified as a positive scalar.

Data Types: double

### THETAM — Scan sector limits

$[-60 \ 60]$  (default) | scalar | two-element vector

Scan sector limits, specified as a scalar or two-element vector. If THETAM is a scalar, then the scan sector spans from -THETAM to +THETAM. If THETAM is a two-element vector of the form [theta1 theta2], the scan sector spans from theta1 to theta2. The default value is [-60 60]. Units are in degrees.

Data Types: double

**SW — Scan sector limits**

'Swerling0' (default) | 'Swerling1' | 'Swerling2' | 'Swerling3' | 'Swerling4' | 'Swerling5'

Scan sector limits, specified as the Swerling case for the chi-squared distributed target. The default value of SW is 'Swerling0'.

**COSINEPOWER — Gain loss cosine exponent**

2.5 (default) | positive scalar

Exponent of the cosine modeling the gain loss of an array scanned off broadside, specified as a positive scalar. Typically, the exponent value lies between 2 and 3. The default value is 2.5.

Data Types: double

## Output Arguments

**LSS — Two-way statistical scan loss**

*J*-by-*K* matrix

Two-way statistical scan sector loss, returned as a *J*-by-*K* matrix, where *J* and *K* are the dimensions of the PD and PFA arguments. Units are in decibels (dB).

## Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

beamloss | beamdwellfactor | detectability

**Introduced in R2021a**



# beamdwellfactor

Range-dependent loss for rapidly scanning beam

## Syntax

```
fdb = beamdwellfactor(r,hpbw,scanrate)
```

## Description

`fdb = beamdwellfactor(r,hpbw,scanrate)` calculates the range-dependent beam-dwell factor on page 1-223 `fdb` for an antenna at the specified range `r`, half-power beamwidth `hpbw`, and scan rate `scanrate`. The `beamdwellfactor` function assumes that the transmitter and receiver antennas have equal beamwidth and an ideal Gaussian antenna pattern with no side lobes.

## Examples

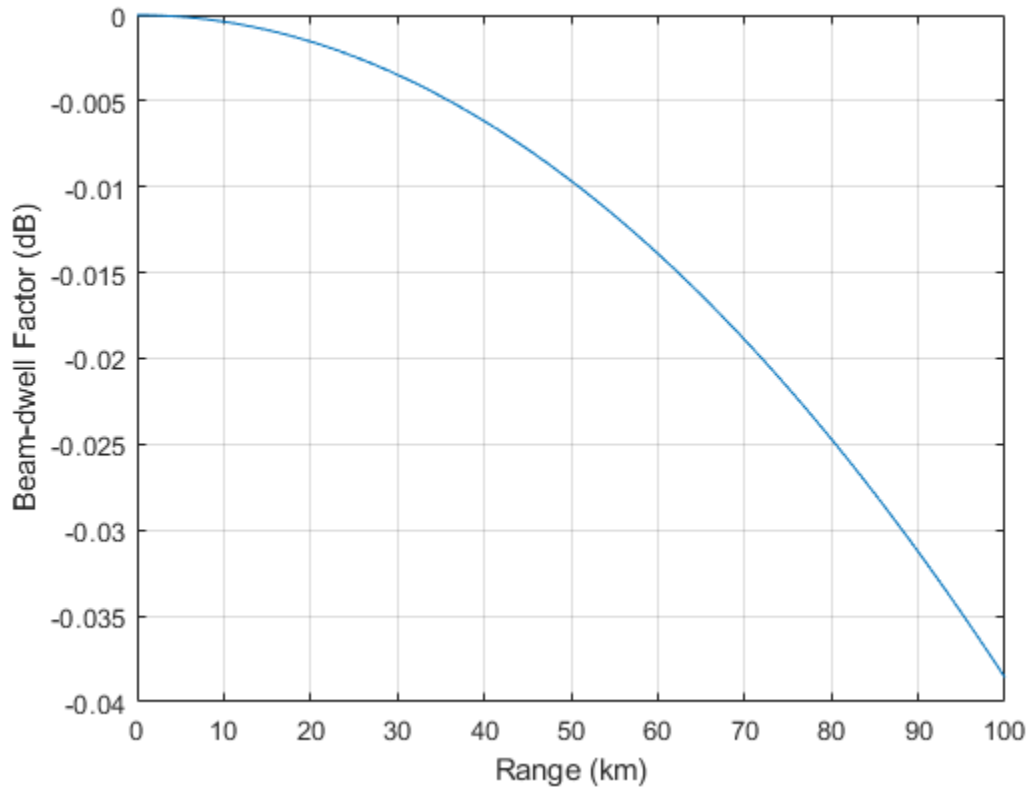
### Calculate Beam-Dwell Factor

Calculate the beam-dwell factor for a surveillance radar at 100 linearly-spaced ranges in the interval `[0,100000]` meters. Specify the beamwidth as 1 degree and the scan rate as 120 degrees per second.

```
r = linspace(0,100000);  
hpbw = 1;  
scanrate = 120;  
fdb = beamdwellfactor(r,hpbw,scanrate);
```

Plot the beam-dwell factor as a function of range. Before plotting, convert the range from meters to kilometers.

```
plot(r*0.001,fdb)  
grid on  
xlabel('Range (km)')  
ylabel('Beam-dwell Factor (dB)')
```



## Input Arguments

### **r — Range**

scalar | vector

Range in meters, specified as a scalar or vector.

Data Types: double

### **hpbw — Half-power beamwidth**

scalar | vector

Half-power beamwidth of the antenna in degrees, specified as a scalar or vector. If `hpbw` is a vector, then `scanrate` must be a scalar or a vector of the same size.

Data Types: double

### **scanrate — Scan rate**

scalar | vector

Scan rate of the antenna in degrees per second. If `scanrate` is a vector, then `hpbw` must be a scalar or a vector of the same size.

Data Types: double

## Output Arguments

### fbd — Range-dependent beam-dwell factor

matrix

Range-dependent beam-dwell factor in dB, returned as a  $j$ -by- $k$  matrix such that  $j$  is the size of `r` and  $k$  is the size of `hpbw` or `scanrate`, whichever is larger.

The rows of `fbd` correspond to the ranges in `r`. The columns depend on the sizes of `hpbw` and `scanrate`.

- If `hpbw` is a vector and `scanrate` is a scalar, then the columns of `fbd` correspond to the half-power beamwidths in `hpbw`.
- If `hpbw` is a scalar and `scanrate` is a vector, then the columns of `fbd` correspond to the scan rates in `scanrate`.
- If `hpbw` and `scanrate` are both vectors, then the columns of `fbd` correspond to both the half-power beamwidths in `hpbw` and the scan rates in `scanrate`.

Data Types: `double`

## More About

### Beam-Dwell Factor

The beam-dwell factor accounts for the misalignment between transmitter and receiver beam axes when a scanning system has a high scan rate and long-range targets.

The equation for the beam-dwell factor,  $F_{bd}$ , is

$$F_{bd} = L \int_{-\pi}^{\pi} f^2(\theta) f^2(\theta - \delta) d\theta$$

where the terms in the equation are:

- $L$  — Normalizing factor that brings  $F_{bd}$  to unity for  $\delta = 0$
- $\delta = t_d / t_0$  — Fractional beamwidth scanned during the delay, where:
  - $t_d = 2R / c$  — Time delay for a target, where  $R$  is the range and  $c$  is the wave propagation speed
  - $t_0 = \theta_3 / \omega_s$  — The time the system takes to continuously scan through one beamwidth, where  $\theta_3$  is the half-power beamwidth and  $\omega_s$  is the scan rate
- $f(\theta)$  — Antenna pattern

The `beamdwelfactor` function assumes an ideal Gaussian antenna pattern with no side lobes. The equation for the ideal Gaussian antenna pattern with no side lobes,  $f(\theta)$ , is:

$$f(\theta) = \exp\left[-2(\ln 2) \frac{\theta^2}{\theta_3^2}\right]$$

## References

- [1] Barton, David Knox. "Beam-Dwell Factor  $F_{bd}$ ." In *Radar Equations for Modern Radar*, 362. Artech House Radar Series. Boston, Mass: Artech House, 2013.
- [2] Barton, David Knox. "Antenna Patterns." In *Radar Equations for Modern Radar*, 147. Artech House Radar Series. Boston, Mass: Artech House, 2013.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`beamloss` | `arrayscanloss` | `detectability`

**Introduced in R2021a**

# beamloss

Beam shape loss for Gaussian antenna pattern

## Syntax

```
lb = beamloss
lb = beamloss(is2d)
```

## Description

`lb = beamloss` calculates the beam shape loss on page 1-226 `lb` for a radar that scans over one angular dimension (1-D). The `beamloss` function assumes the antenna has a Gaussian pattern and densely samples the angular domain. For the angular domain to be densely sampled, beam dwells must be spaced by less than 0.71 of the one-way half-power beamwidth.

You can use `lb` as an accurate approximation of loss for antenna patterns other than Gaussian patterns.

`lb = beamloss(is2d)`, where `is2d` is 1 (`true`), calculates the beam shape loss for a scanning radar over two angular dimensions (2-D). The default for `is2d` is 0 (`false`), which calculates the beam shape loss for a scanning radar in one angular dimension.

## Examples

### Calculate Power-Aperture Product with Beam Shape Loss

Calculate the power-aperture product for a search radar performing a two-dimensional search by using the `radareqsearchpap` function. Include beam shape loss by using the `beamloss` function.

Specify a search volume of  $0.2\pi$  steradians and a search time of 4 seconds. The radar requires a signal-to-noise ratio (SNR) of 20 decibels to detect a 1 square meter radar cross-section (RCS) target at a range of 100000 meters. By default, the system noise temperature is 290 kelvin.

```
omega = 0.2*pi;
tsearch = 4;
snr = 20;
range = 100000;
```

Calculate the power-aperture product, including the beam shape loss. Assume the rest of the losses for the system are 0 decibels.

```
lb = beamloss;
pap = radareqsearchpap(range, snr, omega, tsearch, 'Loss', lb)

pap = 105.0012
```

## Input Arguments

### is2d — Scanning in two angular dimensions

false or 0 (default) | true or 1

Scanning in two angular dimensions, specified as numeric or logical 1 (true) or 0 (false). When you do not specify is2d, or specify is2d as 0 (false), the function assumes the radar scans in one angular dimension.

Data Types: logical

## Output Arguments

### lb — Beam shape loss

scalar

Beam shape loss in decibels, returned as a scalar.

Data Types: double

## More About

### Beam-Shape Loss

Incorporate beam shape loss into the standard form of the radar range equation implemented by the `radareqsearchsnr`, `radareqsearchrng`, and `radareqsearchpap` functions to account for the use of peak gain instead of effective gain. The effective gain results from the two-way pattern of the scanning antenna modulating the received train of pulses.

The power equation for 1-D beam shape loss for an antenna with a Gaussian pattern,  $L_{p1}$ , is

$$L_{p1} = \sqrt{\frac{8\ln 2}{\pi}} = 1.3288$$

The power equation for 2-D beam shape loss for an antenna with a Gaussian pattern,  $L_{p2}$ , is

$$L_{p1} = \frac{8\ln 2}{\pi} = 1.7658$$

In decibels, the 1-D beam shape loss is 1.2338 and the 2-D beam shape loss is 2.4677.

You can use beam shape loss for an antenna with a Gaussian pattern as an accurate approximation of loss for antennas with other patterns.

## References

- [1] Barton, David Knox. "Beamshape Loss for Different Patterns." In *Radar Equations for Modern Radar*, 148-149. Artech House Radar Series. Boston, Mass: Artech House, 2013.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

[beamwellfactor](#) | [arrayscanloss](#) | [detectability](#)

**Introduced in R2021a**

## radareqsearchpap

Power-aperture product using search radar equation

### Syntax

```
pap = radareqsearchpap(range,snr,omega,tsearch)
pap = radareqsearchpap( ___,Name,Value)
```

### Description

`pap = radareqsearchpap(range,snr,omega,tsearch)` computes the available power-aperture product, `pap`, for a surveillance radar based on the range, `range`, required signal-to-noise ratio (SNR), `snr`, solid angular search volume, `omega`, and search time, `tsearch`.

`pap = radareqsearchpap( ___,Name,Value)` computes the available power-aperture product with additional options specified by one or more name-value arguments. For example, `'Loss',6` specifies system losses as 6 decibels.

### Examples

#### Compute Power-Aperture Product Using Search Radar Equation

Compute the power-aperture product for a search radar that is required to detect a 1 square meter RCS target at a range of 111 kilometers. Assume the antenna rotates at a rate of 12.5 RPM, the signal-to-noise ratio required to make a detection is 13 decibels, the system noise temperature is 487 Kelvin, and the total system loss is 20 decibels.

```
range = 111e3;
tsearch = 60 / 12.5;
snr = 13;
ts = 487;
loss = 20;
```

The radar traverses a search volume with azimuths in the range [-180,180] degrees and elevations in the range [0,45] degrees. Find the solid angular search volume in steradians by using the `solidangle` function.

```
az = [-180;180];
el = [0;45];
omega = solidangle(az,el);
```

Calculate the power-aperture product. By default, the target RCS is 1 square meter.

```
snr = radareqsearchpap(range,snr,omega,tsearch,'Ts',ts,'Loss',loss)
snr = 2.3689e+04
```



### Plot Power-Aperture Product as Function of Required SNR

Plot the power-aperture product as a function of the required SNR for a search radar system located at a range of 100 kilometers. Incorporate path loss due to absorption into the calculation of the power-aperture product.

Specify the required SNR as values in the range [-5,25] decibels. Assume the search volume is 1.5 steradians and the search time is 12 seconds.

```
range = 100e3;
snr = -5:25;
omega = 1.5;
tsearch = 12;
```

Find the path loss due to atmospheric gaseous absorption by using the `gaspl` function. Specify the radar operating frequency as 10 GHz, the temperature as 15 degrees Celsius, the dry air pressure as 1013 hPa, and the water vapour density as 7.5 g/m<sup>3</sup>.

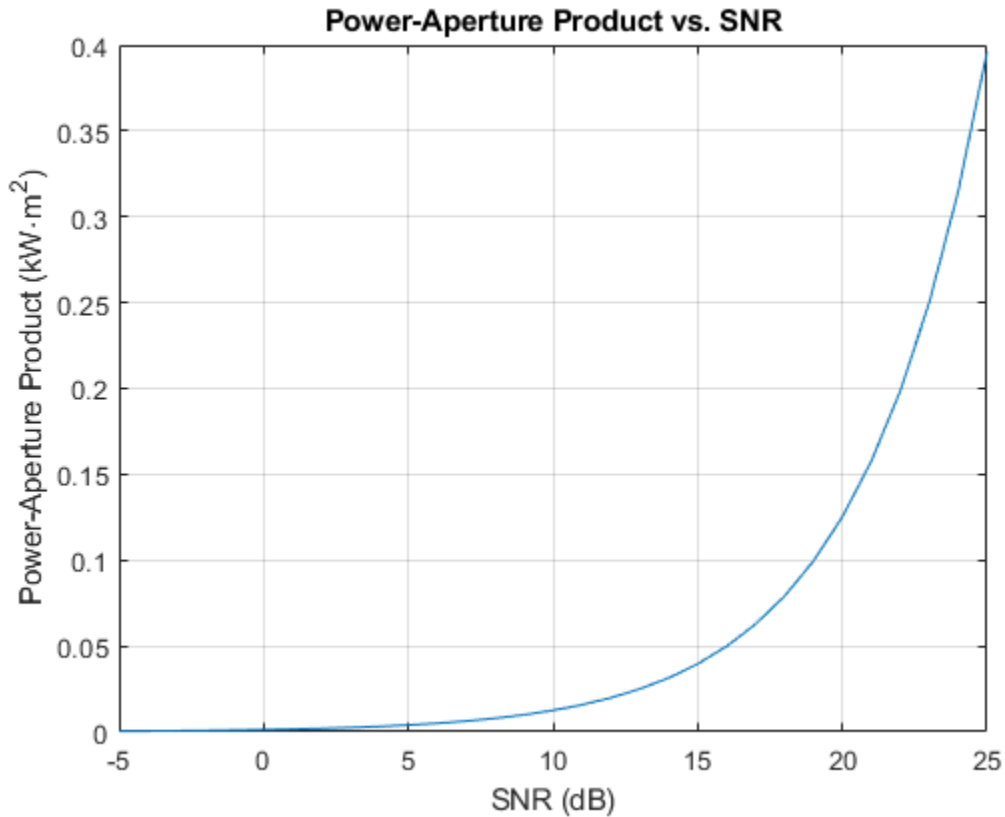
```
freq = 10e9;
temp = 15;
pressure = 1013e2;
density = 7.5;
loss = gaspl(range, freq, temp, pressure, density);
```

Compute the power-aperture product. By default, the target RCS is 1 square meter.

```
pap = radareqsearchpap(range, snr, omega, tsearch, 'AtmosphericLoss', loss);
```

Plot the power-aperture product as a function of the required SNR. Before plotting, convert the power-aperture product from  $W \cdot m^2$  to  $kW \cdot m^2$ .

```
plot(snr, pap*0.001)
grid on
xlabel('SNR (dB)')
ylabel('Power-Aperture Product (kW\cdotm^2)')
title('Power-Aperture Product vs. SNR')
```



## Input Arguments

### **range** — Range

scalar | length- $J$  vector of positive values

Range, specified as a scalar or a length- $J$  vector of positive values, where  $J$  is the number of range samples. Units are in meters.

Example: `1e5`

Data Types: `double`

### **snr** — Required signal-to-noise ratio

scalar | length- $J$  vector of real values

Required signal-to-noise ratio (SNR), specified as a scalar or a length- $J$  vector of real values. Units are in decibels.

Example: `13`

Data Types: `double`

### **omega** — Solid angular search volume

scalar

Solid angular search volume, specified as a scalar. Units are in steradians.

Given the elevation and azimuth ranges of a region, you can find the solid angular search volume by using the `solidangle` function.

Example: 0.3702

Data Types: double

### **tsearch — Search time**

scalar

Search time, specified as a scalar. Units are in seconds.

Example: 10

Data Types: double

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'Ts', 487 specifies the system noise temperature as 487 Kelvin

### **RCS — Radar cross section**

1 (default) | positive scalar | length-*J* vector of positive values

Radar cross section of the target, specified as a positive scalar or length-*J* vector of positive values. The `radareqsearchpap` function assumes the target RCS is nonfluctuating (Swerling case 0). Units are in square meters.

Data Types: double

### **Ts — System noise temperature**

290 (default) | positive scalar

System noise temperature, specified as a positive scalar. Units are in Kelvin.

Data Types: double

### **Loss — System losses**

0 (default) | scalar | length-*J* vector of real values

System losses, specified as a scalar or a length-*J* vector of real values. Units are in decibels.

Example: 1

Data Types: double

### **AtmosphericLoss — One-way atmospheric absorption loss**

0 (default) | scalar | length-*J* vector of real values

One-way atmospheric absorption loss, specified as a scalar or a length-*J* vector of real values. Units are in decibels.

Example: [10, 20]

Data Types: double

**PropagationFactor — One-way propagation factor**0 (default) | scalar | length- $J$  vector of real values

One-way propagation factor for the transmit and receive paths, specified as a scalar or a length- $J$  vector of real values. Units are in decibels.

Example: [10, 20]

Data Types: double

**CustomFactor — Custom loss factors**0 (default) | scalar | length- $J$  vector of real values

Custom loss factors, specified as a scalar or a length- $J$  vector of real values. These factors contribute to the reduction of the received signal energy and can include range-dependent sensitivity time control (STC), eclipsing, and beam-dwell factors. Units are in decibels.

Example: [10, 20]

Data Types: double

**Output Arguments****pap — Power-aperture product**scalar | length- $J$  column vector of positive values

Power-aperture product, returned as a scalar or a length- $J$  column vector of positive values, where  $J$  is the number of range samples. Units are in  $\text{W}\cdot\text{m}^2$ .

Data Types: double

**More About****Power-Aperture Product Form of Search Radar Equation**

The power-aperture product form of the search radar equation,  $P_{av}A$ , is:

$$P_{av}A = \frac{4\pi\Omega R^4 k T_s (SNR) L_a^2 L}{t_s \sigma F^2 F_c}$$

where the terms of the equation are:

- $\Omega$  — Search volume in steradians
- $R$  — Target range in meters. The equation assumes the radar is monostatic
- $k$  — Boltzmann constant
- $T_s$  — System temperature in Kelvin
- $SNR$  — Required signal-to-noise ratio
- $L_a$  — One-way atmospheric absorption loss
- $L$  — Combined system losses
- $t_s$  — Search time in seconds
- $\sigma$  — Nonfluctuating target radar cross section in square meters

- $F$  — One-way propagation factor for the transmit and receive paths
- $F_c$  — Combined range-dependent factors that contribute to the reduction of the received signal energy

You can derive this equation by rearranging the SNR form of the search radar equation. See the `radareqsearchsnr` function for more information.

## References

- [1] Barton, David Knox. *Radar Equations for Modern Radar*. Artech House Radar Series. Boston, Mass: Artech House, 2013.
- [2] Skolnik, Merrill I. *Introduction to Radar Systems*. Third edition. McGraw-Hill Electrical Engineering Series. Boston, Mass. Burr Ridge, IL Dubuque, IA: McGraw Hill, 2001.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`radareqsearchrng` | `radareqsearchsnr` | `radareqsnr` | `radareqrng` | `radareqpow`

**Introduced in R2021a**

## radareqsearchrng

Maximum detectable range using search radar equation

### Syntax

```
range = radareqsearchrng(snr,pap,omega,tsearch)
range = radareqsearchrng( ____,Name,Value)
```

### Description

`range = radareqsearchrng(snr,pap,omega,tsearch)` computes the maximum detectable range, `range`, for a surveillance radar based on the required signal-to-noise ratio (SNR), `snr`, power-aperture product, `pap`, solid angular search volume, `omega`, and search time, `tsearch`.

`range = radareqsearchrng( ____,Name,Value)` computes the maximum detectable range with additional options specified by one or more name-value arguments. For example, `'Loss',6` specifies system losses as 6 decibels.

### Examples

#### Compute Maximum Detectable Range Using Search Radar Equation

Compute the maximum detectable range at which a surveillance radar can detect a target.

The radar operates at a frequency of 2.5 GHz and transmits an average power of 2.1 kW. The gain of the receiving antenna is 34 decibels. Calculate the power-aperture product using these values.

```
lambda = freq2wavelen(2.5e9);
pav = 2100;
g = 34;
a = gain2aperture(g,lambda);
pap = pav*a;
```

The radar traverses a search volume with azimuths in the range [-180,180] degrees and elevations in the range [0,40] degrees. Find the solid angular search volume in steradians by using the `solidangle` function.

```
az = [-180;180];
el = [0;40];
omega = solidangle(az,el);
```

The antenna rotates at a rate of 12.5 RPM. Assume the system noise temperature is 487 Kelvin, the total system loss is 20 decibels, and the minimum SNR required to make a detection is 13 decibels.

```
tsearch = 60 / 12.5;
ts = 487;
loss = 20;
snr = 13;
```

Compute the maximum detectable range. By default, the target RCS is 1 square meter.

```
R = radareqsearchrng(snr,pap,omega,tsearch,...
    'Ts',ts,'Loss',loss,'unitstr','km')
```

```
R = 80.7673
```

## Input Arguments

### **snr** — Required signal-to-noise ratio

scalar | length-*J* vector of real values

Required signal-to-noise ratio (SNR), specified as a scalar or a length-*J* vector of real values. Units are in decibels.

Example: 13

Data Types: double

### **pap** — Power-aperture product

scalar | length-*J* vector of positive values

Power-aperture product, specified as a scalar or a length-*J* vector of positive values. Units are in  $W \cdot m^2$ .

Example: 3e6

Data Types: double

### **omega** — Solid angular search volume

scalar

Solid angular search volume, specified as a scalar. Units are in steradians.

Given the elevation and azimuth ranges of a region, you can find the solid angular search volume by using the `solidangle` function.

Example: 0.3702

Data Types: double

### **tsearch** — Search time

scalar

Search time, specified as a scalar. Units are in seconds.

Example: 10

Data Types: double

## Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'Ts',487` specifies the system noise temperature as 487 Kelvin

**RCS — Radar cross section**

1 (default) | positive scalar | length-*J* vector of positive values

Radar cross section of the target, specified as a positive scalar or length-*J* vector of positive values. The `radareqsearchrng` function assumes the target RCS is nonfluctuating (Swerling case 0). Units are in square meters.

Data Types: `double`

**Ts — System noise temperature**

290 (default) | positive scalar

System noise temperature, specified as a positive scalar. Units are in Kelvin.

Data Types: `double`

**Loss — System losses**

0 (default) | scalar | length-*J* vector of real values

System losses, specified as a scalar or a length-*J* vector of real values. Units are in decibels.

Example: 1

Data Types: `double`

**CustomFactor — Custom loss factors**

0 (default) | scalar | length-*J* vector of real values

Custom loss factors, specified as a scalar or a length-*J* vector of real values. These factors contribute to the reduction of the received signal energy. Units are in decibels.

Example: [10,20]

Data Types: `double`

**unitstr — Range units**

'm' (default) | 'km' | 'mi' | 'nmi'

Range units, specified as one of the following values:

- 'm' — Return range using meters
- 'km' — Return range using kilometers
- 'mi' — Return range using statute miles
- 'nmi' — Return range using nautical miles (US)

If you do not specify range units, then the `radareqsearchrng` function returns ranges using meters.

Data Types: `string` | `char`

**Output Arguments****range — Maximum detectable range**

scalar | length-*J* column vector of positive values

Maximum detectable range, returned as a scalar or a length-*J* column vector of positive values. Units are in meters.



## More About

### Maximum Detectable Range Form of Search Radar Equation

The maximum detectable range form of the search radar equation,  $R$ , is:

$$R = \left[ \frac{P_{av} A t_s \sigma F^2 F_c}{4\pi k T_s (SNR) L_a^2 L \Omega} \right]^{1/4}$$

where the terms of the equation are:

- $P_{av}$  — Average transmit power in watts
- $A$  — Antenna effective aperture in square meters
- $t_s$  — Search time in seconds
- $\sigma$  — Nonfluctuating target radar cross section in square meters
- $F$  — One-way propagation factor for the transmit and receive paths
- $F_c$  — Combined range-dependent factors that contribute to the reduction of the received signal energy
- $k$  — Boltzmann constant
- $T_s$  — System temperature in Kelvin
- $SNR$  — Required signal-to-noise ratio
- $L_a$  — One-way atmospheric absorption loss
- $L$  — Combined system losses
- $\Omega$  — Search volume in steradians

You can derive this equation by rearranging the SNR form of the search radar equation. See the `radareqsearchsnr` function for more information.

## References

- [1] Barton, David Knox. *Radar Equations for Modern Radar*. Artech House Radar Series. Boston, Mass: Artech House, 2013.
- [2] Skolnik, Merrill I. *Introduction to Radar Systems*. Third edition. McGraw-Hill Electrical Engineering Series. Boston, Mass. Burr Ridge, IL Dubuque, IA: McGraw Hill, 2001.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`radareqsearchpap` | `radareqsearchsnr` | `radareqsnr` | `radareqrng` | `radareqpow` | `gain2aperture`

**Introduced in R2021a**

# radareqsearchsnr

Range-dependent SNR using search radar equation

## Syntax

```
snr = radareqsearchsnr(range,pap,omega,tsearch)
snr = radareqsearchsnr( ___,Name,Value)
```

## Description

`snr = radareqsearchsnr(range,pap,omega,tsearch)` computes the available signal-to-noise ratio (SNR), `snr`, for a surveillance radar based on the range, `range`, power-aperture product, `pap`, solid angular search volume, `omega`, and search time, `tsearch`.

`snr = radareqsearchsnr( ___,Name,Value)` computes the available SNR with additional options specified by one or more name-value arguments. For example, `'Loss',6` specifies system losses as 6 decibels.

## Examples

### Compute SNR Using Search Radar Equation

Compute the available signal-to-noise ratio (SNR) for a search radar at a target range of 1000 kilometers with a power-aperture product of  $3 \times 10^6 \text{ W} \cdot \text{m}^2$ . Assume the search time is 10 seconds, the RCS of the target is  $-10 \text{ dBsm}$ , the system noise temperature is 487 Kelvin, and the total system loss is 6 decibels.

```
range = 1000e3;
pap = 3e6;
tsearch = 10;
rcs = db2pow(-10);
ts = 487;
loss = 6;
```

The radar surveys a region of space with azimuths in the range [0,30] degrees and elevations in the range [0,45] degrees. Find the solid angular search volume in steradians by using the `solidangle` function.

```
az = [0;30];
el = [0;45];
omega = solidangle(az,el);
```

Calculate the available SNR.

```
snr = radareqsearchsnr(range,pap,omega,tsearch,'RCS',rcs,'Ts',ts,'Loss',loss)
snr = 13.8182
```

### Plot SNR as Function of Range

Plot the available signal-to-noise ratio (SNR) as a function of the range for a search radar with a power-aperture product of  $2.5 \times 10^6 \text{ W} \cdot \text{m}^2$ . Incorporate path loss due to absorption into the calculation of the SNR.

Specify the ranges as 1000 linearly-spaced values in the interval [0,1000] kilometers. Assume the search volume is 1.5 steradians and the search time is 12 seconds.

```
range = linspace(1,1000e3,1000);  
pap = 2.5e6;  
omega = 1.5;  
tsearch = 12;
```

Find the path loss due to atmospheric gaseous absorption by using the `gaspl` function. Specify the radar operating frequency as 10 GHz, the temperature as 15 degrees Celsius, the dry air pressure as 1013 hPa, and the water vapour density as  $7.5 \text{ g/m}^3$ .

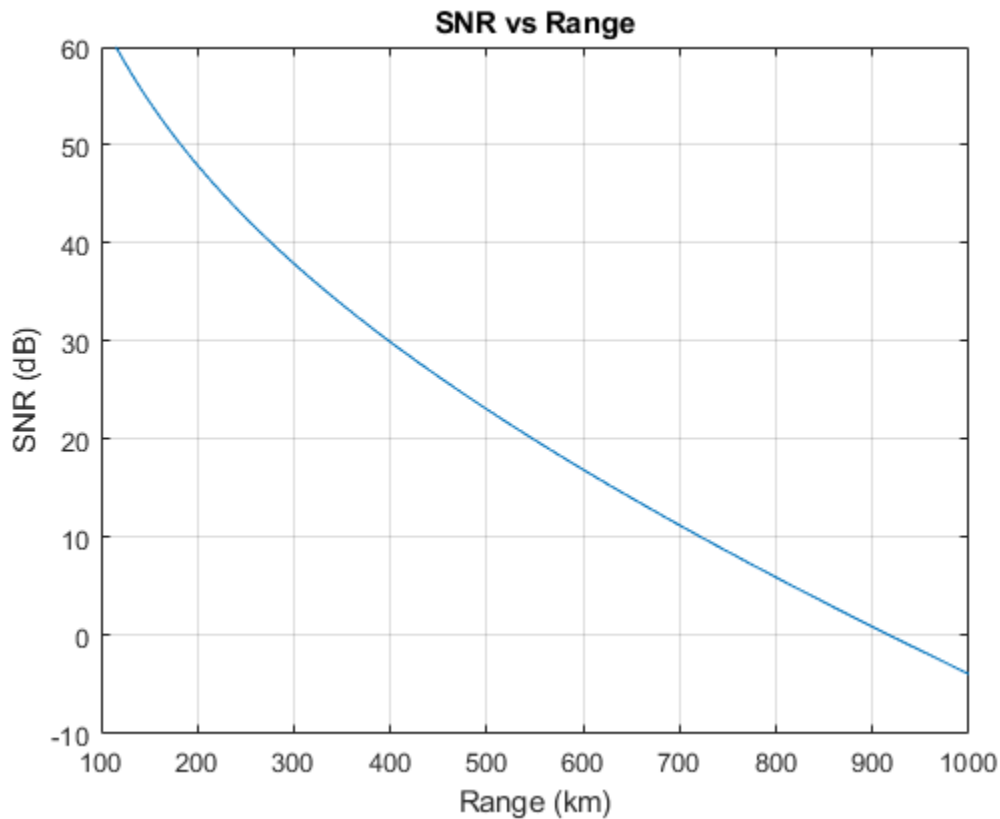
```
freq = 10e9;  
temp = 15;  
pressure = 1013e2;  
density = 7.5;  
loss = gaspl(range,freq,temp,pressure,density);
```

Compute the available SNR. By default, the target RCS is 1 square meter.

```
snr = radareqsearchsnr(range,pap,omega,tsearch,'AtmosphericLoss',loss);
```

Plot the SNR as a function of the range. Before plotting, convert the range from meters to kilometers.

```
plot(range*0.001,snr)  
grid on  
ylim([-10 60])  
xlabel('Range (km)')  
ylabel('SNR (dB)')  
title('SNR vs Range')
```



## Input Arguments

### **range** — Range

scalar | length- $J$  vector of positive values

Range, specified as a scalar or a length- $J$  vector of positive values, where  $J$  is the number of range samples. Units are in meters.

Example: `1e5`

Data Types: `double`

### **pap** — Power-aperture product

scalar | length- $J$  vector of positive values

Power-aperture product, specified as a scalar or a length- $J$  vector of positive values. Units are in  $\text{W}\cdot\text{m}^2$ .

Example: `3e6`

Data Types: `double`

### **omega** — Solid angular search volume

scalar

Solid angular search volume, specified as a scalar. Units are in steradians.

Given the elevation and azimuth ranges of a region, you can find the solid angular search volume by using the `solidangle` function.

Example: 0.3702

Data Types: `double`

### **tsearch — Search time**

scalar

Search time, specified as a scalar. Units are in seconds.

Example: 10

Data Types: `double`

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'Ts', 487` specifies the system noise temperature as 487 Kelvin

### **RCS — Radar cross section**

1 (default) | positive scalar | length-*J* vector of positive values

Radar cross section of the target, specified as a positive scalar or length-*J* vector of positive values. The `radareqsearchsnr` function assumes the target RCS is nonfluctuating (Swerling case 0). Units are in square meters.

Data Types: `double`

### **Ts — System noise temperature**

290 (default) | positive scalar

System noise temperature, specified as a positive scalar. Units are in Kelvin.

Data Types: `double`

### **Loss — System losses**

0 (default) | scalar | length-*J* vector of real values

System losses, specified as a scalar or a length-*J* vector of real values. Units are in decibels.

Example: 1

Data Types: `double`

### **AtmosphericLoss — One-way atmospheric absorption loss**

0 (default) | scalar | length-*J* vector of real values

One-way atmospheric absorption loss, specified as a scalar or a length-*J* vector of real values. Units are in decibels.

Example: [10, 20]

Data Types: `double`

**PropagationFactor — One-way propagation factor**0 (default) | scalar | length- $J$  vector of real values

One-way propagation factor for the transmit and receive paths, specified as a scalar or a length- $J$  vector of real values. Units are in decibels.

Example: [10, 20]

Data Types: double

**CustomFactor — Custom loss factors**0 (default) | scalar | length- $J$  vector of real values

Custom loss factors, specified as a scalar or a length- $J$  vector of real values. These factors contribute to the reduction of the received signal energy and can include range-dependent sensitivity time control (STC), eclipsing, and beam-dwell factors. Units are in decibels.

Example: [10, 20]

Data Types: double

**Output Arguments****snr — Available signal-to-noise ratio**scalar | length- $J$  column vector of real values

Available signal-to-noise ratio, returned as a scalar or a length- $J$  column vector of real values, where  $J$  is the number of range samples. Units are in decibels.

**More About****SNR Form of Search Radar Equation**

The signal-to-noise ratio form of the search radar equation,  $SNR$ , is:

$$SNR = \frac{P_{av} A t_s \sigma F^2 F_c}{4\pi k T_s R^4 L_a^2 L \Omega}$$

where the terms of the equation are:

- $P_{av}$  — Average transmit power in watts
- $A$  — Antenna effective aperture in square meters
- $t_s$  — Search time in seconds
- $\sigma$  — Nonfluctuating target radar cross section in square meters
- $F$  — One-way propagation factor for the transmit and receive paths
- $F_c$  — Combined range-dependent factors that contribute to the reduction of the received signal energy
- $k$  — Boltzmann constant
- $T_s$  — System temperature in Kelvin
- $R$  — Target range in meters. The equation assumes the radar is monostatic.
- $L_a$  — One-way atmospheric absorption loss

- $L$  — Combined system losses
- $\Omega$  — Search volume in steradians

You can derive this equation based on assumptions about the SNR form of the standard radar equation. For more information about the SNR form of the standard radar equation, see the `radareqsnr` function. These are the assumptions:

- The radar is monostatic, so that  $R = R_t = R_r$ , where  $R_t$  is the range from the transmitter to the target and  $R_r$  is the range from the receiver to the target.
- The search time is the time the transmit beam takes to scan the entire search volume. As a result, you can express the search time,  $t_s$ , in terms of the search volume,  $\Omega$ , the area of the beam in steradians,  $\Omega_t$ , and the dwell time in seconds,  $T_d$ .

$$t_s = T_d \frac{\Omega}{\Omega_t}$$

- The transmit antenna beam has an ideal rectangular shape. As a result, you can express the transmit antenna gain,  $G_t$ , in terms of the angular area of the antenna beam.

$$G_t = \frac{4\pi}{\Omega_t}$$

- The receive antenna is ideal. This means you can express the receive antenna gain,  $G_r$ , in terms of the antenna effective aperture,  $A$ , and the radar operating frequency wavelength,  $\lambda$ .

$$G_r = \frac{4\pi A}{\lambda^2}$$

## References

- [1] Barton, David Knox. *Radar Equations for Modern Radar*. Artech House Radar Series. Boston, Mass: Artech House, 2013.
- [2] Skolnik, Merrill I. *Introduction to Radar Systems*. Third edition. McGraw-Hill Electrical Engineering Series. Boston, Mass. Burr Ridge, IL Dubuque, IA: McGraw Hill, 2001.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`radareqsearchrng` | `radareqsearchpap` | `radareqsnr` | `radareqrng` | `radareqpow`

**Introduced in R2021a**



# solidangle

Solid angle of region bounded by azimuth and elevation angles

## Syntax

```
omega = solidangle(az,el)
```

## Description

`omega = solidangle(az,el)` returns the solid angle `omega` in steradians for a region of a sphere bounded by the azimuth angles `az` and the elevation angles `el`. `az` and `el` must have the same number of columns or one of the inputs must be a 2-by-1 column vector.

## Examples

### Compute Solid Angle

Compute the solid angle for three regions of a sphere that have the same azimuth limits.

```
az = [0;65];
el = [-15 20 15;5 30 80];

omega = solidangle(az,el)

omega = 1x3
    0.3925    0.1792    0.8236
```

## Input Arguments

### az — Azimuth angles

two-row matrix

Azimuth angles in degrees, specified as a two-row matrix. Each column in `az` has the form `[az1;az2]`, where `az1` and `az2` are the azimuth limits of `omega` created by traveling from `az1` to `az2` counter-clockwise. `az1` and `az2` must be between -180 and 180.

Data Types: double

### el — Elevation angles

two-row matrix

Elevation angles in degrees, specified as a two-row matrix. Each column in `el` has the form `[el1;el2]`, where `el1` and `el2` are the limits of the elevation sector spanned by `omega`. `el1` and `el2` must be between -90 and 90.

Data Types: double

## Output Arguments

### **omega** — Solid angle

row vector

Solid angle in steradians, returned as a row vector. The output `omega` depends on the sizes of `az` and `el`:

- If both `az` and `el` are matrices, each element of `omega` is computed for azimuth and elevation angles in the corresponding column of `az` and `el`.
- If `az` is a column vector and `el` is a matrix, `omega` is computed assuming the same azimuth angles for all columns in `el`.
- If `az` is a matrix and `el` is a column vector, `omega` is computed assuming the same elevation angles for all columns in `az`.

## References

[1] Barton, David K. *Radar Equations for Modern Radar*. Artech House Radar Series. Norwood, Mass: Artech House, 2013.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

`radareqsearchrng` | `radareqsearchsnr` | `radareqsearchpap`

**Introduced in R2021a**

# binaryintloss

Loss due to M-of-N binary pulse integration

## Syntax

```
LB = binaryintloss(PD,PFA,N)
LB = binaryintloss(PD,PFA,N,M)
[LB,PDSP,PFASP] = binaryintloss( ___ )
```

## Description

LB = binaryintloss(PD,PFA,N) calculates the binary integration loss, LB, in dB due to M-of-N pulse integration. The function computes the loss assuming that you are using a square-law detector and a nonfluctuating target.

---

**Note** The number of detections, M in the M-of-N integration scheme is set to  $M=0.955*N^{0.8}$ . This value is close to the optimal value that results in the binary integration loss lower than 1.5 dB for the N in the range between [5, 700].

---

LB = binaryintloss(PD,PFA,N,M) calculates the binary integration loss using number of detections M.

[LB,PDSP,PFASP] = binaryintloss( \_\_\_ ) also calculates single-pulse probabilities of detection, PDSP, and single-pulse probabilities of false alarm, PFASP, which are required at the input of the binary integrator to achieve the desired PD and PFA. Specify the input arguments from any of the previous syntax.

## Examples

### Calculate M-of-N Binary Integration Loss

Calculate binary integration loss for 12 detections from 24 received pulses. Assume a probability of detection of 0.9 and probability of false alarm of 1e-6

```
PD = 0.9;
PFA = 1e-6;
N = 24;
M = 12;
binaryintloss(PD,PFA,N,M)
```

```
ans = 1.0596
```

## Input Arguments

### PD — Probability of detection

positive scalar | length-J vector

Probability of detection in the range  $[0.1, 0.999999]$ , specified as a positive scalar or as a length- $J$  vector with each element in the range  $[0.1, 0.999999]$ .

**PFA — Probability of false alarm**

positive scalar | length- $K$  vector

Probability of false alarm, specified as a positive scalar in the range  $[1e-15, 1e-3]$  or as a length- $K$  vector with each element in the range  $[1e-15, 1e-3]$ .

**N — Number of received pulses**

1 (default) | positive scalar

Number of received pulses, specified as a positive scalar.

**M — Number of detections**

$0.955 * N^{0.8}$  (default) | positive scalar

Number of detections, specified as positive scalar.

## Output Arguments

**LB — Binary integration loss**

$J$ -by- $K$  matrix

Binary integration loss, returned as a  $J$ -by- $K$  matrix in dB with rows corresponding to the number of elements in PD and columns corresponding to the number of elements in PFA.

**PDSP — Single-pulse probabilities of detection**

$J$ -by- $K$  matrix

Single-pulse probabilities of the detection, returned as a  $J$ -by- $K$  matrix with rows corresponding to the number of elements in PD and columns corresponding to the number of elements in PFA.

**PFASP — Single-pulse probabilities of false alarm**

$J$ -by- $K$  matrix

Single-pulse probabilities of the false alarm, returned as a  $J$ -by- $K$  matrix with rows corresponding to the number of elements in PD and columns corresponding to the number of elements in PFA.

## Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`cfarloss` | `matchingloss` | `detectability` | `mtiLoss`

**Introduced in R2021a**

## cfarloss

Loss due to constant false alarm rate (CFAR) adaptive processing

### Syntax

```
LCFAR = cfarloss(PFA,NRC)
LCFAR = cfarloss(PFA,NRC,Name,Value)
```

### Description

`LCFAR = cfarloss(PFA,NRC)` computes approximated CFAR loss, LCFAR, in dB for the probability of false alarm, PFA, and number of reference cells, NRC, that you specify. The function calculates loss for the cell-averaging (CA) CFAR method and a square-law detector based on the Gregers-Hansen's universal CFAR loss curve.

`LCFAR = cfarloss(PFA,NRC,Name,Value)` specifies additional options using name-value arguments. For example, `LCFAR = cfarloss(1e-8,4:4:64,'Method','CA')` computes approximate loss using the CA CFAR process.. You can specify multiple name-value arguments.

### Examples

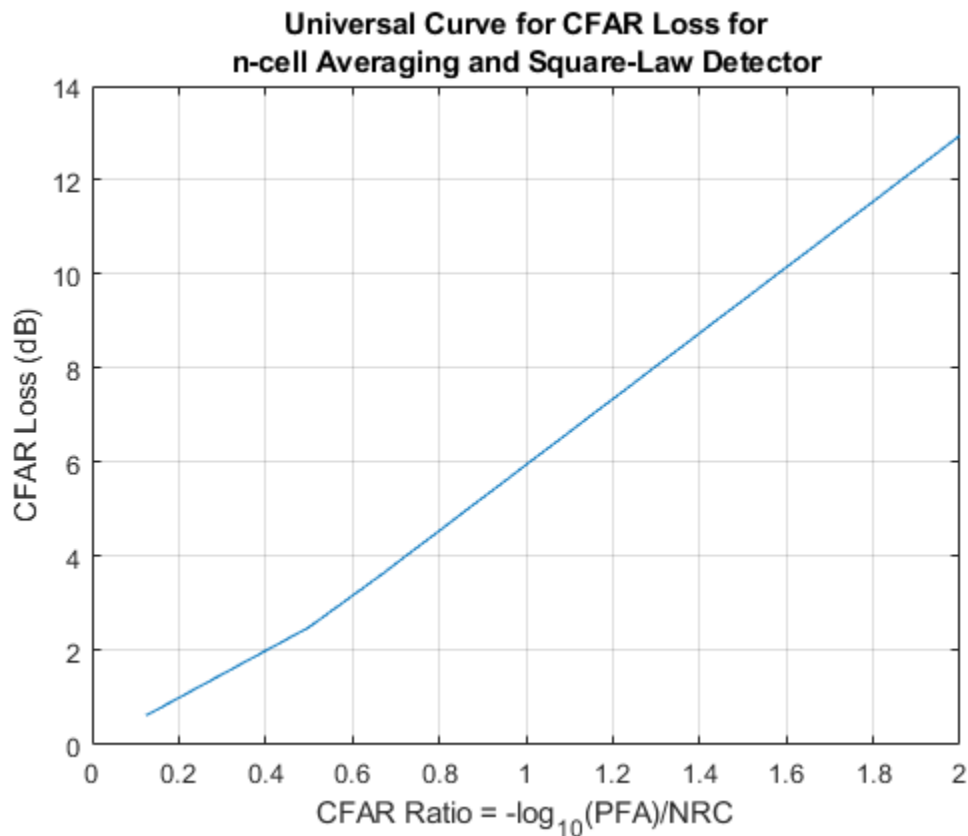
#### Compute CFAR Loss

Calculate the CFAR loss for an n-cell averaging and a square-law detector. Assume the numbers of reference cells from 4–64 and the probability of false alarm of  $1e-8$ .

```
PFA = 1e-8;
NRC = 4:4:64;
LCFAR = cfarloss(PFA,NRC);
```

Plot the resulting loss vs CFAR ratio. The CFAR ratio is calculated using the equation,  $X = -\log_{10}(\text{PFA})/\text{NRC}$ .

```
plot(-log10(PFA)./NRC,LCFAR)
grid on;
xlabel('CFAR Ratio = -log_{10}(PFA)/NRC');
ylabel('CFAR Loss (dB)');
title({'Universal Curve for CFAR Loss for',...
      'n-cell Averaging and Square-Law Detector'});
```



## Input Arguments

### PFA — Probability of false alarm

positive scalar | length- $K$  vector

Probability of false alarm, specified as a positive scalar in the range  $[1e-15, 1e-3]$  or as a length- $K$  vector with each element in the range  $[1e-15, 1e-3]$ .

### NRC — Number of reference cells

positive scalar | length- $K$  vector

Number of reference cells used in CFAR processing, specified as a positive scalar or length- $K$  vector.

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `LCFAR = cfarloss(1e-8,4:4:64,'DetectorType','Log')`

### Method — Type of CFAR process

'CA' (default) | 'GOCA'

Type of CFAR process, specified as either 'CA' for cell-averaging process or 'GOCA' for greatest-of cell-averaging process.

Example: 'Method', 'GOCA'

#### **DetectorType — Type of detector in use**

'SquareLaw' (default) | 'Linear' | 'Log'

Type of detector in use, specified as either 'SquareLaw', 'Linear', or 'Log'.

Example: 'DetectorType', 'Linear'

## **Output Arguments**

### **LCFAR — CFAR loss**

K-element vector

CFAR loss, returned as a K-element vector in dB.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

binaryintloss | matchingloss | mtiloss

**Introduced in R2021a**

# detectability

Radar detectability factor

## Syntax

```
D = detectability(PD,PFA)
D = detectability(PD,PFA,N)
D = detectability(PD,PFA,N,SW)
```

## Description

`D = detectability(PD,PFA)` returns the detectability factor of a single radar pulse given the probability of detection `PD` and probability of false alarm `PFA`.The function assumes that you are using a square-law detector and a nonfluctuating target.

`D = detectability(PD,PFA,N)` returns the detectability factor using the number of pulses for noncoherent integration `N`. .The function assumes that you are using a nonfluctuating target.

`D = detectability(PD,PFA,N,SW)` returns detectability factor using the Swerling case number `SW`. The function assumes you are using a chi-squared distributed target.

## Examples

### Detectability Factor for Swerling 1 Case Target

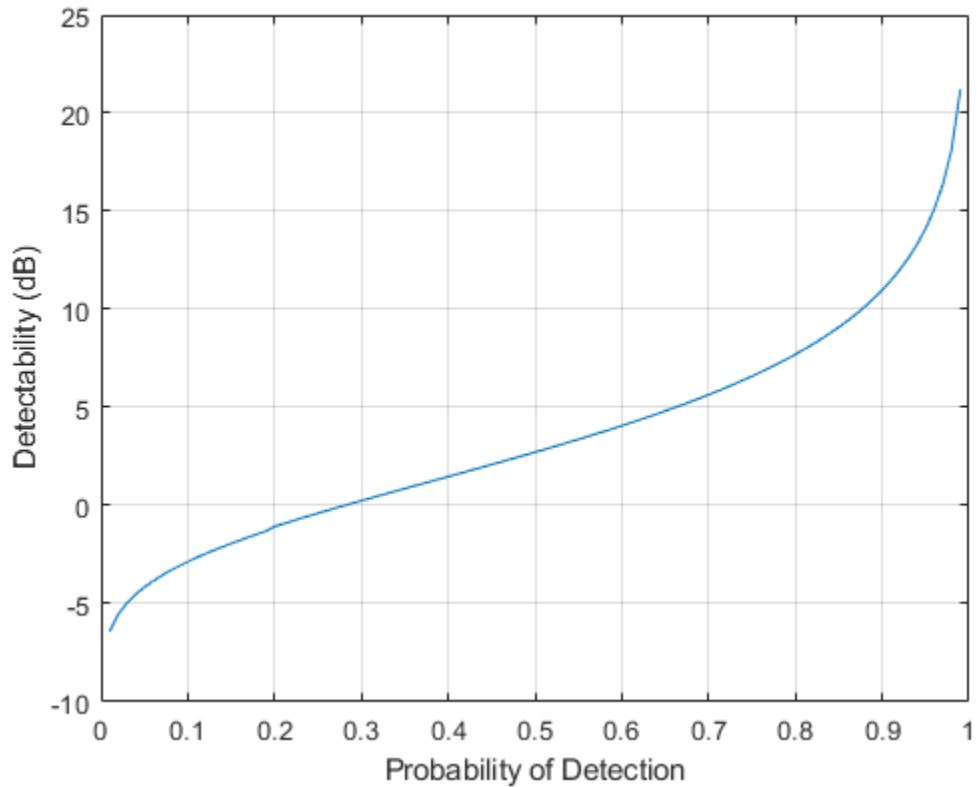
Calculate the detectability factor for a Swerling 1 case target. Assume a probability of detection from 0.01–0.99, probability of false alarm of 1e-6, and 24 received pulses.

```
PFA = 1e-6;
PD = 0.01:0.01:0.99;
N = 24;
D = detectability(PD,PFA,N,'Swerling1');
```

Plot the detectability factor.

```
plot(PD,D)
xlabel('Probability of Detection');
ylabel('Detectability (dB)');
grid on
```





## Input Arguments

### PD — Probability of detection

positive scalar | length- $J$  vector

Probability of detection, specified as a positive scalar in the range  $(0, 1)$  or as a length- $J$  vector with each element in the range  $(0, 1)$ .

### PFA — Probability of false alarm

positive scalar | length- $K$  vector

Probability of false alarm, specified as a positive scalar in the range  $(0, 1)$  or as a length- $K$  vector with each element in the range  $(0, 1)$ .

### N — Number of pulses for noncoherent integration

1 (default) | positive scalar

Number of pulses for noncoherent integration, specified as a positive scalar.

### SW — Swerling case number

'Swerling0' (default) | 'Swerling1' | 'Swerling2' | 'Swerling3' | 'Swerling4' | 'Swerling5'

Swerling case number, specified as one of these

- 'Swerling0'
- 'Swerling1'
- 'Swerling2'
- 'Swerling3'
- 'Swerling4'
- 'Swerling5'

## Output Arguments

### D — Detectability factor

*J*-by-*K* matrix

Detectability factor, returned as a *J*-by-*K* matrix in dB with rows corresponding to the number of elements in PD and columns corresponding to the number of elements in PFA.

## Algorithms

### Computation methods used in detectability function

The function computes detectability using the computation methods summarized in this table.

Swerling Case Number	PD is in the range [0.2, 1-1e-6] and PFA < 1e-4	PD outside the range [0.2, 1-1e-6] or PFA ≥ 1e-4
0 or 5	Shnidman's approximation	Exact computation
1, 2, 3, 4	Barton's universal equation	Exact computation

For Swerling1 and  $N = 1$  and Swerling2 and  $N$  set to any positive scalar, the function computes the radar detectability factor with no approximation errors using Barton's universal equation. For other Swerling cases, there are small approximation errors when PD is in the range [0.2, 1-1e-6] and  $PFA < 1e-4$ .

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

eclipsingfactor

### Topics

“Radar Vertical Coverage over Terrain”

“Modeling Target Position Errors due to Refraction”

**Introduced in R2021a**

# eclipsingfactor

Range-dependent eclipsing factor

## Syntax

```
FECL = eclipsingfactor(R,DU,PRF)
```

## Description

`FECL = eclipsingfactor(R,DU,PRF)` computes the range-dependent eclipsing factor FECL in decibels, given unambiguous range R duty cycle for a simple rectangular pulse or vector of samples from an arbitrary waveform DU and pulse repetition frequency PRF.

## Examples

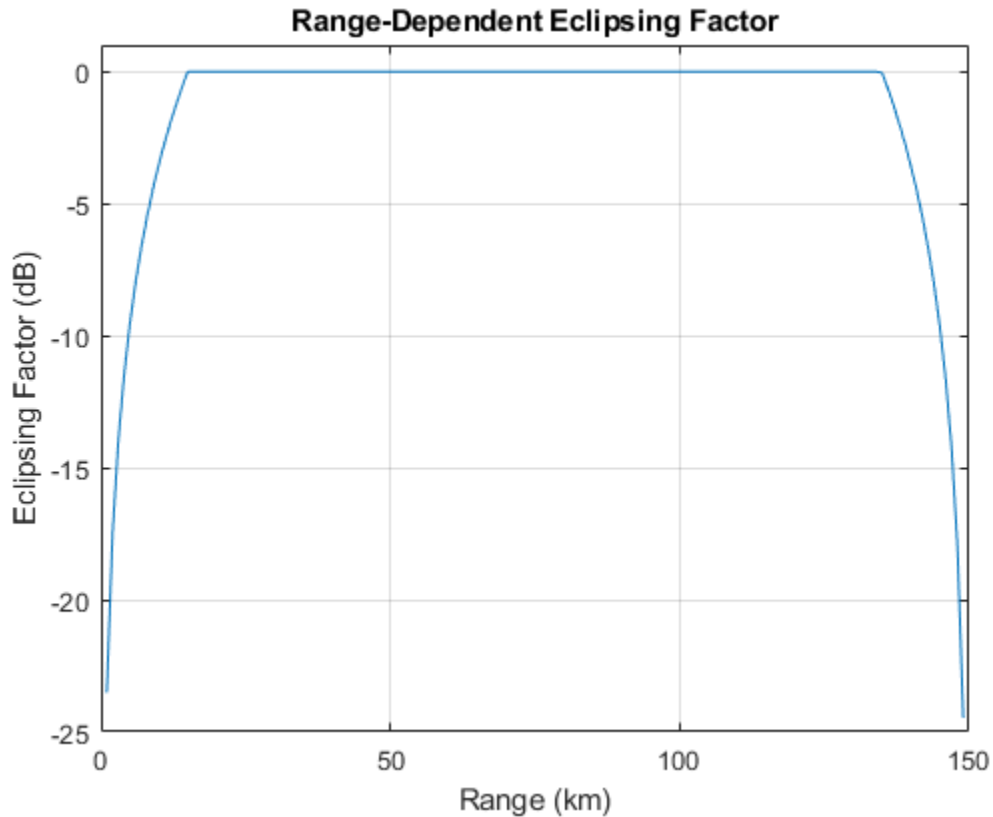
### Calculate and Plot Range-Dependent Eclipsing Factor

Calculate the range-dependent eclipsing factor at 1 km intervals between zero and the unambiguous range, R, assuming an unmodulated rectangular pulse with a duty cycle of 0.1 and the pulse repetition frequency of 1000 Hz.

```
DU = 0.1;  
PRF = 1e3;  
R = 0:1000:time2range(1/PRF);  
FECL = eclipsingfactor(R,DU,PRF);
```

Plot the range-dependent eclipsing factor.

```
plot(R*1e-3,FECL)  
xlabel('Range (km)');  
ylabel('Eclipsing Factor (dB)');  
ylim([-25 1]);  
grid on;  
title('Range-Dependent Eclipsing Factor');
```



## Input Arguments

### **R** — Range at which to compute eclipsing factor

positive scalar | length- $J$  vector

Range at which to compute the eclipsing factor, specified as a positive scalar or as a length- $J$  vector in meters.

### **DU** — Duty cycle

nonnegative scalar | length- $M$  vector

Duty cycle, specified as a nonnegative scalar in the range  $[0, 1]$  or length- $M$  vector with each element in the range  $[0, 1]$ .

- If you specify **DU** as a scalar, the eclipsing factor is computed for an unmodulated rectangular pulse with the specified duty cycle.
- If you specify **DU** as a length- $M$  vector, the eclipsing factor is computed for a waveform, using time domain samples taken over a one-pulse interval.

### **PRF** — Pulse repetition frequency

positive scalar | length- $K$  vector

Pulse repetition frequency, specified as a positive scalar or as a length- $K$  vector in Hz.

## Output Arguments

### FECL — Eclipsing factor

*J*-by-*K* matrix

Eclipsing factor, returned as a *J*-by-*K* matrix in decibels with rows corresponding to the ranges in *R* and columns corresponding to the values in *PRF*.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

eclipsingloss

**Introduced in R2021a**

## eclipsingloss

Loss due to pulse eclipsing

### Syntax

```
LECL = eclipsingloss(PD,PFA,N)
LECL = eclipsingloss(PD,PFA,N,DU)
LECL = eclipsingloss(PD,PFA,N,DU, SW)
```

### Description

`LECL = eclipsingloss(PD,PFA,N)` computes the statistical eclipsing loss, LECL, in decibels for an unmodulated rectangular pulse with a duty cycle of 0.1 given the probability of detection, PD, the probability of false alarm, PFA, and the number of received pulses, N. The function assuming you are using a square-law detector and a nonfluctuating target.

`LECL = eclipsingloss(PD,PFA,N,DU)` computes the statistical eclipsing loss for an unmodulated rectangular pulse given the duty cycle, DU, of the transmitted waveform as an additional input argument.

`LECL = eclipsingloss(PD,PFA,N,DU, SW)` computes the statistical eclipsing loss for radar echoes received from a chi-squared distributed target given the Swerling case number, SW, as an additional input argument.

### Examples

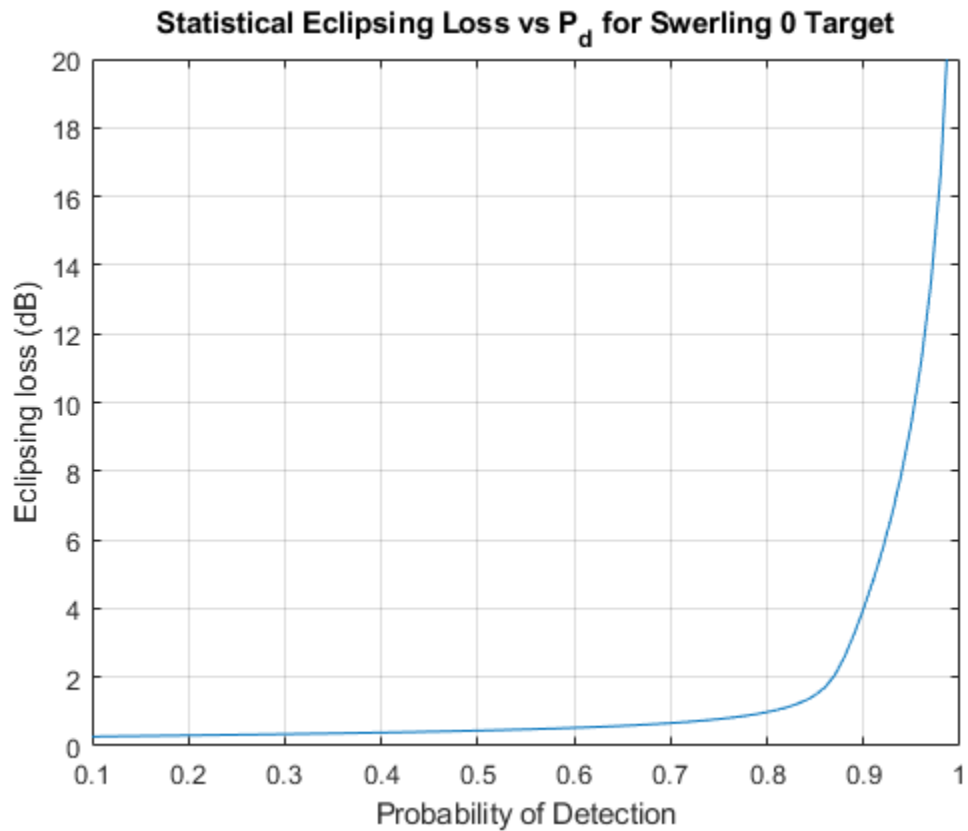
#### Eclipsing Loss for Single Unmodulated Rectangular Pulse

Compute the statistical eclipsing loss for a single unmodulated rectangular pulse. Specify the probability of detection from 0.1–0.99 and probability of false alarm of 1e-6.

```
PD = 0.1:0.01:0.99;
PFA = 1e-6;
N = 1;
LECL = eclipsingloss(PD,PFA,N);
```

Plot the eclipsing loss.

```
plot(PD,LECL)
ylim([0 20]);
xlabel('Probability of Detection');
ylabel('Eclipsing loss (dB)');
title('Statistical Eclipsing Loss vs P_d for Swerling 0 Target');
grid on;
```



## Input Arguments

### PD — Probability of detection

positive scalar | length- $J$  vector

Probability of detection, specified as a positive scalar in the range of  $[0.1, 0.999999]$  or as a length- $J$  vector with each element in the range  $[0.1, 0.999999]$ .

### PFA — Probability of false alarm

scalar | length- $K$  vector

Probability of false alarm, specified as a positive scalar or as a length- $K$  vector with each element in the range  $[1e-15, 1e-3]$ .

### N — Number of received pulses

positive scalar

Number of received pulses, specified as a positive scalar.

### DU — Duty cycle

0.1 (default) | scalar | length- $M$  vector

Duty cycle, specified as a scalar or length- $M$  vector.

- If you set **DU** as a scalar, the function computes the eclipsing loss for an unmodulated rectangular pulse with duty cycle in the range (0, 1).
- If you set **DU** as a length-*M* vector, the function computes the eclipsing loss for an arbitrary waveform specified using the time domain samples taken over a one pulse repetition interval.

**SW — Swerling case number**

'Swerling0' (default) | 'Swerling1' | 'Swerling2' | 'Swerling3' | 'Swerling4' | 'Swerling5'

Swerling case number, specified as one of these

- 'Swerling0'
- 'Swerling1'
- 'Swerling2'
- 'Swerling3'
- 'Swerling4'
- 'Swerling5'

**Output Arguments****LECL — Eclipsing loss**

*J*-by-*K* matrix

Eclipsing loss, returned as a *J*-by-*K* matrix in decibels with rows corresponding to PD and columns corresponding to PFA.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

binaryintloss | matchingloss | cfarloss

**Introduced in R2021a**



# matchinggain

Gain due to matched filtering

## Syntax

```
gain = matchinggain(pw,bw)
gain = matchinggain(pw,bw,lr)
```

## Description

`gain = matchinggain(pw,bw)` returns the gain due to matched filtering.

`gain = matchinggain(pw,bw,lr)` specifies the reduction in signal-to-noise ratio (SNR) gain due to nonideal filtering.

## Examples

### Range Processing Gain

Compute the range processing gain of a side-looking airborne synthetic aperture radar (SAR). The waveform has an effective pulse width of 100 microseconds. The antenna noise bandwidth is 5 MHz. Assume a nonideal range filtering loss of 1.3 dB.

```
pw = 100e-6;
bw = 5e6;
lr = 1.3;
```

Compute the range processing gain.

```
gain = matchinggain(pw,bw,lr)
gain = 25.6897
```

## Input Arguments

### **pw** — Effective pulse width

positive real scalar | vector

Effective pulse width of the radar waveform in seconds, specified as a positive real scalar or a vector.

Data Types: `double`

### **bw** — Noise bandwidth

positive real scalar | vector

Noise bandwidth at the antenna in hertz, specified as a positive real scalar or a vector.

Data Types: `double`

**lr — Reduction in SNR gain** $\emptyset$  (default) | nonnegative scalar

Reduction in signal-to-noise ratio (SNR) gain in decibels, specified as a nonnegative scalar. This argument corresponds to the loss with respect to the ideal gain. Typical window functions like `hamming` and `hann` exhibit losses on the order of 1 dB. The argument defaults to  $\emptyset$ , which assumes a rectangular window.

Data Types: `double`

**Output Arguments****gain — Gain due to matched filtering**`matrix`

Gain due to matched filtering in decibels, returned as a matrix. The rows of `gain` correspond to the pulse width values in `pw`. The columns of `gain` correspond to the bandwidth values in `bw`.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`sarazgain` | `sarchirprate` | `sarinttime` | `sarpointdopbw` | `sarprf` | `sarscenedopbw`

**Introduced in R2021a**

# matchingloss

Receiver filter matching loss

## Syntax

```
Lm = matchingloss(S,H)
```

## Description

`Lm = matchingloss(S,H)` calculates the receiver filter loss,  $L_m$ , in dB. The receiver loss is introduced due to a mismatch between the spectrum of the received signal,  $S$ , and the frequency response of the mismatched filter,  $H$ .

## Examples

### Calculate Matching Loss

Compute the matching loss for a rectangular pulse and a mismatched second-order Butterworth filter.

Define sampling frequency, pulsewidth, and filter bandwidth.

```
Fs = 10;    % Sampling frequency (Hz)
tau = 1.2;  % Pulsewidth (s)
B = 1.0;    % Filter bandwidth (Hz)
```

Calculate the rectangular pulse in the time domain.

```
s = ones(1,Fs*tau);
```

Calculate the spectrum of the received pulse.

```
nfft = 2^(nextpow2(tau*Fs)+1);
S = fft(s,nfft);
```

Calculate the frequency response of a second-order Butterworth filter with bandwidth  $B$ .

```
[b,a] = butter(2,B/Fs);
[H,w] = freqz(b,a,nfft,'whole',Fs);
```

Compute the matching loss for the pulsewidth-bandwidth product,  $\tau*B = 1.2$ .

```
Lm = matchingloss(S,H.')
```

```
Lm = 0.9806
```

## Input Arguments

### S — Spectrum of received signal

$J$ -by- $N$  matrix

Spectrum of the received signal, specified as a  $J$ -by- $N$  matrix with rows corresponding to spectra of  $J$  signals and columns corresponding to  $N$  frequency bins.

**H — Frequency response of mismatch filter**

$K$ -by- $N$  matrix

Frequency response of the mismatch filter, specified as a  $K$ -by- $N$  matrix with rows corresponding to frequency responses of  $K$  filters and columns corresponding to  $N$  frequency bins.

---

**Note** The columns of  $S$  and  $H$  must correspond to the same  $N$  frequency bins.

---

**Output Arguments****Lm — Matching loss**

$J$ -by- $K$  matrix

Matching loss, returned as a  $J$ -by- $K$  matrix in dB. The matching loss matrix is computed for each combination of  $J$  signals and  $K$  filters.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`cfarloss` | `eclipsingloss`

**Introduced in R2021a**

# mtifactor

Improvement factor due to moving target indicator (MTI) processing

## Syntax

```
IM = mtifactor(M,FREQ,PRF)
IM = mtifactor(M,FREQ,PRF,Name,Value)
```

## Description

`IM = mtifactor(M,FREQ,PRF)` calculates the MTI improvement factor in dB given the number of pulses in an  $(M - 1)$  delay canceler,  $M$ , the transmitted frequency, `FREQ`, and the pulse repetition frequency, `PRF`. This syntax assumes you are using coherent processing, a clutter with mean velocity of 0 m/s, and a standard deviation in clutter spread of 2 m/s.

`IM = mtifactor(M,FREQ,PRF,Name,Value)` specifies additional options using name-value arguments. For example, `IM = mtifactor(4,200e9,250,'IsCoherent',false)` calculates the MTI improvement factor assuming you are using noncoherent MTI processing. You can specify multiple name-value arguments.

## Examples

### Calculate MTI Improvement Factor for Three-Delay Canceler

Calculate the MTI improvement factor for a three-delay canceler with the transmitted frequency set to 300 MHz and the pulse repetition frequency set to 200 Hz.

```
M = 4;
FREQ = 300e6;
PRF = 200;
```

Calculate the coherent MTI improvement factor.

```
ImCoherent = mtifactor(M,FREQ,PRF)
```

```
ImCoherent = 55.3986
```

Calculate the noncoherent MTI improvement factor.

```
ImNoncoherent = mtifactor(M,FREQ,PRF,'IsCoherent',false)
```

```
ImNoncoherent = 49.4972
```

The noncoherent improvement factor is less than the coherent MTI factor.

## Input Arguments

**M** — Number of pulses in  $(M - 1)$  delay canceler

2 | 3 | 4

Number of pulses in the  $(M - 1)$  delay canceler, specified as 2, 3, or 4. For example, specify  $M = 2$  for a single-delay canceler,  $M = 3$  for a double-delay canceler, and so on.

**FREQ — Transmitted frequency**

positive scalar | length- $K$  vector

Transmitted frequency, specified as a positive scalar or length- $K$  vector in Hz.

**PRF — Pulse repetition frequency**

positive scalar | length- $K$  vector

Pulse repetition frequency, specified as a positive scalar or length- $K$  vector in Hz.

**Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `IM = mtifactor(4,200e9,250,'ClutterStandardDeviation',3)`

**IsCoherent — Coherent or non-coherent MTI processing**

true (default) | false

Coherent or noncoherent MTI processing, specified as a true or false.

- If you set the value of `IsCoherent` to `true`, the improvement factor is calculated assuming you are using a coherent MTI process.
- If you set the value of `IsCoherent` to `false`, the improvement factor is calculated assuming you are using a noncoherent MTI process.

Example: `IM = mtifactor(4,200e9,250,'IsCoherent',false)`

**ClutterStandardDeviation — Standard deviation of clutter spread**

2 (default) | positive scalar

Standard deviation of the clutter spread, specified as a positive scalar in m/s.

Example: `IM = mtifactor(4,200e9,250,'ClutterStandardDeviation',1)`

**NullVelocity — Null velocity**

0 (default) | positive scalar

Null velocity, specified as a positive scalar in m/s.

---

**Note** This name-value argument is valid only for coherent MTI processing. For noncoherent MTI processing, the function ignores this input.

---

Example: `IM = mtifactor(4,200e9,250,'NullVelocity',1)`

**ClutterVelocity — Clutter velocity**

0 (default) | positive scalar

Clutter velocity, specified as a positive scalar in m/s.

---

**Note** This name-value argument is valid only for coherent MTI processing. For noncoherent MTI processing, the function ignores this input.

---

Example: `IM = mtifactor(4,200e9,250,'ClutterVelocity',1)`

## Output Arguments

### **IM** — MTI improvement factor

1-by-*K* vector

MTI improvement factor, returned as 1-by-*K* vector in dB.

## References

[1] Barton, David Knox. *Radar Equations for Modern Radar*. Artech House Radar Series. Boston, MA. Artech House, 2013.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

`mtiLoss` | `cFarLoss`

**Introduced in R2021a**

## mtiloss

Losses due to moving target indicator (MTI) processing

### Syntax

```
[LI,LV] = mtiloss(PD,PFA,N)
[LI,LV] = mtiloss(PD,PFA,N,M)
[LI,LV] = mtiloss(PD,PFA,N,M,SW)
[LI,LV] = mtiloss( ____,Name,Value)
[ ____,LBP] = mtiloss( ____ )
```

### Description

`[LI,LV] = mtiloss(PD,PFA,N)` computes integration loss, `LI`, and velocity response loss, `LV`, due to MTI processing with a two-pulse (first-order) canceller given the probability of detection, `PD`, probability of false alarm, `PFA`, and the number of received pulses available at the MTI input, `N`.

The function computes the loss assuming you are using a square-law detector and a nonfluctuating target.

`[LI,LV] = mtiloss(PD,PFA,N,M)` computes losses due to MTI processing with an `M`-pulse canceller.

`[LI,LV] = mtiloss(PD,PFA,N,M,SW)` computes MTI losses for radar echoes received from a chi-squared distributed target specified using the Swerling case number, `SW`.

`[LI,LV] = mtiloss( ____,Name,Value)` computes MTI losses using one or more name-value arguments. For example, `[LI,LV] = mtiloss(0.64,1e-12,8,'Method','Batch')` calculates `LI` and `LV` for MTI with batch processing. Specify the name-value arguments after any of the input arguments from the previous syntax.

`[ ____,LBP] = mtiloss( ____ )` computes the blind phase loss `LBP` only when you set `IsQuadrature` name-value argument to `false`.

### Examples

#### Plot Velocity Response Loss

Calculate the velocity response loss for an MTI processing with a three-pulse canceller, with the probability of false alarm of  $1e-6$  and 24 pulses received from a nonfluctuating target.

```
PFA = 1e-6;
N = 24;
M = 3;
PD = 0.1:0.01:0.99;
[~,LV] = mtiloss(PD,PFA,N,M);
```

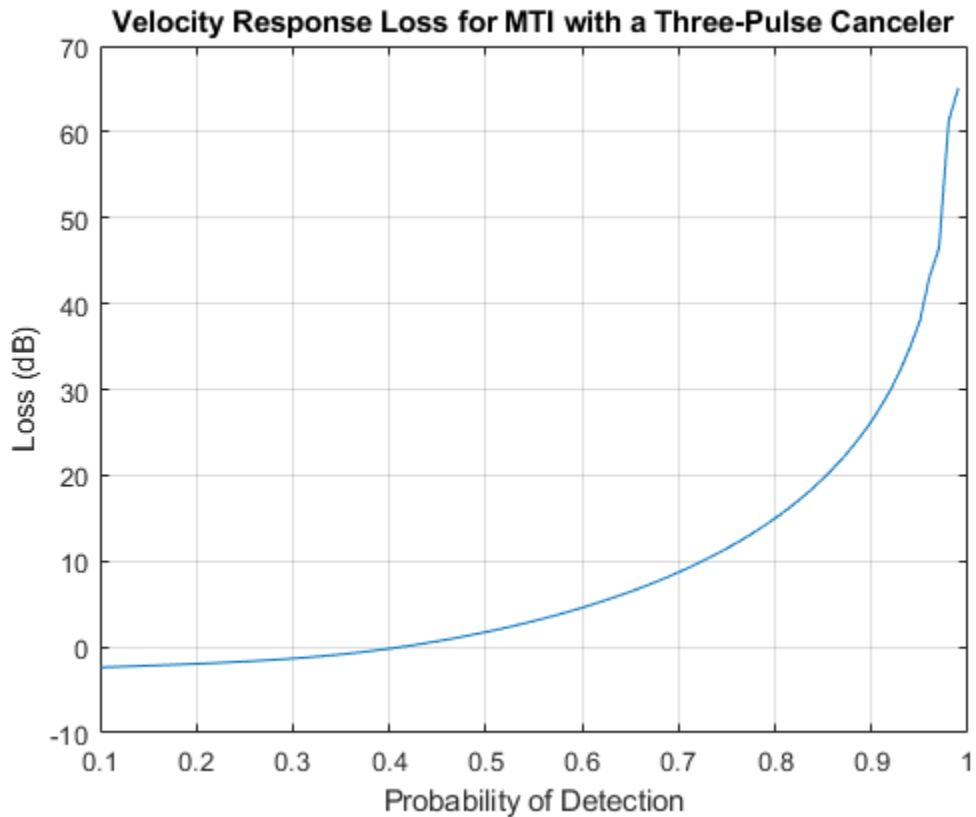
Plot the velocity response loss.



```

plot(PD,LV)
xlabel('Probability of Detection')
ylabel('Loss (dB)')
title('Velocity Response Loss for MTI with a Three-Pulse Canceler')
grid on

```



### Compute Integration or Noise Correlation Loss

Compute the noise correlation loss for MTI processing with a three-pulse canceler. Assume that the desired probability of detection is 0.9, the probability of false alarm is  $1e-6$ , and 24 pulses are received from a Swerling 1 target.

```

PD = 0.9;
PFA = 1e-6;
N = 24;
M = 3;
LI = mtiloss(PD,PFA,N,M, 'Swerling1')

LI = 2.0811

```

### Compute Blind Phase Loss for Two-Pulse Canceler

Compute the blind phase loss for an MTI with a two-pulse canceler with the desired probability of detection of 0.95, the probability of false alarm of  $1e-8$ , and 10 pulses received from a nonfluctuating target.

```
PD = 0.95;  
PFA = 1e-8;  
N = 10;  
[~,~,LBP] = mtloss(PD,PFA,N,'IsQuadrature',false)  
  
LBP = 2.3881
```

## Input Arguments

### PD — Probability of detection

positive scalar | length-*J* vector

Probability of detection in the range [0.1, 0.999999], specified as a positive scalar or as a length-*J* vector with each element in the range [0.1, 0.999999].

### PFA — Probability of false alarm

positive scalar | length-*K* vector

Probability of false alarm, specified as a positive scalar in the range [ $1e-15$ ,  $1e-3$ ] or as a length-*K* vector with each element in the range [ $1e-15$ ,  $1e-3$ ].

### N — Number of received pulses

positive integer equal to or greater than 2

Number of received pulses available at the input of the MTI, specified as a positive integer equal to or greater than 2.

### M — Number of pulses in M-pulse MTI canceler

2 (default) | positive integer in the range [2, 15]

Number of pulses in an M-pulse MTI canceler, specified as a positive integer in the range [2, 15]. The M-pulse canceler is constructed using cascading M-1 two-pulse cancellers.

### SW — Swerling case number

'Swerling0' (default) | 'Swerling1' | 'Swerling2' | 'Swerling3' | 'Swerling4' | 'Swerling5'

Swerling case number, specified as one of these

- 'Swerling0'
- 'Swerling1'
- 'Swerling2'
- 'Swerling3'
- 'Swerling4'
- 'Swerling5'

## Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `[LI,LV] = mtiloss(0.7,1e-8,10,'Method','Batch')`

### Method — Pulse processing method

'Sequential' (default) | 'Batch' | character vector | string scalar

Pulse processing method, specified as a character vector or string scalar.

- If you set 'Method' to 'Sequential', the received pulses are processed sequentially resulting in  $N-M$  pulses at the output of the pulse canceler.
- If you set 'Method' to 'Batch', the  $N$  received pulses are divided into  $N/(M+1)$  batches, which are processed separately resulting in  $N/(M+1)$  pulses at the output of the MTI.

Example: `[LI,LV] = mtiloss(0.7,1e-9, 8,'Method','Batch')`

### IsQuadrature — Quadrature-channel or single-channel MTI processing

true (default) | false

Quadrature-channel (vector) or single-channel MTI processing, specified as a logical value.

- If you set 'IsQuadrature' to true, the MTI processing has two parallel cancelers for the I and Q components. By default, the function sets 'IsQuadrature' to true and the blind phase loss output is zero.
- If you set 'IsQuadrature' to false, only the I or the Q channel is used for MTI resulting in blind phase loss LBP.

Example: `[LI,LV,LBP] = mtiloss(0.9,1e-8,10,'IsQuadrature',false)`

## Output Arguments

### LI — Integration loss

*J*-by-*K* matrix

Integration loss due to correlation in the noise samples at the output of the MTI filter, returned as a *J*-by-*K* matrix in dB with rows corresponding to the values in PD and columns to the values in PFA.

### LV — Velocity response loss

*J*-by-*K* matrix

Velocity response loss due to target velocity lying near the null of the MTI pulse canceler, returned as a *J*-by-*K* matrix in dB with rows corresponding to the values in PD and columns to the values in PFA.

### LBP — Blind phase loss

*J*-by-*K* matrix

Blind phase loss, returned as a *J*-by-*K* matrix in dB with rows corresponding to the values in PD and columns to the values in PFA. LBP is computed only when you set the value of the 'IsQuadrature' argument to false.

**See Also**

`binaryintloss` | `matchingloss` | `cfarloss` | `eclipsingloss`

**Introduced in R2021a**

# stcfactor

Sensitivity time control (STC) factor

## Syntax

```
FSTC = stcfactor(R,RC,X)
```

## Description

`FSTC = stcfactor(R,RC,X)` computes the range-dependent STC factor, FSTC, in dB given `R`, the STC cutoff range, `RC`, and an exponent `X`.

## Examples

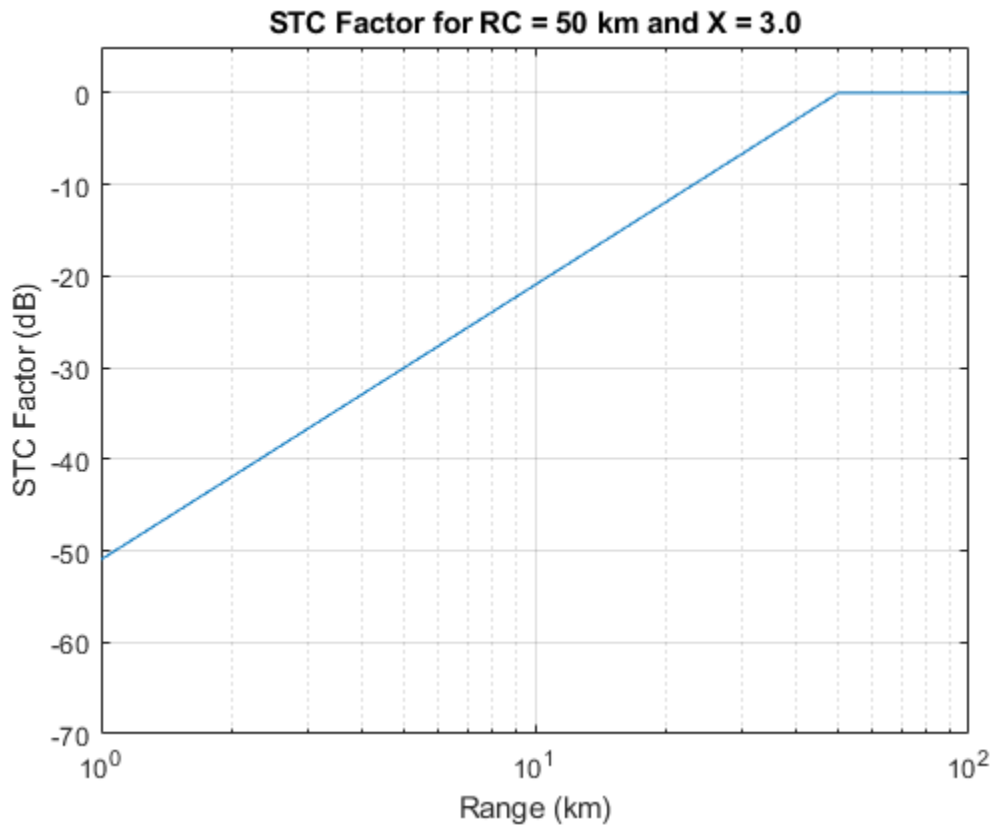
### Compute and Plot STC Factor

Compute the STC factor for the STC cutoff range of 50 km and the STC exponent of 3.0.

```
R = 0:1e3:100e3;  
RC = 50e3;  
X = 3.0;  
FSTC = stcfactor(R,RC,X);
```

Plot the STC factor.

```
semilogx(R*1e-3,FSTC)  
grid on;  
xlabel('Range (km)');  
ylabel('STC Factor (dB)');  
ylim([-70 5]);  
title('STC Factor for RC = 50 km and X = 3.0');
```



## Input Arguments

### **R** — Range at which to compute FSTC

positive scalar | length-*J* vector

Range at which to compute FSTC, specified as a positive scalar or length-*J* vector in meters.

### **RC** — STC cutoff range

positive scalar | length-*K* vector

STC cutoff range, specified as a positive scalar or as a length-*K* vector in meters.

### **X** — Exponent to maintain target detectability

positive scalar | length-*K* vector

Exponent to maintain the target detectability, specified as a positive scalar in the range of [3, 4] or as a length-*K* vector with each element in the range of [3, 4]. The exponent maintains the target detectability below the STC cutoff range.

## Output Arguments

### **FSTC** — Range-dependent STC factor

*J*-by-*K* matrix

Range-dependent STC factor, returned as a  $J$ -by- $K$  matrix in dB with rows corresponding to the values in  $R$  and columns corresponding to the ranges in  $RC$  and  $X$ .

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[detectability](#) | [eclipsingfactor](#)

**Introduced in R2021a**

## toccgh

Compute track probabilities using the CGH algorithm

### Syntax

```
[pdt,pft,eft] = toccgh(pd,pfa)
[pdt,pft,eft] = toccgh(pd,pfa,Name,Value)
```

```
toccgh( ___ )
```

### Description

`[pdt,pft,eft] = toccgh(pd,pfa)` computes track probabilities using the “Common Gate History Algorithm” on page 1-283. The algorithm uses a 2-out-of-3 track confirmation logic, where 2 hits must be observed in 3 updates for a track to be confirmed.

`[pdt,pft,eft] = toccgh(pd,pfa,Name,Value)` specifies additional options using name-value arguments. Options include the confirmation logic, the gate size in bins, and the gate growth sequence.

`toccgh( ___ )` with no output arguments plots the tracker operating characteristic (TOC), which is the probability of target track, `pdt`, as a function of the probability of false track, `pft`.

### Examples

#### Tracker Operating Characteristic Curves

The tracker operating characteristic (TOC) curve is a plot of the probability of a target track as a function of the probability of a false track. Plot the TOC curves for three different values of signal-to-noise ratio (SNR) assuming a 2/3 confirmation logic and use a one-dimensional constant-velocity Kalman filter to generate the tracker gate growth sequence.

Compute the probability of detection and the probability of false alarm for SNR values of 3, 6, and 9 dB. Assume a coherent receiver with a nonfluctuating target. Generate 20 probability-of-false-alarm values logarithmically equally spaced between  $10^{-10}$  and  $10^{-3}$  and calculate the corresponding probabilities of detection.

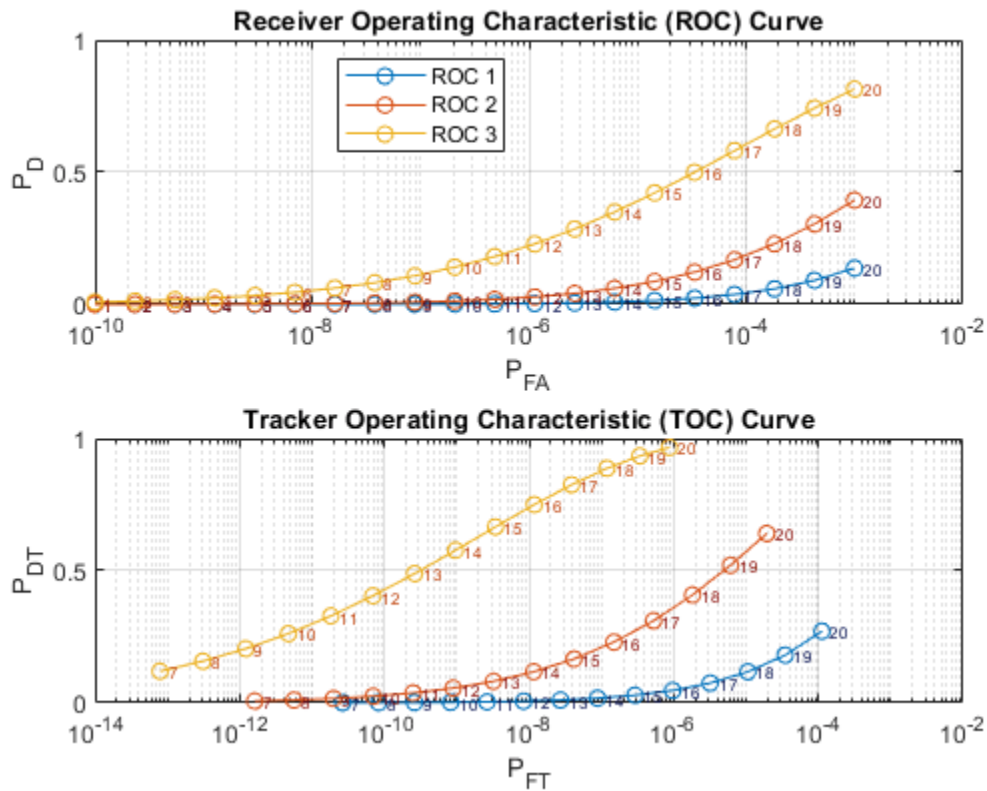
```
SNRdB = [3 6 9];
```

```
[pd,pfa] = rocsnr(SNRdB,'SignalType','NonfluctuatingCoherent', ...
    'NumPoints',20,'MaxPfa',1e-3);
```

Compute and plot the TOC curves and the corresponding receiver operating characteristic (ROC) curves.

```
toccgh(pd,pfa)
```





### Compute Track Probabilities

Compute the probability of target track, the probability of false track, and the expected number of false tracks corresponding to a probability of detection of 0.9, a probability of false alarm of  $10^{-6}$ , and a 3-of-5 track confirmation logic.

```
pd = 0.9;
pfa = 1e-6;
logic = [3 5];
```

Use a modified version of the default one-dimensional constant-velocity Kalman filter to generate the tracker gate growth sequence. Specify an update time of 0.3 second and a maximum target acceleration of 20 meters per square second.

```
KFpars = {'UpdateTime',0.3,'MaxAcceleration',20};
```

Compute the probabilities and the expected number of false tracks.

```
[pdf,pft,eft] = toccgh(pd,pfa,'ConfirmationThreshold',logic,KFpars{:})
```

```
pdf = 0.9963
```

```
pft = 2.1555e-19
```

```
eft = 1
```

### Custom Gate Growth Sequence

Use the common gate history algorithm to compute the probability of target track and the probability of track for a probability of detection of 0.5 and a probability of false alarm of  $10^{-3}$ . Use a custom gate growth sequence and a confirmation threshold of 3/4.

```
pd = 0.5;
pfa = 1e-3;

cp = [3 4];
gs = [21 39 95 125];
```

Compute the probabilities.

```
[pdf,pft] = toccgh(pd,pfa,'ConfirmationThreshold',cp, ...
    'GateGrowthSequence',gs)

pdf = 0.5132
pft = 9.9973e-07
```

### Varying False-Alarm Probabilities

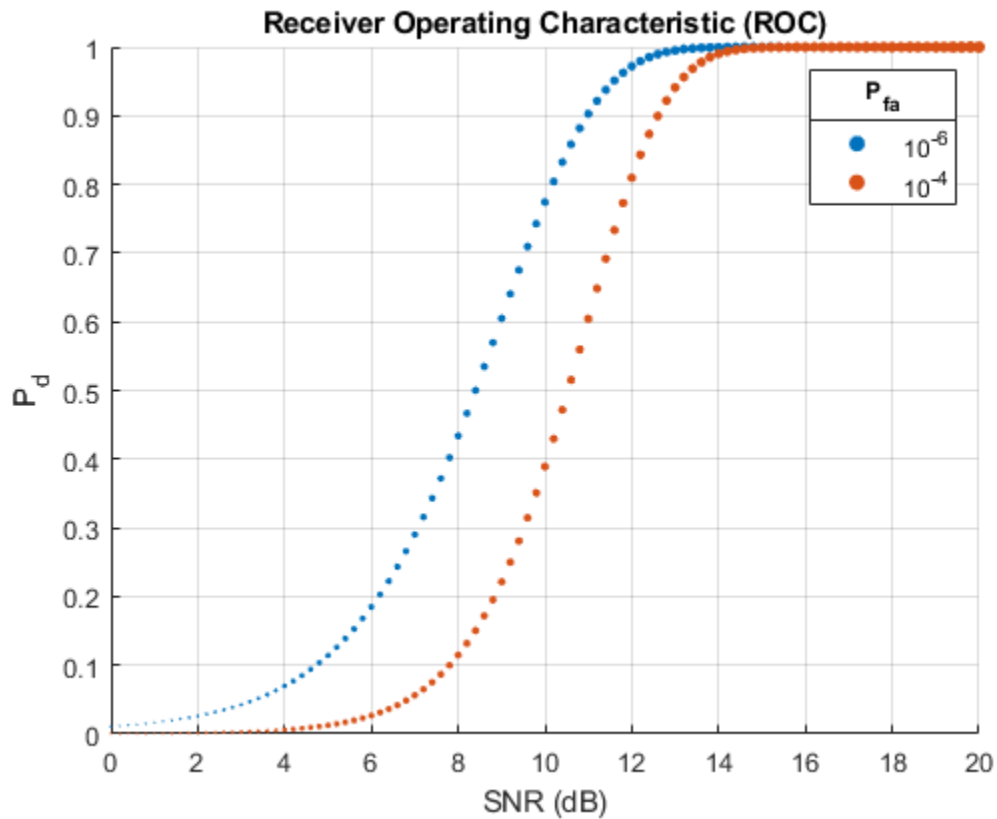
Investigate how receiver operating characteristic (ROC) and tracker operating characteristic (TOC) curves change with the probability of false alarm.

Compute probability-of-detection and signal-to-noise-ratio (SNR) values corresponding to probabilities of false alarm of  $10^{-4}$  and  $10^{-6}$ . Assume a coherent receiver with a nonfluctuating target. Plot the resulting ROC curves. Use larger markers to denote a larger SNR value.

```
pfa = [1e-4 1e-6];
[pd,SNRdB] = rocpfa(pfa,'SignalType','NonfluctuatingCoherent');

scatter(SNRdB,pd,max(SNRdB,1),'filled')

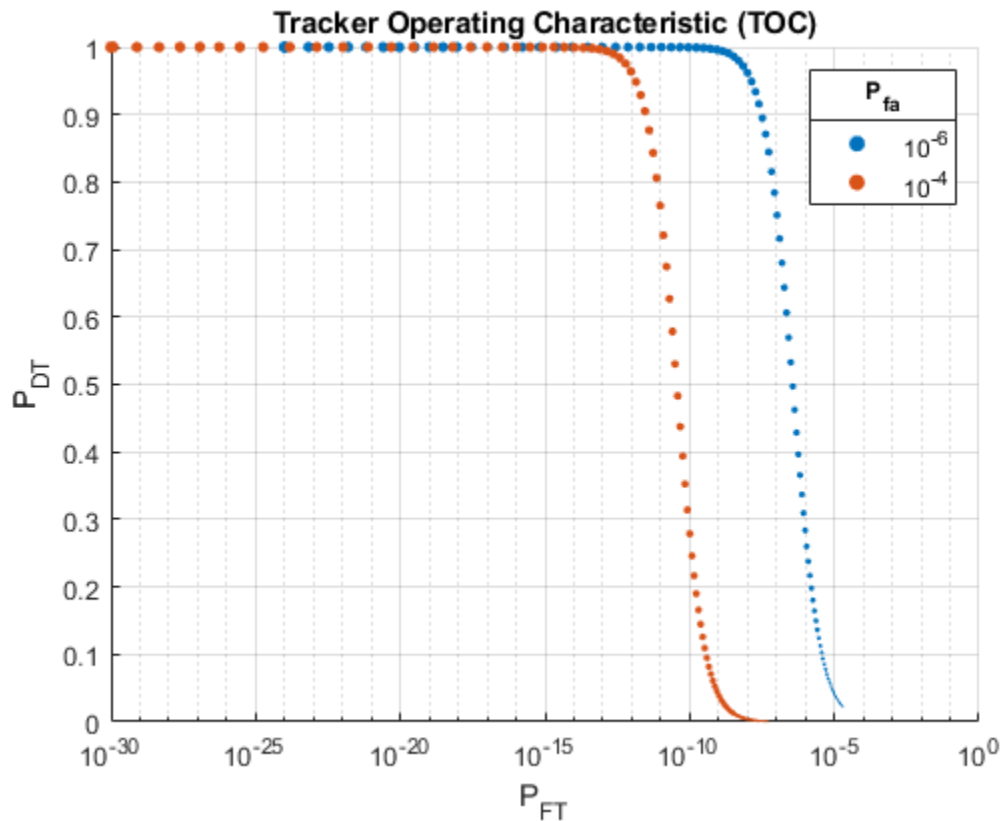
title('Receiver Operating Characteristic (ROC)')
xlabel('SNR (dB)')
ylabel('P_d')
grid on
title(legend('10^{-6}','10^{-4}'),'P_{fa}')
```



Compute the TOC curves using the probabilities of detection and probabilities of false alarm that you obtained. As the SNR increases, the probability of a false track in the presence of target detection increases. As the SNR decreases, the probability of target detection decreases, thereby increasing the probability of a false track.

```
[pct,pcf] = toccgh(pd.',pfa);
scatter(pcf,pct,max(SNRdB,1),'filled')

set(gca,'XScale','log')
title('Tracker Operating Characteristic (TOC)')
xlabel('P_{FT}')
ylabel('P_{DT}')
grid on
title(legend('10^{-6}','10^{-4}'),'P_{fa}')
```



## Input Arguments

### pd — Probability of detection

vector | matrix

Probability of detection, specified as a vector or a matrix of values in the range [0, 1].

- If `pd` is a vector, then it must have the same number of elements as `pfa`
- If `pd` is a matrix, then its number of rows must equal the number of elements of `pfa`. In that case, the number of columns of `pd` equals the length of the signal-to-noise (SNR) ratio input to `rocsnr` or output by `roc pfa`.

---

**Note** If you use `roc pfa` to obtain `pd`, you must transpose the output before using it as input to `toccg`. If you use `rocsnr` to obtain `pd`, you do not have to transpose the output.

---

Example: `[pd,pfa] = rocsnr(6)` returns single-pulse detection probabilities and false-alarm probabilities for a coherent receiver with a nonfluctuating target and a signal-to-noise ratio of 6 dB.

Data Types: double

### pfa — Probability of false alarm

vector

Probability of false alarm per cell (bin), specified as a vector of values in the range [0, 1].

---

**Tip** Use `pfa` values of  $10^{-3}$  or smaller to satisfy the assumptions of the common gate history algorithm.

---

Example: `[pd,pfa] = rocsnr(6)` returns single-pulse detection probabilities and false-alarm probabilities for a coherent receiver with a nonfluctuating target and a signal-to-noise ratio of 6 dB.

Data Types: `double`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'UpdateTime',0.25,'MaximumAcceleration',8` specifies that the 1-D constant-velocity track Kalman filter used to compute the track gate growth has an update time of 0.25 second and a maximum acceleration of targets of interest of 8 meters per square second.

### ConfirmationThreshold — Confirmation threshold

[2 3] (default) | two-element row vector of positive integers | positive integer scalar

Confirmation threshold, specified as a two-element row vector of positive integers or a scalar. The two-element vector  $[M\ N]$  corresponds to an  $M$ -out-of- $N$  or  $M/N$  confirmation logic, a test that stipulates that an event must occur at least  $M$  times in  $N$  consecutive updates.

- A track is confirmed if there are at least  $M$  detections in  $N$  updates.
- A track is deleted if there are less than  $M$  detections in  $N$  updates.

If this argument is specified as a scalar, `toccgh` treats it as a two-element vector with identical elements.  $N$  cannot be larger than 50.

Data Types: `double`

### NumCells — Number of cells

16384 (default) | positive integer scalar

Number of cells, specified as a positive integer scalar. Use this argument to compute the expected number of false tracks.

Data Types: `double`

### NumTargets — Number of targets

1 (default) | positive integer scalar

Number of targets, specified as a positive integer scalar. Use this argument to compute the expected number of false tracks.

Data Types: `double`

### UpdateTime — Update time for Kalman filter

0.5 (default) | positive scalar in seconds

Update time for the default one-dimensional constant-velocity Kalman filter, specified as a positive scalar in seconds. This argument impacts the track gate growth.

Data Types: `double`

### **MaxAcceleration — Maximum acceleration of targets of interest**

10 (default) | nonnegative scalar in meters per square second

Maximum acceleration of targets of interest, specified as a nonnegative scalar in meters per square second. Use this input to tune the process noise in the default one-dimensional constant-velocity Kalman filter. This argument impacts the track gate growth.

Data Types: `double`

### **Resolution — Range and range-rate resolution**

[1 1] (default) | two-element row vector of positive values

Range and range-rate resolution, specified as a two-element row vector of positive values. The first element of 'Resolution' is the range resolution in meters. The second element of 'Resolution' is the range rate resolution in meters per second. This argument is used to convert the predicted tracker gate size to bins.

Data Types: `double`

### **GateGrowthSequence — Tracker gate growth sequence**

vector of positive integers

Tracker gate growth sequence, specified as a vector of positive integers. The values in the vector represent gate sizes in bins corresponding to  $N$  possible misses in  $N$  updates, where  $N$  is specified using 'ConfirmationThreshold'. If 'ConfirmationThreshold' is a two-element vector, then  $N$  is the second element of the vector.

If this argument is not specified, `toccgk` generates the tracker gate growth sequence using a one-dimensional constant-velocity Kalman filter implemented as a `trackingKF` object with these settings:

- Update time — 0.5 second
- Maximum target acceleration — 10 meters per square second
- Range resolution — 1 meter
- Range rate resolution — 1 meter per second
- `StateTransitionModel` — [1 dt; 0 1], where  $dt$  is the update time
- `StateCovariance` — [0 0; 0 0], which means the initial state is known perfectly
- `MeasurementNoise` — 0
- `ProcessNoise` —  $[dt^4/4 \ dt^3/2; \ dt^3/2 \ dt^2]*q$ , where  $dt$  is the update time, the tuning parameter  $q$  is  $amax^2*dt$ , and  $amax$  is the maximum acceleration. The tuning parameter is given in Equation 1.5.2-5 of [2].

To compute the gate sizes, the algorithm:

- 1 Uses the `predict` function to compute the predicted state error covariance matrix.
- 2 Calculates the area of the error ellipse as  $\pi$  times the product of the square roots of the eigenvalues of the covariance matrix.

- 3** Divides the area of the error ellipse by the bin area to express the gate size in bins. The bin area is the product of the range resolution and the range rate resolution.

If this argument is specified, then the 'UpdateTime', 'MaxAcceleration', and 'Resolution' arguments are ignored.

Example: [21 39 95 125 155 259 301] specifies a tracker gate growth sequence that occurs on some radar applications.

Data Types: double

## Output Arguments

### **pdt** — Probability of true target track in presence of false alarms

matrix

Probability of true target track in the presence of false alarms, returned as a matrix. **pdt** has the same size as **pd**.

### **pft** — Probability of false track in presence of targets

matrix

Probability of false alarm track in the presence of targets, returned as a matrix. **pft** has the same size as **pd**.

### **eft** — Expected number of false tracks

matrix

Expected number of false tracks, returned as a matrix of the same size as **pd**. **toccgh** computes the expected number of tracks using

$$E_{ft} = P_{ft,nt}N_c + P_{ft}N_t,$$

where  $P_{ft,nt}$  is the probability of false track in the absence of targets,  $N_c$  is the number of resolution cells specified in 'NumCells',  $P_{ft}$  is the probability of false track in the presence of targets, and  $N_t$  is the number of targets specified in 'NumTargets'.

## More About

### Common Gate History Algorithm

The common gate history (CGH) algorithm was developed by Bar-Shalom and collaborators and published in [1]. For more information about the CGH algorithm, see "Assessing Performance with the Tracker Operating Characteristic".

The algorithm proceeds under these assumptions:

- A track is one of these:
  - 1** Detections from targets only
  - 2** Detections from false alarms only
  - 3** Detections from targets and from false alarms
- The probability of more than one false alarm in a gate is low, which is true when the probability of false alarm  $P_{fa}$  is low ( $P_{fa} \leq 10^{-3}$ ).

- The location of a target in a gate obeys a uniform spatial distribution.

The algorithm sequentially generates the gate history vector  $\omega = [\omega_l, \omega_{lt}, \lambda]$ , where:

- $\omega_l$  is the number of time steps since the last detection, either of a target or of a false alarm.
- $\omega_{lt}$  is the number of time steps since the last detection of a target.
- $\lambda$  is the number of detections.

The state vector evolves as a Markov chain by means of these steps:

- 1 The algorithm initially creates a track. Only two events can initialize a track:

- A target detection
- A false alarm

- 2 There are only four types of events that continue a track:

- $A_1$  — *No detection*

Events of Type 1 occur with probability

$$P\{A_1\} = \left(1 - \frac{g(\omega_l)}{g(\omega_{lt})} P_d\right) (1 - P_{fa})^{g(\omega_l)}$$

where  $P_d$  is the probability of detection specified using `pd`,  $P_{fa}$  is the probability of false alarm specified using `pfa`,  $g(\omega_l)$  is the gate size at step  $\omega_l$ , and  $g(\omega_{lt})$  is the gate size at step  $\omega_{lt}$ .

---

**Note** To reduce  $P_d$  to a lower effective value, `toccg` weights it with the ratio

$$\frac{g(\omega_l)}{g(\omega_{lt})} = \frac{\text{Actual gate size}}{\text{Size of gate taking into account the time elapsed since the last target detection}'}$$

which assumes a uniform spatial distribution of the location of a target in a gate. The gate sizes are specified using '`GateGrowthSequence`'.

---

Events of Type 1 update the gate history vector as  $[\omega_l, \omega_{lt}, \lambda] \rightarrow [\omega_l + 1, \omega_{lt} + 1, \lambda]$ .

- $A_2$  — *Target detection*

Events of Type 2 occur with probability

$$P\{A_2\} = \frac{g(\omega_l)}{g(\omega_{lt})} P_d (1 - P_{fa})^{g(\omega_l)}$$

and update the gate history vector as  $[\omega_l, \omega_{lt}, \lambda] \rightarrow [1, 1, \lambda + 1]$ .

- $A_3$  — *False alarm*

Events of Type 3 occur with probability

$$P\{A_3\} = \left(1 - (1 - P_{fa})^{g(\omega_l)}\right) \left(1 - \frac{g(\omega_l)}{g(\omega_{lt})} P_d\right)$$

and update the gate history vector as  $[\omega_l, \omega_{lt}, \lambda] \rightarrow [1, \omega_{lt} + 1, \lambda + 1]$ .



- $A_4$  — Target detection and false alarm

Events of Type 4 occur with probability

$$P\{A_4\} = \left(1 - (1 - P_{fa})^{g(\omega_l)}\right) \left(\frac{g(\omega_l)}{g(\omega_{lt})} P_d\right)$$

and cause the track to split into a false track and a true track:

- $A_{s,2a}$  — Continue with  $A_3$ , updating  $[\omega_l, \omega_{lt}, \lambda] \rightarrow [1, \omega_{lt} + 1, \lambda + 1]$ .
- $A_{s,2b}$  — Continue with  $A_2$ , updating  $[\omega_l, \omega_{lt}, \lambda] \rightarrow [1, 1, \lambda + 1]$ .

At each step, the algorithm multiplies each track probability by the probability of the event that continues the track.

- 3 The procedure then lumps together the tracks that have a common gate history vector  $\omega$  by adding their probabilities:

- Tracks continued with  $A_4$  are lumped with tracks that continue with  $A_3$  (one false alarm only).
- Tracks continued with  $A_4$  are lumped with tracks that continue with  $A_2$  (target detection only).

This step controls the number of track states within the Markov chain.

At the end, the algorithm computes and assigns the final probabilities:

- A target track is a sequence of detections that satisfies the  $M/N$  confirmation logic and contains at least one detection from a target. To compute the probability of target track:
  - 1 Determine the sequences that satisfy the confirmation logic under the assumption  $A_{s,2b}$  that  $A_4$  yields  $A_2$ .
  - 2 Separately store these probabilities.
- To compute the probability of false track:
  - 1 Compute the probability of target track under the assumption  $A_{s,2a}$  that  $A_4$  yields  $A_3$ .
  - 2 Subtract this probability from the probability of all detection sequences that satisfy the confirmation logic.

## References

- [1] Bar-Shalom, Yaakov, Leon J. Campo, and Peter B. Luh. "From Receiver Operating Characteristic to System Operating Characteristic: Evaluation of a Track Formation System." *IEEE® Transactions on Automatic Control* 35, no. 2 (February 1990): 172-79. <https://doi.org/10.1109/9.45173>.
- [2] Bar-Shalom, Yaakov, Peter K. Willett, and Xin Tian. *Tracking and Data Fusion: A Handbook of Algorithms*. Storrs, CT: YBS Publishing, 2011.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

rocpfa | rocsnr

## **Topics**

“Assessing Performance with the Tracker Operating Characteristic”

“Radar Vertical Coverage over Terrain”

“Linear Kalman Filters”

**Introduced in R2021a**

# sarazgain

SAR azimuth processing gain

## Syntax

```
ag = sarazgain(r,lambda,v,azres,prf)
ag = sarazgain( ___,Name,Value)
```

## Description

`ag = sarazgain(r,lambda,v,azres,prf)` computes the azimuth processing gain due to the coherent integration of multiple pulses, either by presumming or through actual Doppler processing.

`ag = sarazgain( ___,Name,Value)` specifies additional options using name-value arguments. Options include the azimuth impulse broadening factor and the Doppler cone angle.

## Examples

### Azimuth Processing Gain

Compute the azimuth processing gain of a side-looking airborne SAR operating in broadside at a wavelength of 0.05 m with a sensor velocity of 100 m/s and a PRF of 2 kHz for a target at 5 km. The cross-range resolution of the image is 1.5 m. Assume an azimuth broadening factor of 1.2 and a nonideal azimuth filtering loss of 1.2 dB.

```
lambda = 0.05;
PRF = 2e3;
R = 5e3;
res = 1.5;
v = 100;
La = 1.2;
azb = 1.2;
```

Compute the azimuth processing gain.

```
azgain = sarazgain(R,lambda,v,res,PRF,'AzimuthBroadening',azb, ...
    'AzimuthFilteringLoss',La)
```

```
azgain = 31.8103
```

## Input Arguments

### **r** — Range from target to antenna

positive real scalar | vector

Range from target to antenna in meters, specified as a positive real scalar or a vector.

Data Types: double

**lambda — Radar wavelength**

positive real scalar | vector

Radar wavelength in meters, specified as a positive real scalar or a vector.

Data Types: double

**v — Sensor velocity**

positive real scalar

Sensor velocity in meters per second, specified as a positive real scalar.

Data Types: double

**azres — Image azimuth or cross-range resolution**

positive real scalar

Image azimuth or cross-range resolution in meters, specified as a positive real scalar.

Data Types: double

**prf — Radar pulse repetition frequency**

positive real scalar

Radar pulse repetition frequency (PRF) in hertz, specified as a positive real scalar.

Data Types: double

**Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'AzimuthBroadening',1.2,'ConeAngle',60`

**AzimuthBroadening — Azimuth impulse broadening factor**

1 (default) | positive real scalar

Azimuth impulse broadening factor due to data weighting or windowing for sidelobe control, specified as a positive real scalar. This argument expresses the actual -3 dB mainlobe width with respect to the nominal width. Typical window functions like `hamming` and `hann` exhibit values in the range from 1 to 1.5.

Data Types: double

**AzimuthFiltering Loss — Reduction in SNR gain**

0 (default) | nonnegative scalar

Reduction in signal-to-noise ratio (SNR) gain in decibels, specified as a nonnegative scalar. This argument corresponds to the loss with respect to the ideal gain. Typical window functions like `hamming` and `hann` exhibit losses on the order of 1 dB. The argument defaults to 0, which assumes a rectangular window.

Data Types: double

**ConeAngle — Doppler cone angle**

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: double

**Output Arguments****ag — Azimuth processing gain**

matrix

Azimuth processing gain, returned as a matrix. The rows of `ag` correspond to the range values in `r` and its columns correspond to the wavelength values in `lambda`.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`matchinggain` | `sarchirprate` | `sarinttime` | `sarpointdopbw` | `sarprf` | `sarscenedopbw`

**Introduced in R2021a**

## sarchirprate

Azimuth chirp rate of received signal for SAR

### Syntax

```
acr = sarchirprate(r,lambda,v)
acr = sarchirprate(r,lambda,v,dcang)
```

### Description

`acr = sarchirprate(r,lambda,v)` computes the nominal azimuth chirp rate at which the azimuth signal changes frequency as the sensor illuminates a scatterer.

`acr = sarchirprate(r,lambda,v,dcang)` specifies the Doppler cone angle that identifies the direction towards the scene relative to the direction of motion of the array.

### Examples

#### Azimuth Chirp Rate

Compute the azimuth chirp rate of received signal for a side-looking airborne synthetic aperture radar (SAR) operating in broadside at a wavelength of 0.03 m with a sensor velocity of 100 m/s for a target at 10 km. The sensor illuminates the scatterer at a Doppler cone angle of 90°.

```
lambda = 0.03;
R = 10e3;
v = 100;
```

Compute the azimuth chirp rate.

```
azchirp = sarchirprate(R,lambda,v)
```

```
azchirp = 66.6667
```

### Input Arguments

#### **r** — Range from target to antenna

positive real scalar | vector

Range from target to antenna in meters, specified as a positive real scalar or a vector.

Data Types: `double`

#### **lambda** — Radar wavelength

positive real scalar | vector

Radar wavelength in meters, specified as a positive real scalar or a vector.

Data Types: `double`

**v — Sensor velocity**

positive real scalar

Sensor velocity in meters per second, specified as a positive real scalar.

Data Types: double

**dcang — Doppler cone angle**

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: double

**Output Arguments****acr — Nominal azimuth chirp rate**

matrix

Nominal azimuth chirp rate in hertz per second, returned as a matrix. The rows of `acr` correspond to the range values in `r`. The columns of `acr` correspond to the wavelength values in `lambda`.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`matchinggain` | `sarazgain` | `sarinttime` | `sarpointdopbw` | `sarprf` | `sarscenedopbw`

**Introduced in R2021a**

## sarpointdopbw

Doppler bandwidth due to cross-range platform motion

### Syntax

```
dbwch = sarpointdopbw(v,azres)
dbwch = sarpointdopbw(v,azres,Name,Value)
```

### Description

`dbwch = sarpointdopbw(v,azres)` returns the Doppler bandwidth of a single scatterer (chirped) due to cross-range platform motion as the sensor illuminates the scatterer.

`dbwch = sarpointdopbw(v,azres,Name,Value)` specifies additional options using name-value arguments. Options include the azimuth impulse broadening factor and the Doppler cone angle.

### Examples

#### Doppler Bandwidth of Single Scatterer

A side-looking airborne synthetic aperture radar (SAR) operates in broadside at a wavelength of 0.03 m with a sensor velocity of 100 m/s. The sensor illuminates a scatterer over a small cone angle interval having a cross-range resolution of 1 m and Doppler cone angle of 90 degrees. Compute the Doppler bandwidth of the received chirped signal.

```
azres = 1;
v = 100;
```

Compute the Doppler bandwidth.

```
bwchirp = sarpointdopbw(v,azres)
```

```
bwchirp = 100
```

### Input Arguments

#### **v** — Sensor velocity

positive real scalar | vector

Sensor velocity in meters per second, specified as a positive real scalar or vector.

Data Types: `double`

#### **azres** — Image azimuth or cross-range resolution

positive real scalar | vector

Image azimuth or cross-range resolution in meters, specified as a positive real scalar or a vector.

Data Types: `double`



## Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `'AzimuthBroadening',1.3,'ConeAngle',120`

### AzimuthBroadening — Azimuth impulse broadening factor

1 (default) | positive real scalar

Azimuth impulse broadening factor due to data weighting or windowing for sidelobe control, specified as a positive real scalar. This argument expresses the actual -3 dB mainlobe width with respect to the nominal width. Typical window functions like `hamming` and `hann` exhibit values in the range from 1 to 1.5.

Data Types: `double`

### ConeAngle — Doppler cone angle

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: `double`

## Output Arguments

### dbwch — Doppler bandwidth of single scatterer

matrix

Doppler bandwidth of single scatterer (chirped) in hertz, returned as a matrix. The rows of `dbwch` correspond to the velocity values in `v`. The columns of `dbwch` correspond to the azimuth resolution values in `azres`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

`matchinggain` | `sarazgain` | `sarchirprate` | `sarinttime` | `sarprf` | `sarscenedopbw`

### Introduced in R2021a

## sarscenedopbw

Doppler bandwidth of full scene after azimuth dechirping

### Syntax

```
bwdch = sarscenedopbw(r,lambda,v,wa)
bwdch = sarscenedopbw(r,lambda,v,wa,dcang)
```

### Description

`bwdch = sarscenedopbw(r,lambda,v,wa)` returns the Doppler bandwidth of the full scene after azimuth dechirping, corresponding to the composite signal received from all resolution cells within the scene.

`bwdch = sarscenedopbw(r,lambda,v,wa,dcang)` specifies the Doppler cone angle that identifies the direction towards the scene relative to the direction of motion of the array.

### Examples

#### Doppler Bandwidth of Full Scene

A side-looking airborne synthetic aperture radar (SAR) operates in broadside at a wavelength of 0.03 m with a sensor velocity of 100 m/s. The sensor illuminates a scatterer with a Doppler cone angle of 90° at a range of 10 km. The azimuth size of the scene is 916 m. Compute the Doppler bandwidth of the full scene after azimuth dechirping.

```
lambda = 0.03;
R = 10e3;
v = 100;
Wa = 916;
```

Compute the Doppler bandwidth.

```
bwdechirp = sarscenedopbw(R,lambda,v,Wa)
bwdechirp = 610.6667
```

### Input Arguments

#### **r** — Range from target to antenna

positive real scalar | vector

Range from target to antenna in meters, specified as a positive real scalar or a vector.

Data Types: double

#### **lambda** — Radar wavelength

positive real scalar | vector

Radar wavelength in meters, specified as a positive real scalar or a vector.

Data Types: `double`

### **v — Sensor velocity**

positive real scalar

Sensor velocity in meters per second, specified as a positive real scalar.

Data Types: `double`

### **wa — Azimuth size of scene**

positive real scalar

Azimuth size of scene in degrees, specified as a positive real scalar.

Data Types: `double`

### **dcang — Doppler cone angle**

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: `double`

## **Output Arguments**

### **bwdch — Doppler bandwidth of full scene**

matrix

Doppler bandwidth of full scene after azimuth dechirping in hertz, returned as a matrix. The rows of `bwdch` correspond to the range values in `r`. The columns of `bwdch` correspond to the wavelength values in `lambda`.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`matchinggain` | `sarazgain` | `sarchirprate` | `sarinttime` | `sarpointdopbw` | `sarprf`

**Introduced in R2021a**

## sarinttime

Synthetic aperture integration time

### Syntax

```
t = sarinttime(v,synlen)
t = sarinttime(r,lambda,v,azres)
t = sarinttime(r,lambda,v,azres,Name,Value)
```

### Description

`t = sarinttime(v,synlen)` returns the synthetic aperture integration time corresponding to a sensor velocity `v` and a synthetic aperture length `synlen`.

`t = sarinttime(r,lambda,v,azres)` returns the synthetic aperture integration time in terms of azimuth or cross-range resolution.

`t = sarinttime(r,lambda,v,azres,Name,Value)` specifies additional options using name-value arguments. Options include the azimuth impulse broadening factor and the Doppler cone angle.

### Examples

#### Synthetic Aperture Integration Time

A side-looking airborne synthetic aperture radar (SAR) operating in broadside at 10 GHz is travelling with a velocity of 100 m/s. The sensor illuminates the scatterer having cross-range resolution of 1 m and Doppler cone angle of 90 degrees for a target range of 10 Km. Compute the synthetic aperture integration time.

```
R = 10e3;
v = 100;
freq = 10e9;
azres = 1;
```

Compute the synthetic aperture time.

```
lambda = freq2wavelen(freq);
t = sarinttime(R,lambda,v,azres)

t = 1.4990
```

### Input Arguments

#### **v** — Sensor velocity

positive real scalar | vector

Sensor velocity in meters per second, specified as a positive real scalar or vector.

- If you specify `v` and `synLen` as input arguments, then `v` can be a scalar or a vector.
- If you specify `r`, `lambda`, `v`, and `azres` as input arguments, then `v` can only be a vector.

Data Types: double

### **synLen — Synthetic aperture length**

scalar | vector

Synthetic aperture length in meters, specified as a scalar or a vector.

Data Types: double

### **r — Range from target to antenna**

positive real scalar | vector

Range from target to antenna in meters, specified as a positive real scalar or a vector.

Data Types: double

### **lambda — Radar wavelength**

positive real scalar | vector

Radar wavelength in meters, specified as a positive real scalar or a vector.

Data Types: double

### **azres — Image azimuth or cross-range resolution**

positive real scalar

Image azimuth or cross-range resolution in meters, specified as a positive real scalar.

Data Types: double

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'AzimuthBroadening',1.2,'ConeAngle',60`

### **AzimuthBroadening — Azimuth impulse broadening factor**

1 (default) | positive real scalar

Azimuth impulse broadening factor due to data weighting or windowing for sidelobe control, specified as a positive real scalar. This argument expresses the actual -3 dB mainlobe width with respect to the nominal width. Typical window functions like `hamming` and `hann` exhibit values in the range from 1 to 1.5.

Data Types: double

### **ConeAngle — Doppler cone angle**

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: double

## Output Arguments

### **t** — Synthetic aperture integration time

matrix

Synthetic aperture integration time in seconds, returned as a matrix.

- If you specify `v` and `synLen` as input arguments, then the rows of `t` correspond to the velocity values in `v` and its columns correspond to the synthetic length values in `synLen`.
- If you specify `r`, `lambda`, `v`, and `azres` as input arguments, then the rows of `t` correspond to the range values in `r` and its columns correspond to the wavelength values in `lambda`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[matchinggain](#) | [sarazgain](#) | [sarchirprate](#) | [sarpointdopbw](#) | [sarprf](#) | [sarscenedopbw](#)

**Introduced in R2021a**

# sarprf

Synthetic aperture radar PRF

## Syntax

```
prf = sarprf(v,daz)
prf = sarprf(v,daz,Name,Value)
```

## Description

`prf = sarprf(v,daz)` computes the radar pulse repetition frequency (PRF) as a function of the sensor velocity and the antenna dimension in the azimuth direction.

`prf = sarprf(v,daz,Name,Value)` specifies additional options using name-value arguments.

## Examples

### SAR Pulse Repetition Frequency

A side-looking airborne SAR operating in broadside moves with a velocity of 100 m/s. The sensor has an aperture dimension of 1.5 m in azimuth. Compute the radar pulse repetition frequency. Assume an antenna roll-off factor of 1.2.

```
daz = 1.5;
v = 100;
ka = 1.2
```

```
ka = 1.2000
```

Compute the SAR pulse repetition frequency.

```
prf = sarprf(v,daz,'RollOff',ka)
```

```
prf = 160
```

## Input Arguments

### **v** — Sensor velocity

positive real scalar | vector

Sensor velocity in meters per second, specified as a positive real scalar or vector.

Data Types: double

### **daz** — Antenna width in azimuth direction

positive real scalar | vector

Antenna width in the azimuth direction in meters, specified as a positive real scalar or a vector.

Data Types: double

## Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `'RollOff',1.2,'ConeAngle',120`

### **RollOff** — Antenna roll-off factor

1 (default) | positive real scalar

Antenna roll-off factor, specified as a positive real scalar. This argument provides a safety factor that prevents mainlobe returns from aliasing in the pulse repetition frequency (PRF) time interval. Adjust the roll-off factor to make the PRF greater than the mainlobe Doppler bandwidth.

Data Types: `double`

### **ConeAngle** — Doppler cone angle

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: `double`

## Output Arguments

### **prf** — Radar pulse repetition frequency

matrix

Radar pulse repetition frequency in hertz, returned as a matrix. The rows of `prf` correspond to the velocity values in `v`. The columns of `prf` correspond to the antenna dimension values in `daz`.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`matchinggain` | `sarazgain` | `sarchirprate` | `sarinttime` | `sarpointdopbw` | `sarscenedopbw`

**Introduced in R2021a**



# sarmaxswath

Upper bound on swath length for SAR

## Syntax

```
swlenc = sarmaxswath(v,azres,grazang)
swlenc = sarmaxswath(v,azres,grazang,dcang)
```

## Description

`swlenc = sarmaxswath(v,azres,grazang)` computes the upper bound on swath length based on SAR constraints.

`swlenc = sarmaxswath(v,azres,grazang,dcang)` specifies the Doppler cone angle that identifies the direction towards the scene relative to the direction of motion of the array.

## Examples

### Swath Length Constraint

Estimate the constraint on swath length for a side-looking airborne SAR operating in broadside with a sensor velocity of 100 m/s. The radar has a cross-range resolution of 1.5 m and a nominal grazing angle of 30°.

```
v = 100;
azres = 1.5;
grazang = 30;
```

Compute the swath length constraint.

```
swlen = sarmaxswath(v,azres,grazang)
swlen = 2.5963e+06
```

## Input Arguments

### **v** — Sensor velocity

positive real scalar | vector

Sensor velocity in meters per second, specified as a positive real scalar or vector.

Data Types: `double`

### **azres** — Image azimuth or cross-range resolution

positive real scalar | vector

Image azimuth or cross-range resolution in meters, specified as a positive real scalar or a vector.

Data Types: `double`

**grazang — Grazing angle**

scalar in the range [0, 90]

Grazing angle in degrees, specified as a scalar in the range [0, 90].

Data Types: double

**dcang — Doppler cone angle**

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: double

**Output Arguments****swlenc — Upper bound on swath length**

matrix

Upper bound on swath length in meters, returned as a matrix. The rows of `swlenc` correspond to the velocity values in `v`. The columns of `swlenc` correspond to the azimuth resolution values in `azres`.**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

aperture2swath | sarmaxcovrate | sarminaperture | sarprfbounds | sarrange

**Introduced in R2021a**

# sarmaxcovrate

Upper bound on area coverage rate for SAR

## Syntax

```
acr = sarmaxcovrate(azres,grazang)
```

## Description

`acr = sarmaxcovrate(azres,grazang)` returns the upper bound on area coverage rate based on SAR constraints.

## Examples

### Area Coverage Rate Constraint

Estimate the constraint on area coverage rate of a side-looking airborne SAR. The radar has a cross-range resolution of 1.5 m and a nominal grazing angle of 30°.

```
azres = 1.5;  
grazang = 30;
```

Compute the area coverate rate constraint.

```
coverage = sarmaxcovrate(azres,grazang)  
coverage = 2.5963e+08
```

## Input Arguments

### azres — Image azimuth or cross-range resolution

positive real scalar | vector

Image azimuth or cross-range resolution in meters, specified as a positive real scalar or a vector.

Data Types: double

### grazang — Grazing angle

scalar in the range [0, 90] | vector

Grazing angle in degrees, specified as a scalar in the range [0, 90] or a vector.

Data Types: double

## Output Arguments

### acr — Upper bound on area coverage rate

matrix

Upper bound on area coverage rate in square meters per second, returned as a matrix. The rows of `acr` correspond to the azimuth resolution values in `azres`. The columns of `acr` correspond to the grazing angle values in `grazang`.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`aperture2swath` | `sarmaxswath` | `sarminaperture` | `sarprfbounds` | `sarrange`

**Introduced in R2021a**

# sarminaperture

Lower bound on antenna area for SAR

## Syntax

```
aac = sarminaperture(r,lambda,v,grazang)
aac = sarminaperture(r,lambda,v,grazang,dcang)
```

## Description

`aac = sarminaperture(r,lambda,v,grazang)` returns the lower bound on antenna area based on synthetic aperture radar (SAR) constraints.

`aac = sarminaperture(r,lambda,v,grazang,dcang)` specifies the Doppler cone angle that identifies the direction towards the scene relative to the direction of motion of the array.

## Examples

### Lower Bound on Antenna Area

Estimate the antenna area constraint of a side-looking airborne SAR operating in broadside at 16.7 GHz with a sensor velocity of 100 m/s for a target range of 10 km. Assume a nominal grazing angle of 30°.

```
fc = 16.7e9;
lambda = freq2wavelen(fc);
grazang =30;
v = 100;
R = 10e3;
```

Compute the antenna area constraint.

```
area = sarminaperture(R,lambda,v,grazang)
area = 4.1486e-04
```

## Input Arguments

### **r** — Range from target to antenna

positive real scalar | vector

Range from target to antenna in meters, specified as a positive real scalar or a vector.

Data Types: `double`

### **lambda** — Radar wavelength

positive real scalar | vector

Radar wavelength in meters, specified as a positive real scalar or a vector.

Data Types: double

**v — Sensor velocity**

positive real scalar

Sensor velocity in meters per second, specified as a positive real scalar.

Data Types: double

**grazang — Grazing angle**

scalar in the range [0, 90]

Grazing angle in degrees, specified as a scalar in the range [0, 90].

Data Types: double

**dcang — Doppler cone angle**

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: double

## Output Arguments

**aac — Upper bound on area coverage rate**

matrix

Upper bound on area coverage rate in square meters per second, returned as a matrix. The rows of `aac` correspond to the range values in `r`. The columns of `aac` correspond to the wavelength values in `lambda`.

## Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

`aperture2swath` | `sarmaxcovrate` | `sarmaxswath` | `sarprfbounds` | `sarrange`

**Introduced in R2021a**

# sarrange

Maximum unambiguous slant range of SAR

## Syntax

```
mur = sarrange(v,daz,df)
mur = sarrange(v,daz,df,Name,Value)
```

## Description

`mur = sarrange(v,daz,df)` returns the maximum unambiguous slant range of a synthetic aperture radar (SAR) system.

`mur = sarrange(v,daz,df,Name,Value)` specifies additional options using name-value arguments. Options include the Doppler cone angle and the antenna roll-off factor.

## Examples

### Maximum Unambiguous Slant Range

Estimate the maximum unambiguous range of a side-looking airborne synthetic aperture radar (SAR) operating in broadside with a sensor velocity varying from 20 m/s to 300 m/s. The SAR antenna has an aperture dimension of 3 m in the azimuth direction and a transmitter that works with a 5% duty cycle. Plot the resulting unambiguous range as a function of sensor velocity.

```
v = 20:10:300;
daz = 3;
d = 0.05;
```

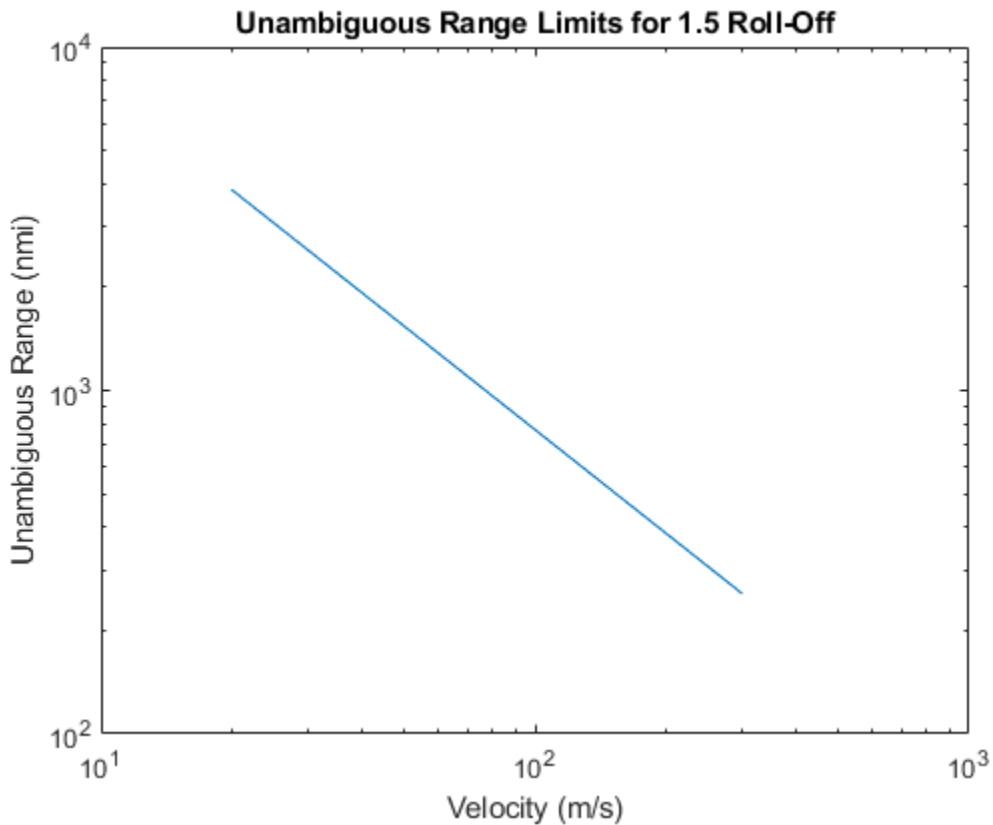
Compute the maximum unambiguous range in meters. Assume an antenna roll-off factor of 1.5. Convert the range to nautical miles.

```
Rmet = sarrange(v,daz,d,'RollOff',1.5);
Rnau = Rmet*0.00053996;
```

Plot the unambiguous range as a function of the sensor velocity.

```
loglog(v,Rnau)

axis([10 1000 100 10000])
xlabel('Velocity (m/s)')
ylabel('Unambiguous Range (nmi)')
title('Unambiguous Range Limits for 1.5 Roll-Off')
```



## Input Arguments

### **v** — Sensor velocity

positive real scalar | vector

Sensor velocity in meters per second, specified as a positive real scalar or vector.

Data Types: double

### **daz** — Antenna width in azimuth direction

positive real scalar

Antenna width in the azimuth direction in meters, specified as a positive real scalar.

Data Types: double

### **df** — Duty factor

positive real scalar in the range [0, 1] | vector

Duty factor, specified as a positive real scalar in the range [0, 1] or a vector. The duty factor is defined as the ratio of the pulse width to the pulse period.

Data Types: double



## Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `'RollOff',1.2,'ConeAngle',120`

### **ConeAngle** — Doppler cone angle

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: double

### **RollOff** — Antenna roll-off factor

1 (default) | positive real scalar

Antenna roll-off factor, specified as a positive real scalar. This argument provides a safety factor that prevents mainlobe returns from aliasing in the pulse repetition frequency (PRF) time interval. Adjust the roll-off factor to make the PRF greater than the mainlobe Doppler bandwidth.

Data Types: double

## Output Arguments

### **mur** — Maximum unambiguous range

matrix

Maximum unambiguous range, returned as a matrix. The rows of `mur` correspond to the velocity values in `v`. The columns of `mur` correspond to the duty factor values in `df`.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

`aperture2swath` | `sarmaxcovrate` | `sarmaxswath` | `sarminaperture` | `sarprfbounds`

**Introduced in R2021a**

## aperture2swath

Swath extent for radar on ground plane

### Syntax

```
[swlen,swwidth] = aperture2swath(r,lambda,d,grazang)
```

### Description

[swlen,swwidth] = aperture2swath(r,lambda,d,grazang) returns the swath length and width for a radar system at its maximum extent, assuming a flat Earth.

### Examples

#### Swath Length and Width

Estimate the maximum swath length and width of side-looking airborne synthetic aperture radar (SAR) operating at 16.7 GHz for a target range of 10 km. The radar has an aperture length of 3 m in the elevation dimension and of 4 m in the azimuth dimension. Assume a nominal grazing angle of 30°.

```
lambda = freq2wavelen(16.7e9);  
R = 10e3;
```

```
elaz = [3 4];
```

```
grazang = 30;
```

Compute the swath length and the swath width.

```
[swl,swwid] = aperture2swath(R,lambda,elaz,grazang)
```

```
swl = 119.6776
```

```
swwid = 44.8791
```

### Input Arguments

#### **r** — Range from target to antenna

positive real scalar | vector

Range from target to antenna in meters, specified as a positive real scalar or a vector.

Data Types: double

#### **lambda** — Radar wavelength

positive real scalar | vector

Radar wavelength in meters, specified as a positive real scalar or a vector.

Data Types: double

**d — Antenna dimensions**

positive real scalar | 1-by-2 row vector

Antenna dimensions in meters, specified as a positive real scalar or a 1-by-2 row vector.

- If you specify **d** as a two-element vector, the first element of **d** represents the antenna dimension in elevation and the second element represents the antenna dimension in azimuth.
- If you specify **d** as a scalar, `aperture2swath` assumes the antenna has equal elevation and azimuth dimensions.

Data Types: `double`

**grazang — Grazing angle**

scalar in the range [0, 90]

Grazing angle in degrees, specified as a scalar in the range [0, 90].

Data Types: `double`

**Output Arguments****swlen, swwidth — Swath length and width**

matrices

Swath length and width in meters, returned as matrices.

- The rows of the swath length `swlength` correspond to the range values in `r`. The columns of `swlength` correspond to the wavelength values in `lambda`.
- The rows of the swath width `swwidth` correspond to the range values in `r`. The columns of `swwidth` correspond to the wavelength values in `lambda`.

The swath width also corresponds to the azimuth or cross-range resolution of a real aperture antenna.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`sarmaxcovrate` | `sarmaxswath` | `sarminaperture` | `sarprfbounds` | `sarrange`

**Introduced in R2021a**

## sarprfbounds

Upper and lower bound on PRF for SAR

### Syntax

```
[prfmin,prfmax] = sarprfbounds(v,azres,swlen,grazang)
[prfmin,prfmax] = sarprfbounds(v,azres,swlen,grazang,Name,Value)
```

### Description

[prfmin,prfmax] = sarprfbounds(v,azres,swlen,grazang) returns the lower bound and the upper bound on the pulse repetition frequency (PRF) of a SAR system based on eclipsing constraints.

[prfmin,prfmax] = sarprfbounds(v,azres,swlen,grazang,Name,Value) specifies additional options using name-value arguments.

### Examples

#### PRF Constraint

Estimate the lower and upper PRF bounds due to eclipsing of a side-looking airborne SAR operating in broadside. The sensor has a velocity of 100 m/s. The transmitted waveform has a pulse width of 100 microseconds. The radar is grazing at an angle of 30° with an image azimuth resolution of 1.5 m and a swath length of 100 m.

```
v = 100;
pw = 100e-6;
grazang = 30;
azres = 1.5;
swl = 100;
```

Compute the PRF constraints.

```
[prfmin,prfmax] = sarprfbounds(v,azres,swl,grazang,'PulseWidth',pw)

prfmin = 66.6667
prfmax = 9.9426e+03
```

### Input Arguments

#### **v** — Sensor velocity

positive real scalar | vector

Sensor velocity in meters per second, specified as a positive real scalar or vector.

Data Types: double

**azres — Image azimuth or cross-range resolution**

positive real scalar | vector

Image azimuth or cross-range resolution in meters, specified as a positive real scalar or a vector.

Data Types: double

**swlen — Swath length**

positive scalar | vector

Swath length in meters, specified as a positive scalar or a vector.

Data Types: double

**grazang — Grazing angle**

scalar in the range [0, 90]

Grazing angle in degrees, specified as a scalar in the range [0, 90].

Data Types: double

**Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'ConeAngle',60,'PulseWidth',2e-6

**ConeAngle — Doppler cone angle**

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: double

**PulseWidth — Pulse width**

1e-6 (default) | positive real scalar

Pulse width in seconds, specified as a positive real scalar

Data Types: double

**Output Arguments****prfmin — PRF lower bound**

matrix

PRF lower bound in hertz, returned as a matrix. The rows of `prfmin` correspond to the velocity values in `v`. The columns of `prfmin` correspond to the resolution values in `azres`

**prfmax — PRF upper bound**

vector

PRF upper bound in hertz, returned as a vector of the same size as `swlen`.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

[aperture2swath](#) | [sarmaxcovrate](#) | [sarmaxswath](#) | [sarminaperture](#) | [sarrange](#)

**Introduced in R2021a**

# sarbeamwidth

Synthetic aperture azimuth beamwidth

## Syntax

```
synhpbw = sarbeamwidth(lambda,synlen)
synhpbw = sarbeamwidth(____,Name,Value)
[synhpbw,synfnbw] = sarbeamwidth(____)
```

## Description

`synhpbw = sarbeamwidth(lambda,synlen)` computes the half-power azimuth beamwidth synthesized by the coherent summation operation of the synthetic aperture radar (SAR).

`synhpbw = sarbeamwidth(____,Name,Value)` specifies additional options using name-value arguments. Options include the azimuth impulse broadening factor and the Doppler cone angle.

`[synhpbw,synfnbw] = sarbeamwidth(____)` also returns the first null azimuth beamwidth in the synthesized antenna pattern.

## Examples

### Half-Power and First Null Azimuth Beamwidths

Estimate the synthesized half-power beamwidth and the first null beamwidth of a side-looking airborne SAR operating in broadside at a wavelength of 0.05 m. The radar has a synthetic aperture length of 75 m and an azimuth impulse broadening factor of 0.9.

```
lambda = 0.05;
len = 75;
azb = 0.9;
```

Compute the synthetic aperture half-power and first null azimuth beamwidths.

```
[synhpbw,synfnbw] = sarbeamwidth(lambda,len,'AzimuthBroadening',azb)
```

```
synhpbw = 0.0172
```

```
synfnbw = 0.0191
```

## Input Arguments

### lambda — Radar wavelength

positive real scalar | vector

Radar wavelength in meters, specified as a positive real scalar or a vector.

Data Types: double

**synLen — Synthetic aperture length**

scalar | vector

Synthetic aperture length in meters, specified as a scalar or a vector.

Data Types: double

**Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'AzimuthBroadening',1.2,'CoherentIntegrationAngle',0.3

**AzimuthBroadening — Azimuth impulse broadening factor**

1 (default) | positive real scalar

Azimuth impulse broadening factor due to data weighting or windowing for sidelobe control, specified as a positive real scalar. This argument expresses the actual -3 dB mainlobe width with respect to the nominal width. Typical window functions like `hamming` and `hann` exhibit values in the range from 1 to 1.5.

Data Types: double

**ConeAngle — Doppler cone angle**

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: double

**CoherentIntegrationAngle — Coherent integration angle**

0.1 (default) | scalar in the range [0, 180]

Coherent integration angle in degrees, specified as a scalar in the range [0, 180]. This argument specifies the angle through which the target is viewed during the coherent processing aperture.

Data Types: double

**Output Arguments****synhpbw — Half-power azimuth beamwidth**

matrix

Half-power azimuth beamwidth in degrees, returned as a matrix. The rows of `synhpbw` correspond to the radar wavelength values in `lambda` and its columns correspond to the synthetic aperture length values in `synLen`.

**synfnbw — First null azimuth beamwidth**

matrix



First null azimuth beamwidth in degrees, returned as a matrix. The rows of `synfnbw` correspond to the radar wavelength values in `lambda`. The columns of `synfnbw` correspond to the synthetic aperture length values in `synlen`.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`sarbeamcompratio` | `sarlen`

**Introduced in R2021a**

# sarbeamcompratio

SAR beam compression ratio

## Syntax

```
bcr = sarbeamcompratio(r,lambda,synlen,wa)
bcr = sarbeamcompratio(r,lambda,synlen,wa,Name,Value)
```

## Description

`bcr = sarbeamcompratio(r,lambda,synlen,wa)` computes the beam compression ratio to illuminate a scene.

`bcr = sarbeamcompratio(r,lambda,synlen,wa,Name,Value)` specifies additional options using name-value arguments.

## Examples

### Beam Compression Ratio

Estimate the beam compression ratio of a side-looking airborne SAR operating in broadside at a wavelength of 0.05 m for a target range of 5 km. The radar has a synthetic aperture length of 75 m. The azimuth size of the scene is 50 m. Assume an azimuth impulse broadening factor of 1.3.

```
lambda = 0.05;
Wa = 50;
R = 5e3;
len = 75;
azb = 1.3;
```

Compute the beam compression ratio.

```
bcr = sarbeamcompratio(R,lambda,len,Wa,'AzimuthBroadening',azb)
bcr = 23.0769
```

## Input Arguments

### **r** — Range from target to antenna

positive real scalar | vector

Range from target to antenna in meters, specified as a positive real scalar or a vector.

Data Types: `double`

### **lambda** — Radar wavelength

positive real scalar | vector

Radar wavelength in meters, specified as a positive real scalar or a vector.

Data Types: double

### **synLen — Synthetic aperture length**

scalar

Synthetic aperture length in meters, specified as a scalar.

Data Types: double

### **wa — Azimuth size of scene**

positive real scalar

Azimuth size of scene in degrees, specified as a positive real scalar.

Data Types: double

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: 'AzimuthBroadening', 1.2, 'ConeAngle', 60

### **AzimuthBroadening — Azimuth impulse broadening factor**

1 (default) | positive real scalar

Azimuth impulse broadening factor due to data weighting or windowing for sidelobe control, specified as a positive real scalar. This argument expresses the actual -3 dB mainlobe width with respect to the nominal width. Typical window functions like `hamming` and `hann` exhibit values in the range from 1 to 1.5.

Data Types: double

### **ConeAngle — Doppler cone angle**

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: double

## **Output Arguments**

### **bcr — Beam compression ratio**

matrix

Beam compression ratio, returned as a matrix. The rows of `bcr` correspond to the range values in `r` and its columns correspond to the radar wavelength values in `lambda`.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

sarbeamwidth | sarlen

**Introduced in R2021a**

# sarlen

Synthetic aperture length

## Syntax

```
len = sarlen(v,t)
```

```
len = sarlen(r)  
len = sarlen(r,Name,Value)
```

```
len = sarlen(r,lambda,daz)  
len = sarlen(r,lambda,daz,Name,Value)
```

## Description

`len = sarlen(v,t)` returns the synthetic aperture length for a synthetic aperture radar given the sensor velocity and the synthetic aperture time.

`len = sarlen(r)` returns the synthetic aperture length for the spotlight mode.

`len = sarlen(r,Name,Value)` specifies additional options using the `ConeAngle` and `CoherentIntegrationAngle` name-value arguments.

`len = sarlen(r,lambda,daz)` returns the synthetic aperture length for the strip-map mode.

`len = sarlen(r,lambda,daz,Name,Value)` specifies additional options using the `ConeAngle` and `AzimuthBroadening` name-value arguments.

## Examples

### Synthetic Aperture Length

Estimate the synthetic aperture length of a side-looking airborne stripmap synthetic aperture radar (SAR) operating in broadside at a wavelength of 0.05 m for a target range of 10 km. The radar antenna has an aperture length of 3 m in the azimuth dimension and an azimuth impulse broadening factor of 1.3.

```
lambda = 0.05;  
Daz = 3;  
R = 10e3;  
azb = 1.3;
```

Compute the synthetic aperture length.

```
synlen = sarlen(R,lambda,Daz,'AzimuthBroadening',azb)
```

```
synlen = 216.6667
```

## Input Arguments

### **v — Sensor velocity**

positive real scalar | vector

Sensor velocity in meters per second, specified as a positive real scalar or vector.

Data Types: double

### **t — Synthetic aperture time**

positive real scalar | vector

Synthetic aperture time in seconds, specified as a positive real scalar or a vector.

Data Types: double

### **r — Range from target to antenna**

positive real scalar | vector

Range from target to antenna in meters, specified as a positive real scalar or a vector.

Data Types: double

### **lambda — Radar wavelength**

positive real scalar | vector

Radar wavelength in meters, specified as a positive real scalar or a vector.

Data Types: double

### **daz — Antenna width in azimuth direction**

positive real scalar | vector

Antenna width in the azimuth direction in meters, specified as a positive real scalar or a vector.

Data Types: double

## Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'AzimuthBroadening',1.3,'ConeAngle',120

### **AzimuthBroadening — Azimuth impulse broadening factor**

1 (default) | positive real scalar

Azimuth impulse broadening factor due to data weighting or windowing for sidelobe control, specified as a positive real scalar. This argument expresses the actual -3 dB mainlobe width with respect to the nominal width. Typical window functions like hamming and hann exhibit values in the range from 1 to 1.5.

Data Types: double

### **CoherentIntegrationAngle — Coherent integration angle**

0.1 (default) | scalar in the range [0, 180]

Coherent integration angle in degrees, specified as a scalar in the range [0, 180]. This argument specifies the angle through which the target is viewed during the coherent processing aperture.

Data Types: `double`

### **ConeAngle — Doppler cone angle**

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: `double`

## **Output Arguments**

### **len — Synthetic aperture length**

matrix

Synthetic aperture length, returned as a matrix.

- If you specify `v` and `t` as input arguments, then `len` is a matrix with rows corresponding to the velocity values in `v` and columns corresponding to the aperture time values in `t`.
- If you specify `r` as input for the spotlight mode, then `len` has the same dimensions as `r`.
- If you specify `r`, `lambda`, and `daz` as input for the strip-map mode, then `len` is a matrix with rows corresponding to the radar range values in `r` and columns corresponding to the antenna azimuth dimension in `daz`.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`sarbeamcompratio` | `sarbeamwidth`

**Introduced in R2021a**

## sarazres

Azimuth or cross-range resolution for SAR

### Syntax

```
azres = sarazres(r,lambda,synlen)
azres = sarazres( ___,Name,Value)
```

### Description

`azres = sarazres(r,lambda,synlen)` returns the azimuth or cross-range resolution for the synthetic aperture.

`azres = sarazres( ___,Name,Value)` specifies additional options using name-value arguments.

### Examples

#### Azimuth Resolution

Estimate the azimuth resolution of a side-looking airborne SAR operating in broadside at a wavelength of 0.03 m for a target range of 10 km. The radar has a synthetic aperture length of 195 m and a range impulse broadening factor of 1.3.

```
lambda = 0.03;
len = 195;
R = 10e3;
azb = 1.3;
```

Compute the azimuth resolution for the synthetic aperture.

```
synazres = sarazres(R,lambda,len,'AzimuthBroadening',azb)
synazres = 1.0000
```

### Input Arguments

#### **r** — Range from target to antenna

positive real scalar | vector

Range from target to antenna in meters, specified as a positive real scalar or a vector.

Data Types: `double`

#### **lambda** — Radar wavelength

positive real scalar | vector

Radar wavelength in meters, specified as a positive real scalar or a vector.

Data Types: `double`



**synLen — Synthetic aperture length**

scalar

Synthetic aperture length in meters, specified as a scalar.

Data Types: double

**Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'AzimuthBroadening',1.2,'CoherentIntegrationAngle',0.3`

**AzimuthBroadening — Azimuth impulse broadening factor**

1 (default) | positive real scalar

Azimuth impulse broadening factor due to data weighting or windowing for sidelobe control, specified as a positive real scalar. This argument expresses the actual -3 dB mainlobe width with respect to the nominal width. Typical window functions like `hamming` and `hann` exhibit values in the range from 1 to 1.5.

Data Types: double

**CoherentIntegrationAngle — Coherent integration angle**

0.1 (default) | scalar in the range [0, 180]

Coherent integration angle in degrees, specified as a scalar in the range [0, 180]. This argument specifies the angle through which the target is viewed during the coherent processing aperture.

Data Types: double

**ConeAngle — Doppler cone angle**

90 (default) | scalar in the range [0, 180]

Doppler cone angle in degrees, specified as a scalar in the range [0, 180]. This argument identifies the direction toward the scene relative to the direction of motion of the array.

Data Types: double

**Output Arguments****azres — Azimuth or cross-range resolution**

matrix

Azimuth or cross-range resolution in meters, returned as a matrix. The rows of `azres` correspond to the range values in `r` and its columns correspond to the wavelength values in `lambda`.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`grnd2slantrange` | `rainelres` | `slant2grndrange`

**Introduced in R2021a**

# rainelres

Elevation resolution of rain limited by radar resolution

## Syntax

```
elres = rainelres(r,beamw,grazang)
elres = rainelres(r,beamw,grazang, hgt)
```

## Description

`elres = rainelres(r,beamw,grazang)` returns the elevation resolution of rain limited by the resolution of the radar.

`elres = rainelres(r,beamw,grazang, hgt)` also specifies the height extent of rain.

## Examples

### Elevation Resolution of Rain

Estimate the elevation resolution of rain for a side-looking airborne synthetic aperture radar (SAR) with elevation beamwidth of  $9^\circ$  grazing at  $60^\circ$  for a target range of 10 km. Assume the height extent of rain to be 3 km.

```
elbw = 9;
grazang = 60;
```

```
rng = 10e3;
hrain = 3000;
```

Compute the rain elevation resolution.

```
elres = rainelres(rng,elbw,grazang,hrain)
```

```
elres = 782.1723
```

## Input Arguments

### **r** — Range from target to antenna

positive real scalar | vector

Range from target to antenna in meters, specified as a positive real scalar or a vector.

Data Types: `double`

### **beamw** — Elevation beamwidth

positive real scalar | vector

Elevation beamwidth in degrees, specified as a positive real scalar or a vector.

Data Types: double

**grazang — Grazing angle**

scalar in the range [0, 90]

Grazing angle in degrees, specified as a scalar in the range [0, 90].

Data Types: double

**hgt — Height extent of rain**

4000 (default) | real scalar

Height extent of rain in meters, specified as a real scalar.

Data Types: double

**Output Arguments****elres — Elevation resolution of rain**

matrix

Elevation resolution of rain, returned as a matrix. The rows of `elres` correspond to the range values in `r` and its columns correspond to the elevation beamwidth values in `beamw`.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`grnd2slanrange` | `sarazres` | `slant2grndrange`

**Introduced in R2021a**

# grnd2slanrange

Convert ground range projection to slant range

## Syntax

```
slrng = grnd2slanrange(grndrng,grazang)
```

## Description

`slrng = grnd2slanrange(grndrng,grazang)` returns the slant range `slrng` corresponding to the ground range projection `grndrng`.

## Examples

### Ground Range Projection to Slant Range

Determine the slant range given a 1000 m ground range and a grazing angle of 30°.

```
grndrng = 1000;  
grazang = 30;
```

Compute the slant range.

```
slantrng = grnd2slanrange(grndrng,grazang)
```

```
slantrng = 1.1547e+03
```

## Input Arguments

### **grndrng** — Ground range projection

scalar | vector

Ground range projection in meters, specified as a positive real scalar or vector.

Data Types: double

### **grazang** — Grazing angle

scalar in the range [0, 90]

Grazing angle in degrees, specified as a scalar in the range [0, 90].

Data Types: double

## Output Arguments

### **slrng** — Slant range

scalar | vector

Slant range in meters, returned as a positive real scalar or vector. `slrng` has the same dimensionality as `grndrng`.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

[rainelres](#) | [sarazres](#) | [slant2grndrange](#)

**Introduced in R2021a**

# grnd2slantrngres

Convert ground range resolution to slant range resolution

## Syntax

```
slrngres = grnd2slantrngres(grndrngres,grazang)
```

## Description

`slrngres = grnd2slantrngres(grndrngres,grazang)` returns the slant range resolution `slrngres` corresponding to the ground range resolution `grndrngres` and the grazing angle `grazang`.

## Examples

### Ground Range Resolution to Slant Range Resolution

Determine the slant range resolution given a ground range resolution of 1 m and a grazing angle of 30°.

```
grndrngres = 1;  
grazang = 30;
```

Compute the slant range resolution.

```
slrngres = grnd2slantrngres(grndrngres,grazang)
```

```
slrngres = 0.8660
```

## Input Arguments

### **grndrngres** — Ground range resolution

positive real scalar | vector

Ground range resolution in meters, specified as a positive real scalar or a vector.

Data Types: double

### **grazang** — Grazing angle

scalar in the range [0, 90] | vector

Grazing angle in degrees, specified as a scalar in the range [0, 90] or a vector.

Data Types: double

## Output Arguments

### **slrngres** — Slant range resolution

matrix

Slant range resolution in meters, returned as a matrix. The rows in `slrngres` correspond to the ground range resolutions in `grndrngres` and the columns correspond to the grazing angles in `grazang`.

## References

- [1] Doerry, Armin W. "Performance Limits for Synthetic Aperture Radar," 2nd Ed. Sandia National Laboratories, SAND2006-0821, February 2006.
- [2] Carrara, Walter G., Ron S. Goodman, and Ronald M. Majewski. *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*. The Artech House Remote Sensing Library. Boston, Artech House, 1995.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

`slant2grndrngres` | `sarazres`

**Introduced in R2021b**



# slant2grndrange

Convert slant range to ground range projection

## Syntax

```
grndrng = slant2grndrange(slrng,grazang)
```

## Description

`grndrng = slant2grndrange(slrng,grazang)` returns the ground range projection `grndrng` corresponding to the slant range `slrng` and grazing angle `grazang`.

## Examples

### Slant Range to Ground Range Projection

Determine the ground range projection given a slant range of 2000 m and a grazing angle of 30°.

```
slantrng = 2000;
grazang = 30;
```

Compute the ground range projection.

```
grndrng = slant2grndrange(slantrng,grazang)
grndrng = 1.7321e+03
```

### Ground Range Projection for Flat and Curved Earth

Compute the ground range projection for a target having a slant range of 1000 m from a sensor. The sensor is mounted on a platform that is 300 m above ground. Assume the Earth is flat.

```
gang = grazingang(300,1000); % Grazing angle
depang = gang; % Depression angle
grndrng = slant2grndrange(1000,gang)
```

```
grndrng = 953.9561
```

Repeat the computation, but now assume the Earth is curved.

```
Rearth = physconst('earthradius');
```

```
gangsph = grazingang(300,1000,'Curved',Rearth); % Grazing angle
depangsph = depressionang(300,1000,'Curved',Rearth); % Depression angle
tgtHeight = 0; % Smooth Earth
Re = effearthradius(1000,300,tgtHeight); % Effective Earth radius
grndrngcurved = Re*deg2rad(depangsph-gangsph)
```

```
grndrngcurved = 1.2344e+03
```

## Input Arguments

### **slrng** — Slant range

scalar | vector

Slant range in meters, specified as a positive real scalar or vector.

Data Types: double

### **grazang** — Grazing angle

scalar in the range [0, 90]

Grazing angle in degrees, specified as a scalar in the range [0, 90].

Data Types: double

## Output Arguments

### **grndrng** — Ground range projection

scalar | vector

Ground range projection in meters, returned as a positive real scalar or vector. `grndrng` has the same dimensionality as `slrng`.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`grnd2slanrange` | `rainelres` | `sarazres`

**Introduced in R2021a**

# slant2grndrngres

Convert slant range resolution to ground range resolution

## Syntax

```
grndrngres = slant2grndrngres(slrngres,grazang)
```

## Description

`grndrngres = slant2grndrngres(slrngres,grazang)` returns the ground range resolution `grndrngres` corresponding to the slant range resolution `slrngres` and the grazing angle `grazang`.

## Examples

### Slant Range Resolution to Ground Range Resolution

Determine the ground range resolution given a slant range resolution of 2 m and a grazing angle of 30°.

```
slrngres = 2;  
grazang = 30;
```

Compute the ground range resolution.

```
grndrngres = slant2grndrngres(slrngres,grazang)  
grndrngres = 2.3094
```

## Input Arguments

### **slrngres** — Slant range resolution

positive real scalar | vector

Slant range resolution in meters, specified as a positive real scalar or a vector.

Data Types: `double`

### **grazang** — Grazing angle

scalar in the range [0, 90] | vector

Grazing angle in degrees, specified as a scalar in the range [0, 90] or a vector.

Data Types: `double`

## Output Arguments

### **grndrngres** — Ground range resolution

matrix

Ground range resolution in meters, returned as a matrix. The rows in `grndrngres` correspond to the slant range resolutions in `slrngres` and the columns correspond to the grazing angles in `grazang`.

## References

- [1] Doerry, Armin W. "Performance Limits for Synthetic Aperture Radar," 2nd Ed. Sandia National Laboratories, SAND2006-0821, February 2006.
- [2] Carrara, Walter G., Ron S. Goodman, and Ronald M. Majewski. *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*. The Artech House Remote Sensing Library. Boston, Artech House, 1995.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`sarazres` | `grnd2slantrngres`

**Introduced in R2021b**

# sarSurfaceRCS

Radar cross-section of target for SAR

## Syntax

```
rCS = sarSurfaceRCS(sigmaref,freq,freqref, rngazres, grazang)
rCS = sarSurfaceRCS(sigmaref,freq,freqref, rngazres, grazang, n)
rCS = sarSurfaceRCS(nrCS, rngazres, grazang)
```

## Description

`rCS = sarSurfaceRCS(sigmaref, freq, freqref, rngazres, grazang)` returns the target radar cross-section (RCS) for SAR as projected on the ground.

`rCS = sarSurfaceRCS(sigmaref, freq, freqref, rngazres, grazang, n)` specifies a frequency-dependent proportionality factor that depends upon the target characteristics.

`rCS = sarSurfaceRCS(nrCS, rngazres, grazang)` uses as input the surface normalized radar cross-section, also known as the reflectivity or  $\sigma^0$ .

## Examples

### Target Radar Cross-Section

Estimate the target radar cross-section (RCS) of a side-looking airborne SAR operating at frequencies between 16 GHz to 17 GHz and grazing at 30°. The target reflectivity is -25 dB at the Ku band (nominally 16.7 GHz). The radar has a slant range resolution of 15 m and an azimuth resolution of 18 m. Assume a frequency-dependent proportionality factor of 1.

```
f = 16e9:1e7:17e9;
```

```
sigmaref = -25;
fref = 16.7e9;
```

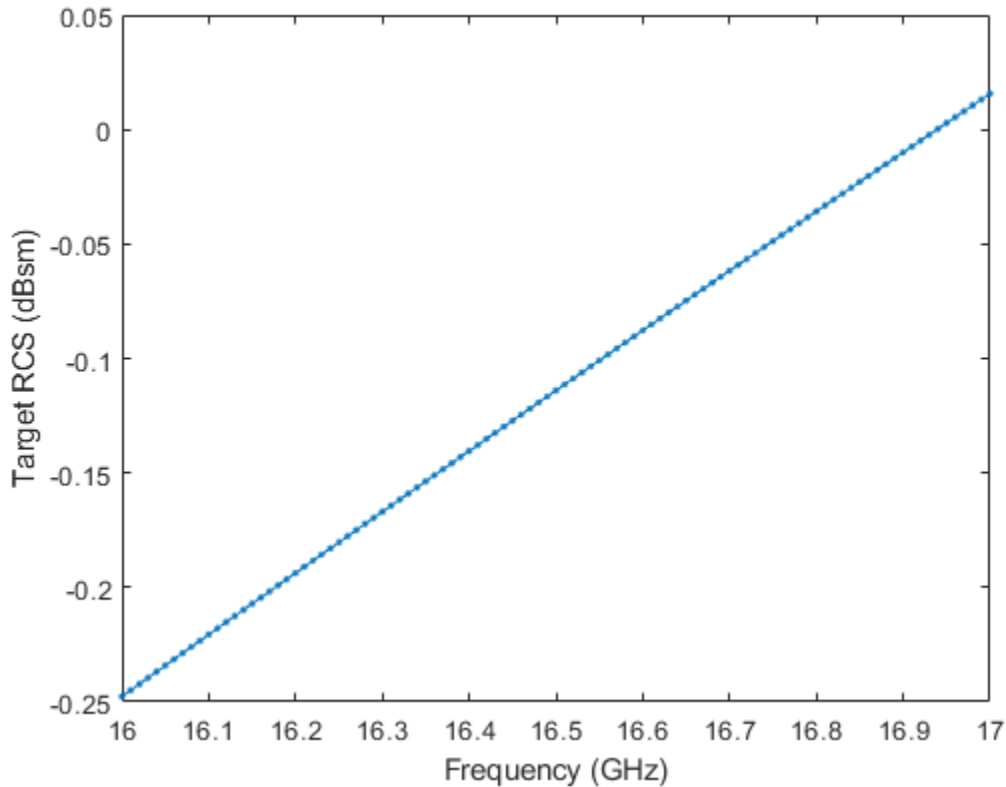
```
rngazres = [15 18];
grazang = 30;
```

Convert the reflectivity to linear units. Compute the target RCS.

```
sigma = sarSurfaceRCS(db2pow(sigmaref), f, fref, rngazres, grazang);
```

Plot the RCS in decibels as a function of frequency.

```
plot(f/1e9, pow2db(sigma), '-.')
xlabel('Frequency (GHz)')
ylabel('Target RCS (dBsm)')
```



## Input Arguments

### **sigmaref** — Reflectivity at nominal reference frequency

positive real scalar

Reflectivity at nominal reference frequency in square meters per square meter, specified as a positive real scalar.

Data Types: double

### **freq** — Radar frequency

positive real scalar | vector

Radar frequency in hertz, specified as a positive real scalar or a vector.

Data Types: double

### **freqref** — Nominal reference frequency

positive real scalar

Nominal reference frequency in hertz, specified as a positive real scalar.

Data Types: double

### **rngazres** — Slant range and azimuth resolutions

1-by-2 row vector of positive real scalars

Slant range and azimuth resolutions, specified as a 1-by-2 row vector of positive real scalars.

- The first element of `rngazres` specifies the slant range resolution in meters.
- The second element of `rngazres` specifies the azimuth or cross-range resolution in meters.

Data Types: `double`

### **grazang — Grazing angle**

scalar in the range [0, 90]

Grazing angle in degrees, specified as a scalar in the range [0, 90].

Data Types: `double`

### **n — Frequency-dependent proportionality factor**

1 (default) | positive real scalar

Frequency-dependent proportionality factor, specified as a real scalar. For distributed targets, `n` varies between 0 and 1. For nondistributed targets, `n` is a positive real scalar.

Data Types: `double`

### **nracs — Surface normalized radar cross-section**

nonnegative scalar | row vector

Surface normalized radar cross-section in square meters per square meter, specified as a nonnegative scalar or row vector. The surface normalized radar cross-section is also known as the reflectivity or  $\sigma^0$ .

Data Types: `double`

## **Output Arguments**

### **racs — Target radar cross-section**

scalar | vector

Target radar cross-section for SAR as projected on the ground in square meters, returned as a scalar or a vector. `racs` has the same dimensions as either `freq` or `nracs`.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`clutterVolumeRCS` | `rainreflectivity`

**Introduced in R2021a**

## clutterVolumeRCS

Radar cross-section of volume clutter

### Syntax

```
rsc = clutterVolumeRCS(volrefl,vol)
```

### Description

`rsc = clutterVolumeRCS(volrefl,vol)` returns the radar cross-section (RCS) of volume clutter defined by the resolution of the radar.

### Examples

#### Radar Cross-Section of Rain

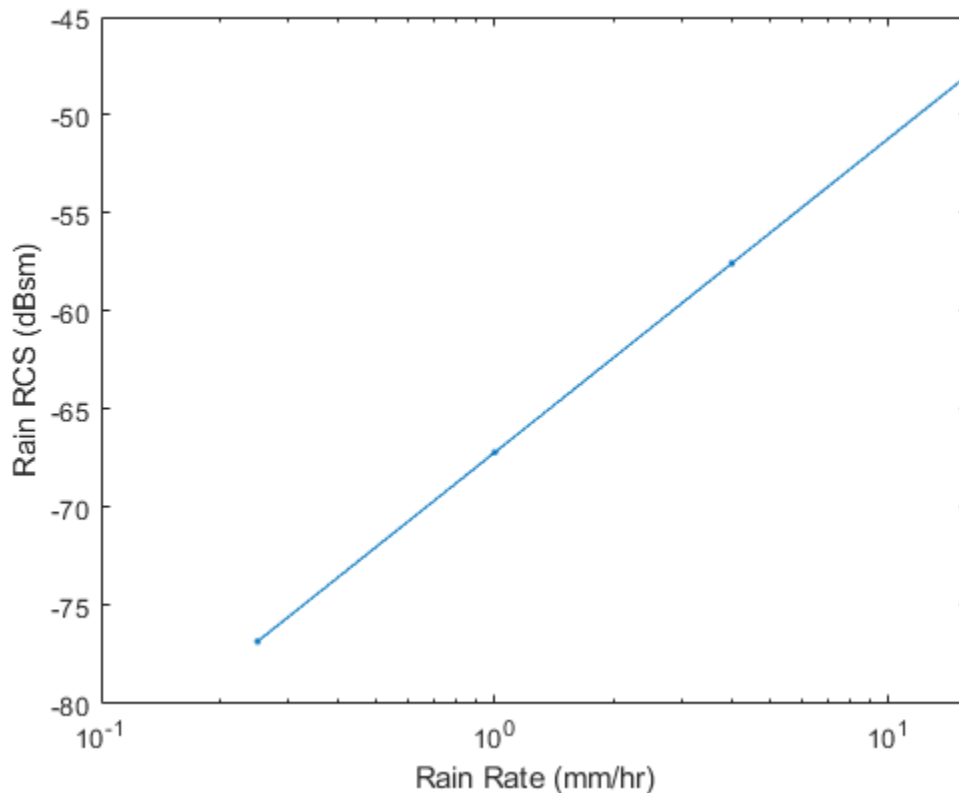
Estimate the radar cross-section of rain for a side-looking airborne SAR operating in the L band at 1.5 GHz. The rain is specified by a range resolution of 15 m, an azimuth resolution of 18 m, and a rain elevation cell resolution of 20 m. The rain rates are 0.25 mm/hr, 1 mm/hr, 4 mm/hr, and 16 mm/hr.

```
f = 1.5e9;  
  
rngres = 15;  
azres = 18;  
elres = 20;  
res = [rngres azres elres];  
  
rr = [0.25 1 4 16];  
  
Compute the rain radar cross-section. Use rainreflectivity to compute the volume reflectivity of  
the scattering particles.  
  
volref = rainreflectivity(f,rr);  
  
rsc = clutterVolumeRCS(volref,res);
```

Plot the rain radar cross-section as a function of the rain rate. Express the cross-section in dB.

```
semilogx(rr,pow2db(rsc),'.-')  
xlabel('Rain Rate (mm/hr)')  
ylabel('Rain RCS (dBsm)')
```





## Input Arguments

### **volrefl** – Volume reflectivity of scattering particles

real scalar | vector

Volume reflectivity of scattering particles in square meters per cubic meter, specified as a scalar or a vector.

Data Types: double

### **vol** – Clutter extent

positive real scalar | 1-by-3 row vector

Clutter extent, specified as a positive real scalar or a 1-by-3 row vector.

- If specified as a positive real scalar, **vol** represents the volume of the clutter in cubic meters.
- If specified as a 1-by-3 row vector:
  - The first element of **vol** is a positive real scalar that represents the clutter within the range resolution in meters of the radar.
  - The second element of **vol** is a positive real scalar that represents the clutter within the azimuth (or cross-range) resolution in meters of the radar.
  - The third element of **vol** is a positive real scalar that represents the clutter within the elevation resolution in meters of the radar.

Data Types: double

## Output Arguments

### **r<sub>cs</sub>** — Radar cross-section of volume clutter

scalar | vector

Radar cross-section of volume clutter in square meters, returned as a scalar or a vector. `rcs` has the same dimensions as `vol_refl`.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`rainreflectivity` | `sarsurfacercs`

**Introduced in R2021a**

# rainreflectivity

Volume reflectivity of rain

## Syntax

```
volrefl = rainreflectivity(freq,rr)  
volrefl = rainreflectivity(freq,rr,pol)
```

## Description

`volrefl = rainreflectivity(freq,rr)` returns the volume reflectivity of rain, computed using the “Marshall-Palmer Model” on page 1-345.

`volrefl = rainreflectivity(freq,rr,pol)` specifies the polarization of the transmitted and received waves.

## Examples

### Rain Volume Reflectivity

Estimate the rain volume reflectivity of a side-looking airborne SAR operating in the L band at 1.5 GHz for rain rates of 0.25 mm/hr, 1 mm/hr, 4 mm/hr, and 16 mm/Hr.

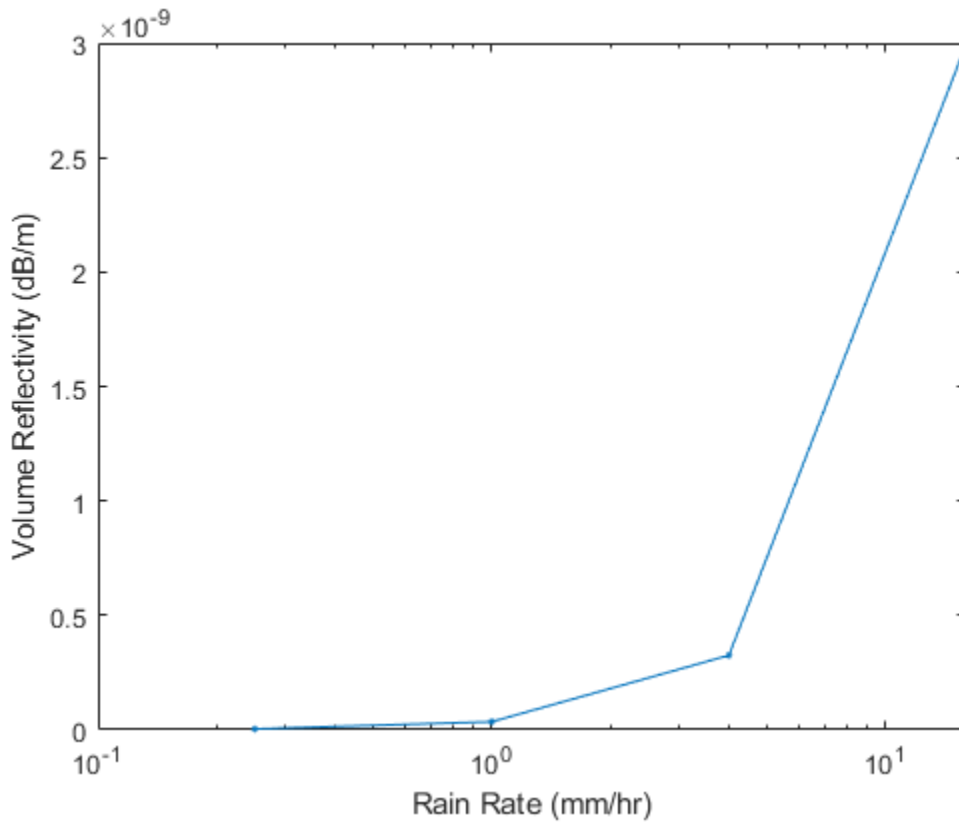
```
f = 1.5e9;  
rr = [0.25 1 4 16];
```

Compute the rain volume reflectivity.

```
volref = rainreflectivity(f,rr);
```

Plot the rain volume reflectivity as a function of the rain rate.

```
semilogx(rr,volref,'.-')  
xlabel('Rain Rate (mm/hr)')  
ylabel('Volume Reflectivity (dB/m)')
```



## Input Arguments

### freq — Radar frequency

positive real scalar | vector

Radar frequency in hertz, specified as a positive real scalar or a vector.

Data Types: double

### rr — Rain rate

real scalar | vector

Rain rate in millimeters per hour, specified as a real scalar or a vector.

Data Types: double

### pol — Polarization of transmitted and received waves

'HH' (default) | 'HV' | 'VV' | 'VH' | 'RCPRCP' | 'RCPLCP' | 'LCPLCP' | 'LR' | 'HRCP' | 'VLCP' | 'RCPV' | 'LCPH'

Polarization of transmitted and received waves, specified as one of these.

Value	Transmitted Wave	Received Wave
'HH'	Horizontal polarization	Horizontal polarization

Value	Transmitted Wave	Received Wave
'HV'	Horizontal polarization	Vertical polarization
'VV'	Vertical polarization	Vertical polarization
'VH'	Vertical polarization	Horizontal polarization
'RCPRCP'	Right-hand circular polarization	Right-hand circular polarization
'RCPLCP'	Right-hand circular polarization	Left-hand circular polarization
'LCPLCP'	Left-hand circular polarization	Left-hand circular polarization
'LR'	Left-hand polarization	Right-hand polarization
'HRCP'	Horizontal polarization	Right-hand circular polarization
'VLCP'	Vertical polarization	Left-hand circular polarization
'RCPV'	Right-hand circular polarization	Vertical polarization
'LCPH'	Left-hand circular polarization	Horizontal polarization

Data Types: char | string

## Output Arguments

### **volrefl** — Volume reflectivity of rain

matrix

Volume reflectivity (radar cross-section per unit volume) of rain in square meters per cubic meter, returned as a matrix. The rows of `volref` correspond to the radar frequency values in `f req`. The columns of `volref` correspond to the rain rate values in `rr`.

## More About

### Marshall-Palmer Model

The rain clutter reflectivity is computed based on the commonly used Marshall-Palmer drop-size distribution model. The model assumes raindrops are generally small with respect to the wavelength and are nearly spherical, indicating Rayleigh scattering.

The Marshall-Palmer model matches experimental results with measured data up to the Ka-band. Additionally, rain is not a static target, and exhibits its own motion spectrum. The motion spectrum is typically centered at some velocity with a recognizable velocity bandwidth. Data suggests a velocity bandwidth sometimes as high as 8 m/s, with a median velocity bandwidth of about 4 m/s.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

`clutterVolumeRCS` | `sarsurfacercs`

Introduced in R2021a

## radareqsarsnr

Signal-to-noise ratio of SAR image

### Syntax

```
imgsnr = radareqsarsnr(r,lambda,pt,tau,rnggain,azgain)
imgsnr = radareqsarsnr(r,lambda,pt,tau,rnggain,azgain,Name,Value)
```

### Description

`imgsnr = radareqsarsnr(r,lambda,pt,tau,rnggain,azgain)` returns the SAR image signal-to-noise ratio (SNR).

`imgsnr = radareqsarsnr(r,lambda,pt,tau,rnggain,azgain,Name,Value)` specifies additional options using name-value arguments.

### Examples

#### SAR Image SNR

Estimate the image SNR for a SAR operating in broadside at a frequency of 5.3 GHz and 5 kW peak power to form an image of a target at 50 km. Assume an RCS of 1 m<sup>2</sup> and rectangular waveform with a bandwidth of 0.05 microseconds. The range processing gain is 29.8 dB and the azimuth processing gain is 42.7 dB. Assume no losses.

```
lambda = freq2wavelen(5.3e9);
pt = 5e3;
r = 50e3;
```

```
tau = 0.05e-6;
```

```
rnggain = 29.8;
azgain = 42.7;
```

Compute the image SNR.

```
snr = radareqsarsnr(r,lambda,pt,tau,rnggain,azgain)
```

```
snr = 34.5704
```

### Input Arguments

#### **r** — Range to target

scalar | column vector | 1-by-2 row vector | 2-column matrix

Range to target in meters, specified as a scalar, a column vector, a 1-by-2 row vector, or a 2-column matrix.

- Specify this argument as a scalar or a column vector for a monostatic radar.
- Specify this argument as a 1-by-2 row vector or as a 2-column matrix for a bistatic radar.
  - The first element or column corresponds to the range from the transmitter to the target.
  - The second element or column corresponds to the range from the target to the receiver.

Data Types: double

### **lambda — Wavelength of radar operating frequency**

positive real scalar

Wavelength of radar operating frequency in meters, specified as a positive real scalar.

Data Types: double

### **pt — Transmitter peak signal power**

positive real scalar | vector

Transmitter peak signal power in watts, specified as a positive real scalar or a vector.

Data Types: double

### **tau — Pulse width at antenna port**

positive real scalar

Pulse width at the antenna port in seconds, specified as a positive real scalar.

Data Types: double

### **rnggain — SNR gain due to range processing**

real scalar

SNR gain due to range processing in decibels, specified as a real scalar.

Data Types: double

### **azgain — SNR gain due to azimuth processing**

real scalar

SNR gain due to azimuth processing in decibels, specified as a real scalar.

Data Types: double

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'Ts',293,'Gain',12`

### **RCS — Target radar cross-section**

1 (default) | scalar | vector

Target radar cross-section in square meters, specified as a scalar or a vector. `radareqsarsnr` assumes a nonfluctuating target (Swerling case 0).

Data Types: double

**Ts — System noise temperature**

290 (default) | positive scalar

System noise temperature in kelvins, specified as a positive scalar.

Data Types: double

**Gain — Antenna gain**

20 (default) | scalar | 1-by-2 row vector

Antenna gain in decibels, specified as a scalar or 1-by-2 row vector.

- If you specify this argument as a two-element vector, the first element represents antenna transmit gain and the second element represents the antenna receive gain.
- If you specify this argument as a scalar, `radareqsarsnr` assumes the antenna has equal transmit and receive gains.

Data Types: double

**Loss — System loss**

0 (default) | scalar | vector

System loss in decibels, specified as a scalar or a vector.

Data Types: double

**AtmosphericLoss — Atmospheric absorption loss**

0 (default) | scalar | column vector | 1-by-2 row vector | 2-column matrix

Atmospheric absorption loss in decibels, specified as a scalar, a column vector, a 1-by-2 row vector, or a 2-column matrix.

- Specify this argument as a scalar or a column vector to represent the atmospheric absorption loss for a one-way path.
- Specify this argument as a 1-by-2 row vector or as a 2-column matrix to represent a transmit path and a receive path.
  - The first element or column corresponds to the atmospheric absorption loss for the transmit path.
  - The second element or column corresponds to the atmospheric absorption loss for the receive path.

Data Types: double

**PropagationFactor — Propagation factor**

0 (default) | scalar | column vector | 1-by-2 row vector | 2-column matrix

Propagation factor in decibels, specified as a scalar, a column vector, a 1-by-2 row vector, or a 2-column matrix.

- Specify this argument as a scalar or a column vector to represent the propagation factor loss for a one-way path.
- Specify this argument as a 1-by-2 row vector or as a 2-column matrix to represent a transmit path and a receive path.



- The first element or column corresponds to the propagation factor for the transmit path.
- The second element or column corresponds to the propagation factor for the receive path.

Data Types: double

### **CustomFactor — Custom factor**

0 (default) | scalar | vector

Custom factor in decibels, specified as a scalar or a vector. This argument contributes to the received signal energy and can include other factors.

Data Types: double

## **Output Arguments**

### **imgsnr — SAR image signal-to-noise ratio**

column vector

SAR image signal-to-noise ratio in decibels, returned as a column vector.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

radareqsarpow | radareqsarrng | rainscr | sarnoiserefl

**Introduced in R2021a**

## radareqsarpow

Minimum peak transmit power using SAR equation

### Syntax

```
pt = radareqsarpow(r,lambda,snr,tau,rnggain,azgain)
pt = radareqsarpow(r,lambda,snr,tau,rnggain,azgain,Name,Value)
```

### Description

`pt = radareqsarpow(r,lambda,snr,tau,rnggain,azgain)` returns the SAR peak transmit power.

`pt = radareqsarpow(r,lambda,snr,tau,rnggain,azgain,Name,Value)` specifies additional options using name-value arguments.

### Examples

#### SAR Peak Transmit Power

Estimate the peak transmit power for a side-looking SAR operating at a frequency of 5.3 GHz to form an image of a target at 50 km. Assume a radar cross-section (RCS) of 1 m<sup>2</sup> and rectangular waveform with a bandwidth of 0.05 microseconds. The antenna gain is 30 dB and the minimum SNR required to make a detection is 30 dB. The range processing gain is 29.8 dB and the azimuth processing gain is 42.7 dB. Assume zero losses.

```
lambda = freq2wavelen(5.3e9);
r = 50e3;
```

```
tau = 0.05e-6;
```

```
G = 30;
SNR = 30;
rnggain = 29.8;
azgain = 42.7;
```

Compute the peak transmit power.

```
pt = radareqsarpow(r,lambda,SNR,tau,rnggain,azgain,'Gain',G)
```

```
pt = 17.4555
```

### Input Arguments

#### **r** — Range to target

scalar | column vector | 1-by-2 row vector | 2-column matrix

Range to target in meters, specified as a scalar, a column vector, a 1-by-2 row vector, or a 2-column matrix.

- Specify this argument as a scalar or a column vector for a monostatic radar.
- Specify this argument as a 1-by-2 row vector or as a 2-column matrix for a bistatic radar.
  - The first element or column corresponds to the range from the transmitter to the target.
  - The second element or column corresponds to the range from the target to the receiver.

Data Types: double

### **lambda — Wavelength of radar operating frequency**

positive real scalar

Wavelength of radar operating frequency in meters, specified as a positive real scalar.

Data Types: double

### **snr — Required signal-to-noise ratio**

real scalar | vector

Required signal-to-noise ratio (SNR) in decibels, specified as a real scalar or a vector.

Data Types: double

### **tau — Pulse width at antenna port**

positive real scalar

Pulse width at the antenna port in seconds, specified as a positive real scalar.

Data Types: double

### **rnggain — SNR gain due to range processing**

real scalar

SNR gain due to range processing in decibels, specified as a real scalar.

Data Types: double

### **azgain — SNR gain due to azimuth processing**

real scalar

SNR gain due to azimuth processing in decibels, specified as a real scalar.

Data Types: double

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'Ts',293,'Gain',12`

### **RCS — Target radar cross-section**

1 (default) | scalar | vector

Target radar cross-section in square meters, specified as a scalar or a vector. `radareqsarpow` assumes a nonfluctuating target (Swerling case 0).

Data Types: double

**Ts — System noise temperature**

290 (default) | positive scalar

System noise temperature in kelvins, specified as a positive scalar.

Data Types: double

**Gain — Antenna gain**

20 (default) | scalar | 1-by-2 row vector

Antenna gain in decibels, specified as a scalar or 1-by-2 row vector.

- If you specify this argument as a two-element vector, the first element represents antenna transmit gain and the second element represents the antenna receive gain.
- If you specify this argument as a scalar, `radareqsarpow` assumes the antenna has equal transmit and receive gains.

Data Types: double

**Loss — System loss**

0 (default) | scalar | vector

System loss in decibels, specified as a scalar or a vector.

Data Types: double

**AtmosphericLoss — Atmospheric absorption loss**

0 (default) | scalar | column vector | 1-by-2 row vector | 2-column matrix

Atmospheric absorption loss in decibels, specified as a scalar, a column vector, a 1-by-2 row vector, or a 2-column matrix.

- Specify this argument as a scalar or a column vector to represent the atmospheric absorption loss for a one-way path.
- Specify this argument as a 1-by-2 row vector or as a 2-column matrix to represent a transmit path and a receive path.
  - The first element or column corresponds to the atmospheric absorption loss for the transmit path.
  - The second element or column corresponds to the atmospheric absorption loss for the receive path.

Data Types: double

**PropagationFactor — Propagation factor**

0 (default) | scalar | column vector | 1-by-2 row vector | 2-column matrix

Propagation factor in decibels, specified as a scalar, a column vector, a 1-by-2 row vector, or a 2-column matrix.

- Specify this argument as a scalar or a column vector to represent the propagation factor loss for a one-way path.
- Specify this argument as a 1-by-2 row vector or as a 2-column matrix to represent a transmit path and a receive path.

- The first element or column corresponds to the propagation factor for the transmit path.
- The second element or column corresponds to the propagation factor for the receive path.

Data Types: double

### **CustomFactor — Custom factor**

0 (default) | scalar | vector

Custom factor in decibels, specified as a scalar or a vector. This argument contributes to the received signal energy and can include other factors.

Data Types: double

## **Output Arguments**

### **pt — SAR peak transmit power**

vector

SAR peak transmit power in watts, returned as a vector.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

radareqsarrng | radareqsarsnr | rainscr | sarnoiserefl

**Introduced in R2021a**

## radareqsarrng

Maximum detectable range using SAR equation

### Syntax

```
rng = radareqsarrng(lambda,snr,pt,tau,rnggain,azgain)
rng = radareqsarrng(lambda,snr,pt,tau,rnggain,azgain,Name,Value)
```

### Description

`rng = radareqsarrng(lambda,snr,pt,tau,rnggain,azgain)` returns the maximum detectable range for a SAR.

`rng = radareqsarrng(lambda,snr,pt,tau,rnggain,azgain,Name,Value)` specifies additional options using name-value arguments.

### Examples

#### Maximum Detectable Range

Estimate the range for a side-looking SAR imaging a target with a radar cross-section (RCS) of 1 m<sup>2</sup>. The radar operates at a frequency of 5.3 GHz and has a peak power of 5 kW. The SAR uses a rectangular waveform with a pulse width of 0.05 microseconds. The antenna gain is 30 dB and the minimum detectable SNR is 30 dB. The range processing gain is 29.8 dB and the azimuth processing gain is 42.7 dB. Assume zero losses.

```
lambda = freq2wavelen(5.3e9);
pt = 5e3;
```

```
tau = 0.05e-6;
```

```
gain = 30;
SNR = 30;
```

```
rnggain = 29.8;
azgain = 42.7;
```

Compute the maximum detectable range. Express the result in kilometers.

```
rng = radareqsarrng(lambda,SNR,pt,tau,rnggain, azgain,'Gain', gain,'UnitStr','km')
rng = 205.6978
```

### Input Arguments

#### lambda — Wavelength of radar operating frequency

positive real scalar

Wavelength of radar operating frequency in meters, specified as a positive real scalar.

Data Types: double

**snr — Required signal-to-noise ratio**

real scalar | vector

Required signal-to-noise ratio (SNR) in decibels, specified as a real scalar or a vector.

Data Types: double

**pt — Transmitter peak signal power**

positive real scalar | vector

Transmitter peak signal power in watts, specified as a positive real scalar or a vector.

Data Types: double

**tau — Pulse width at antenna port**

positive real scalar

Pulse width at the antenna port in seconds, specified as a positive real scalar.

Data Types: double

**rnggain — SNR gain due to range processing**

real scalar

SNR gain due to range processing in decibels, specified as a real scalar.

Data Types: double

**azgain — SNR gain due to azimuth processing**

real scalar

SNR gain due to azimuth processing in decibels, specified as a real scalar.

Data Types: double

**Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'Ts',293,'Gain',12`

**RCS — Target radar cross-section**

1 (default) | scalar | vector

Target radar cross-section in square meters, specified as a scalar or a vector. `radareqsarrng` assumes a nonfluctuating target (Swerling case 0).

Data Types: double

**Ts — System noise temperature**

290 (default) | positive scalar

System noise temperature in kelvins, specified as a positive scalar.

Data Types: double

**Gain — Antenna gain**

20 (default) | scalar | 1-by-2 row vector

Antenna gain in decibels, specified as a scalar or 1-by-2 row vector.

- If you specify this argument as a two-element vector, the first element represents antenna transmit gain and the second element represents the antenna receive gain.
- If you specify this argument as a scalar, `radareqsarrng` assumes the antenna has equal transmit and receive gains.

Data Types: double

**Loss — System loss**

0 (default) | scalar | vector

System loss in decibels, specified as a scalar or a vector.

Data Types: double

**CustomFactor — Custom factor**

0 (default) | scalar | vector

Custom factor in decibels, specified as a scalar or a vector. This argument contributes to the received signal energy and can include other factors.

Data Types: double

**UnitStr — Unit of range length**

'm' (default) | 'km' | 'mi' | 'nmi'

Unit of range length, specified as 'm' (meter), 'km' (kilometer), 'mi' (statute mile), or 'nmi' (nautical mile).

Data Types: char | string

**Output Arguments****rng — Maximum detectable range**

column vector

Maximum detectable range, returned as a column vector expressed in the units specified using `UnitStr`. For bistatic radars, each element of `rng` is the geometric mean of the range from the transmitter to the target and the range from the target to the receiver.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`radareqsarpow` | `radareqsarsnr` | `rainscr` | `sarnoiserefl`



**Introduced in R2021a**

## sarnoiserefl

Noise equivalent reflectivity of SAR

### Syntax

```
neq = sarnoiserefl(freq,freqref,imgsnr,sigma_ref)
neq = sarnoiserefl(freq,freqref,imgsnr,sigma_ref,n)
```

### Description

`neq = sarnoiserefl(freq,freqref,imgsnr,sigma_ref)` computes the noise equivalent reflectivity.

`neq = sarnoiserefl(freq,freqref,imgsnr,sigma_ref,n)` specifies a frequency-dependent proportionality factor that depends upon the target characteristics.

### Examples

#### Noise Equivalent Reflectivity

Estimate the noise equivalent reflectivity of a side-looking SAR operating at a frequency of 16 GHz for a target reflectivity of -25 dB at the Ku band (nominally 16.7 GHz) and to form an image having an SNR of 30 dB.

```
f = 16e9;
sigma_ref = -25;
fref = 16.7e9;
snr = 30;
```

Convert the target reflectivity to linear units. Compute the noise equivalent reflectivity.

```
neq = sarnoiserefl(f,fref,snr,db2pow(sigma_ref))
neq = -55.1860
```

### Input Arguments

#### **freq** — Radar frequency

positive real scalar | vector

Radar frequency in hertz, specified as a positive real scalar or a vector.

Data Types: `double`

#### **freqref** — Nominal reference frequency

positive real scalar

Nominal reference frequency in hertz, specified as a positive real scalar.

Data Types: `double`

**imgsnr — Image signal-to-noise ratio**

real scalar | vector

Image signal-to-noise ratio (SNR) of the SAR in decibels, specified as a real scalar or a vector.

Data Types: double

**sigmaref — Reflectivity at nominal reference frequency**

positive real scalar

Reflectivity at nominal reference frequency in square meters per square meter, specified as a positive real scalar.

Data Types: double

**n — Frequency-dependent proportionality factor**

1 (default) | positive real scalar

Frequency-dependent proportionality factor, specified as a real scalar. For distributed targets,  $n$  varies between 0 and 1. For nondistributed targets,  $n$  is a positive real scalar.

Data Types: double

**Output Arguments****neq — Noise equivalent reflectivity**

matrix

Noise equivalent reflectivity in decibels, returned as a matrix. The rows of `neq` correspond to the frequency values in `freq`. The columns of `neq` correspond to the image SNR values in `imgsnr`.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`radareqsarpow` | `radareqsarrng` | `radareqsarsnr` | `rainscr`

**Introduced in R2021a**

## rainscr

Signal-to-clutter ratio due to rain

### Syntax

```
scr = rainscr(lambda,rrcs,tgtrcs,t)
scr = rainscr(lambda,rrcs,tgtrcs,t,vbrain)
```

### Description

`scr = rainscr(lambda,rrcs,tgtrcs,t)` returns the signal-to-clutter ratio (SCR) due to rain.

`scr = rainscr(lambda,rrcs,tgtrcs,t,vbrain)` specifies the rain velocity bandwidth.

### Examples

#### Signal-to-Clutter Ratio Due to Rain

Estimate the signal-to-clutter-ratio due to rain of a side-looking airborne SAR. The SAR moves at 50 m/s in a direction orthogonal to the antenna boresight and operates at a frequency of 1.5 GHz. The rain rates are 0.25 mm/hr, 1 mm/hr, 4 mm/hr, and 16 mm/hr. The rain clutter volume is 20 m<sup>3</sup>. The SAR module has aperture processing length of 100 m. Assume the target RCS is 1 m<sup>2</sup> and the velocity bandwidth of the rain is 4 m/s.

```
v = 50;
f = 1.5e9;
lambda = freq2wavelen(f);
```

```
rr = [0.25 1 4 16];
vol = 20;
```

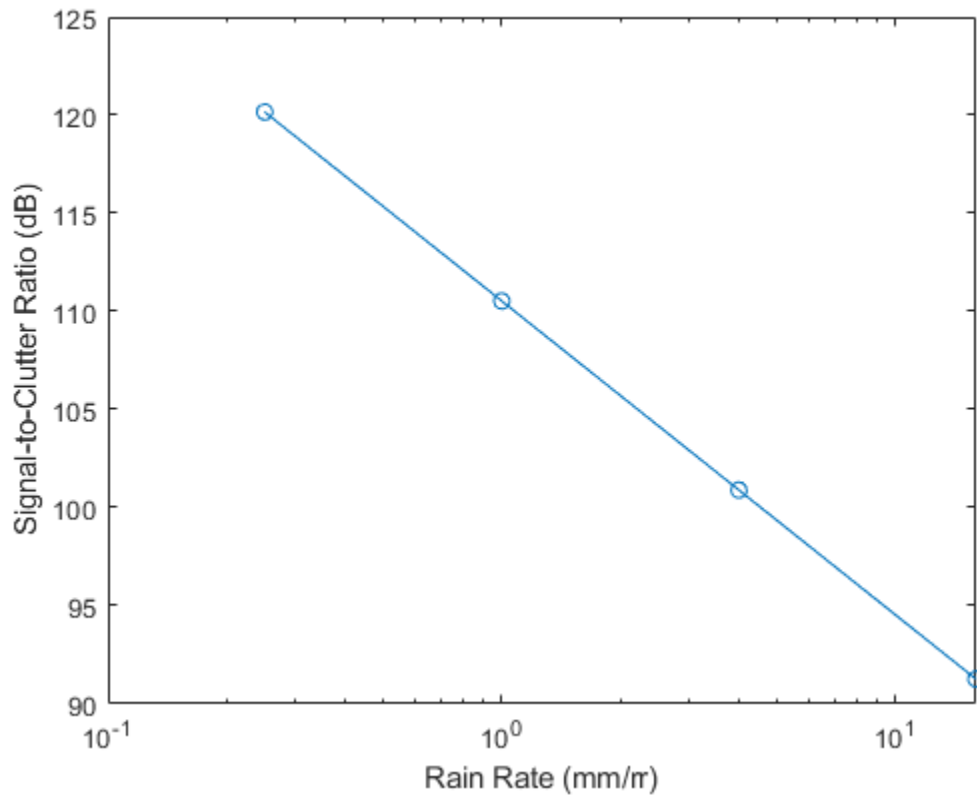
```
L = 100;
tgtrcs = 1;
vbrain = 4;
```

Compute the signal-to-clutter ratio. Use `rainreflectivity` and `clutterVolumeRCS` to compute the rain radar cross-section. Use `sarinttime` to compute the aperture collection interval.

```
volref = rainreflectivity(f,rr);
rrcs = clutterVolumeRCS(volref,vol);
t = sarinttime(v,L);
scr = rainscr(lambda,rrcs,tgtrcs,t,vbrain);
```

Plot the signal-to-clutter ratio as a function of the rain rate.

```
semilogx(rr,scr,'o-')
xlabel('Rain Rate (mm/rr)')
ylabel('Signal-to-Clutter Ratio (dB)')
```



## Input Arguments

### **lambda** — Radar wavelength

positive real scalar | vector

Radar wavelength in meters, specified as a positive real scalar or a vector.

Data Types: double

### **rrcs** — Rain radar cross-section

scalar | vector

Rain radar cross-section (RCS) in square meters, specified as a scalar or a vector.

Data Types: double

### **tgtrcs** — Target radar cross-section

scalar

Target RCS in square meters, specified as a scalar.

Data Types: double

### **t** — Aperture collection interval

positive real scalar

Aperture collection interval in seconds, specified as a positive real scalar.

Data Types: double

**vbrain — Rain velocity bandwidth**

4 (default) | positive scalar

Rain velocity bandwidth in meters per second, specified as a positive scalar.

Data Types: double

**Output Arguments****scr — Signal-to-clutter ratio due to rain**

matrix

Signal-to-clutter ratio due to rain in decibels, returned as a matrix. The rows of `scr` correspond to the radar wavelength values in `lambda`. The columns of `scr` correspond to the rain RCS values in `rrcs`.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`radareqsarpow` | `radareqsarrng` | `radareqsarsnr` | `sarnoiseref1`

**Introduced in R2021a**

# sarintang

Coherent integration angle for SAR

## Syntax

```
ciang = sarintang(lambda,azres)
ciang = sarintang(lambda,azres,azb)
```

## Description

`ciang = sarintang(lambda,azres)` returns the coherent integration angle, `ciang`, through which the target is viewed during the coherent processing aperture.

`ciang = sarintang(lambda,azres,azb)` specifies the azimuth impulse broadening factor, `azb`, due to data weighting or windowing for sidelobe control.

## Examples

### Coherent Integration Angle

Estimate the coherent integration angle of a side-looking airborne synthetic aperture radar (SAR) with an operating frequency of 10 GHz and a cross-range resolution of 1 m. Assume the azimuth broadening factor to be 1.3.

```
f = 10e9;
azres = 1;
azb = 1.3;
```

Compute the coherent integration angle.

```
lambda = freq2wavelen(f);
ciang = sarintang(lambda,azres,azb)

ciang = 1.1165
```

## Input Arguments

### **lambda** — Radar wavelength

positive real scalar | vector

Radar wavelength in meters, specified as a positive real scalar or a vector.

Data Types: `double`

### **azres** — Image azimuth or cross-range resolution

positive real scalar | vector

Image azimuth or cross-range resolution in meters, specified as a positive real scalar or a vector.

Data Types: `double`

**azb — Azimuth impulse broadening factor**

1 (default) | positive real scalar

Azimuth impulse broadening factor, specified as a positive real scalar. `azb` expresses the actual -3 dB mainlobe width with respect to the nominal width. Typical window functions like `hamming` and `hann` exhibit `azb` values in the range from 1 to 1.5.

Data Types: double

**Output Arguments****ciang — Coherent integration angle**

matrix

Coherent integration angle in degrees, returned as a matrix. The rows in `ciang` correspond to the wavelengths in `lambda` and the columns correspond to the resolution in `azres`.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**`sardispgrazang` | `sarsquintang`**Introduced in R2021a**



# sardispgrazang

Display grazing angle for SAR data collection

## Syntax

```
dgrazang = sardispgrazang(pos)
dgrazang = sardispgrazang(pos,slope)
dgrazang = sardispgrazang(pos,slope,axes)
```

## Description

`dgrazang = sardispgrazang(pos)` returns the display grazing angle, `dgrazang`, of an image defined at the aperture reference point.

`dgrazang = sardispgrazang(pos,slope)` specifies the slope angle for the image display plane.

`dgrazang = sardispgrazang(pos,slope,axes)` specifies the antenna phase center traveling axis.

## Examples

### Display Grazing Angle

Compute the grazing angle of a SAR image projected on the image display plane of an antenna phase center located at [1000,2000,5000] meters with respect to a scene centered at [10,10,10] meters. Assume the slope angle for the image display plane is 30°.

```
pos1 = [1000;2000;5000];
pos2 = [10;10;10];
rngvec = pos1-pos2;
slope = 30;
```

Compute the image grazing angle.

```
dgrazang = sardispgrazang(rngvec,slope)
dgrazang = 27.3352
```

## Input Arguments

### **pos** — Measured line of sight vector

3-by-*N* matrix in meters

Measured line of sight vector from the scene center to the antenna phase center, specified as a 3-by-*N* matrix in meters. Each column of `pos` represents a measured line-of-sight position. The geometric location of the antenna phase center at the center of the processing aperture is the aperture reference point. The antenna phase center serves as the reference point for the phase history of the received signal.

Example: [1000;2000;5000]

Data Types: double

### **slope — Slope angle**

$\theta$  (default) | scalar between 0 and 90°

Slope angle, specified as a scalar between 0 and 90°. The slope angle is the angle between the image display plane and the scene center plane.

Data Types: double

### **axes — Antenna phase center traveling axis**

'x' (default) | 'y' | 'z'

Antenna phase center traveling axis, specified as 'x', 'y', or 'z'.

- 'x' — The antenna phase center travels in the x-direction and the surface plane is the xy-plane.
- 'y' — The antenna phase center travels in the y-direction and the surface plane is the yz-plane.
- 'z' — The antenna phase center travels in the z-direction and the surface plane is the zx-plane.

Data Types: double

## **Output Arguments**

### **dgrazang — Display grazing angle**

1-by-*N* row vector in degrees

Display grazing angle, returned as a 1-by-*N* row vector in degrees. The display grazing angle is the angle between the vertical projections of the slant range vector onto the image display plane and the scene center plane. The image display plane is the plane onto which the image formation processor projects the scatterers in a 3-D scene.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

sarintang | sarsquintang

**Introduced in R2021a**

# sarsquintang

Squint angle for SAR data collection

## Syntax

```
[sqang,dsqang] = sarsquintang(pos)
[sqang,dsqang] = sarsquintang(pos,slope)
[sqang,dsqang] = sarsquintang(pos,slope,axes)
```

## Description

[sqang,dsqang] = sarsquintang(pos) returns the squint angle, sqang, and display squint angle, dsqang, of an image defined at the aperture reference point.

[sqang,dsqang] = sarsquintang(pos,slope) specifies the slope angle for the image display plane.

[sqang,dsqang] = sarsquintang(pos,slope,axes) specifies the antenna phase center traveling axis.

## Examples

### Squint Angle and Display Squint Angle

Compute the squint angle of a SAR image and the projected angle on the image display plane of an antenna phase center located at [1000,2000,5000] meters with respect to a scene center. Assume the slope angle for the image display plane is 30°.

```
pos = [1000;2000;5000];
slope = 30;
```

Compute the image squint angle and display squint angle.

```
[sqang,dsqang] = sarsquintang(pos,slope)
```

```
sqang = 63.4349
```

```
dsqang = 66.5868
```

## Input Arguments

### pos — Measured line of sight vector

3-by-*N* matrix in meters

Measured line of sight vector from the scene center to the antenna phase center, specified as a 3-by-*N* matrix in meters. Each column of pos represents a measured line-of-sight position. The geometric location of the antenna phase center at the center of the processing aperture is the aperture reference point. The antenna phase center serves as the reference point for the phase history of the received signal.

Example: [1000;2000;5000]

Data Types: double

### **slope — Slope angle**

$\theta$  (default) | scalar between 0 and 90°

Slope angle, specified as a scalar between 0 and 90°. The slope angle is the angle between the image display plane and the scene center plane.

Data Types: double

### **axes — Antenna phase center traveling axis**

'x' (default) | 'y' | 'z'

Antenna phase center traveling axis, specified as 'x', 'y', or 'z'.

- 'x' — The antenna phase center travels in the x-direction and the surface plane is the xy-plane.
- 'y' — The antenna phase center travels in the y-direction and the surface plane is the yz-plane.
- 'z' — The antenna phase center travels in the z-direction and the surface plane is the zx-plane.

Data Types: double

## **Output Arguments**

### **sqang — Squint angle**

1-by-*N* row vector in degrees

Squint angle, returned as a 1-by-*N* row vector in degrees. The squint angle is the angle between the antenna phase center axis and the vertical projection of the slant range vector onto the scene center plane.

### **dsqang — Display squint angle**

1-by-*N* row vector in degrees

Display squint angle, returned as a 1-by-*N* row vector in degrees. The display squint angle is the angle between the antenna phase center axis and the vertical projection of the slant range vector onto the image display center plane. The image display plane is the plane onto which the image formation processor projects the scatterers in a 3-D scene.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

sardispgrazang | sarintang

**Introduced in R2021a**

# randrot

Uniformly distributed random rotations

## Syntax

```
R = randrot
R = randrot(m)
R = randrot(m1,...,mN)
R = randrot([m1,...,mN])
```

## Description

`R = randrot` returns a unit quaternion drawn from a uniform distribution of random rotations.

`R = randrot(m)` returns an  $m$ -by- $m$  matrix of unit quaternions drawn from a uniform distribution of random rotations.

`R = randrot(m1,...,mN)` returns an  $m1$ -by-...-by- $mN$  array of random unit quaternions, where  $m1$ , ...,  $mN$  indicate the size of each dimension. For example, `randrot(3,4)` returns a 3-by-4 matrix of random unit quaternions.

`R = randrot([m1,...,mN])` returns an  $m1$ -by-...-by- $mN$  array of random unit quaternions, where  $m1$ ,...,  $mN$  indicate the size of each dimension. For example, `randrot([3,4])` returns a 3-by-4 matrix of random unit quaternions.

## Examples

### Matrix of Random Rotations

Generate a 3-by-3 matrix of uniformly distributed random rotations.

```
r = randrot(3)
```

```
r = 3x3 quaternion array
```

```
    0.17446 + 0.59506i - 0.73295j + 0.27976k    0.69704 - 0.060589i + 0.68679j - 0.19699k
    0.21908 - 0.89875i - 0.298j + 0.23548k    -0.049744 + 0.59691i + 0.56459j + 0.56789k
    0.6375 + 0.49338i - 0.24049j + 0.54068k    0.2979 - 0.53568i + 0.31819j + 0.72329k
```

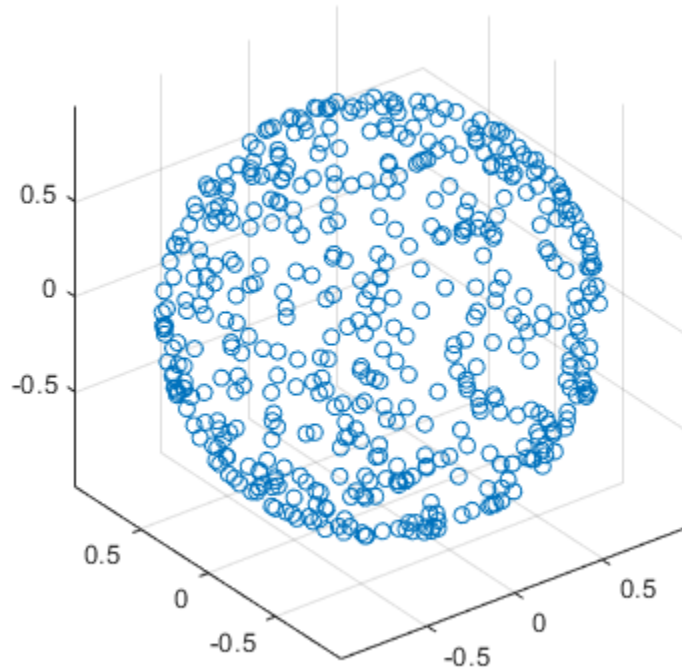
### Create Uniform Distribution of Random Rotations

Create a vector of 500 random quaternions. Use `rotatepoint` to visualize the distribution of the random rotations applied to point (1, 0, 0).

```
q = randrot(500,1);
```

```
pt = rotatepoint(q, [1 0 0]);
```

```
figure
scatter3(pt(:,1), pt(:,2), pt(:,3))
axis equal
```



## Input Arguments

### **m** — Size of square matrix

integer

Size of square quaternion matrix, specified as an integer value. If  $m$  is 0 or negative, then  $R$  is returned as an empty matrix.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **m1, ..., mN** — Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integer values. If the size of any dimension is 0 or negative, then  $R$  is returned as an empty array.

Example: `randrot(2,3)` returns a 2-by-3 matrix of random quaternions.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

### **[m1, ..., mN]** — Vector of size of each dimension

row vector of integer values

Vector of size of each dimension, specified as a row vector of two or more integer values. If the size of any dimension is 0 or negative, then R is returned as an empty array.

Example: `randrot([2,3])` returns a 2-by-3 matrix of random quaternions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### R — Random quaternions

`scalar` | `vector` | `matrix` | `multidimensional array`

Random quaternions, returned as a quaternion or array of quaternions.

Data Types: `quaternion`

## References

[1] Shoemake, K. "Uniform Random Rotations." *Graphics Gems III* (K. David, ed.). New York: Academic Press, 1992.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

`quaternion`

**Introduced in R2021a**

## angvel

Angular velocity from quaternion array

### Syntax

```
AV = angvel(Q,dt,'frame')
AV = angvel(Q,dt,'point')
[AV,qf] = angvel(Q,dt,fp,qi)
```

### Description

`AV = angvel(Q,dt,'frame')` returns the angular velocity array from an array of quaternions, `Q`. The quaternions in `Q` correspond to frame rotation. The initial quaternion is assumed to represent zero rotation.

`AV = angvel(Q,dt,'point')` returns the angular velocity array from an array of quaternions, `Q`. The quaternions in `Q` correspond to point rotation. The initial quaternion is assumed to represent zero rotation.

`[AV,qf] = angvel(Q,dt,fp,qi)` allows you to specify the initial quaternion, `qi`, and the type of rotation, `fp`. It also returns the final quaternion, `qf`.

### Examples

#### Generate Angular Velocity From Quaternion Array

Create an array of quaternions.

```
eulerAngles = [(0:10:90).',zeros(numel(0:10:90),2)];
q = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Specify the time step and generate the angular velocity array.

```
dt = 1;
av = angvel(q,dt,'frame') % units in rad/s
```

```
av = 10x3
```

```
    0         0         0
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
    0         0    0.1743
```



## Input Arguments

### **Q — Quaternions**

*N*-by-1 vector of quaternions

Quaternions, specified as an *N*-by-1 vector of quaternions.

Data Types: quaternion

### **dt — Time step**

nonnegative scalar

Time step, specified as a nonnegative scalar.

Data Types: single | double

### **fp — Type of rotation**

'frame' | 'point'

Type of rotation, specified as 'frame' or 'point'.

### **qi — Initial quaternion**

quaternion

Initial quaternion, specified as a quaternion.

Data Types: quaternion

## Output Arguments

### **AV — Angular velocity**

*N*-by-3 real matrix

Angular velocity, returned as an *N*-by-3 real matrix. *N* is the number of quaternions given in the input *Q*. Each row of the matrix corresponds to an angular velocity vector.

### **qf — Final quaternion**

quaternion

Final quaternion, returned as a quaternion. *qf* is the same as the last quaternion in the *Q* input.

Data Types: quaternion

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

quaternion

**Introduced in R2021a**

## rotvecd

Convert quaternion to rotation vector (degrees)

### Syntax

```
rotationVector = rotvecd(quat)
```

### Description

`rotationVector = rotvecd(quat)` converts the quaternion array, `quat`, to an  $N$ -by-3 matrix of equivalent rotation vectors in degrees. The elements of `quat` are normalized before conversion.

### Examples

#### Convert Quaternion to Rotation Vector in Degrees

Convert a random quaternion scalar to a rotation vector in degrees.

```
quat = quaternion(randn(1,4));  
rotvecd(quat)
```

```
ans = 1×3
```

```
    96.6345  -119.0274   45.4312
```

### Input Arguments

#### **quat** — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

### Output Arguments

#### **rotationVector** — Rotation vector (degrees)

$N$ -by-3 matrix

Rotation vector representation, returned as an  $N$ -by-3 matrix of rotation vectors, where each row represents the  $[x\ y\ z]$  angles of the rotation vectors in degrees. The  $i$ th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: single | double

## Algorithms

All rotations in 3-D can be represented by four elements: a three-element axis of rotation and a rotation angle. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

$$q = \cos(\theta/2) + \sin(\theta/2)(xi + yj + zk),$$

where  $\theta$  is the angle of rotation in degrees, and  $[x,y,z]$  represent the axis of rotation.

Given a quaternion of the form

$$q = a + bi + cj + dk,$$

you can solve for the rotation angle using the axis-angle form of quaternions:

$$\theta = 2\cos^{-1}(a).$$

Assuming a normalized axis, you can rewrite the quaternion as a rotation vector without loss of information by distributing  $\theta$  over the parts  $b$ ,  $c$ , and  $d$ . The rotation vector representation of  $q$  is

$$q_{rv} = \frac{\theta}{\sin(\theta/2)}[b, c, d].$$

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

rotvec | euler | eulerd

### Objects

quaternion

**Introduced in R2021a**

## eulerd

Convert quaternion to Euler angles (degrees)

### Syntax

```
eulerAngles = eulerd(quat, rotationSequence, rotationType)
```

### Description

`eulerAngles = eulerd(quat, rotationSequence, rotationType)` converts the quaternion, `quat`, to an  $N$ -by-3 matrix of Euler angles in degrees.

### Examples

#### Convert Quaternion to Euler Angles in Degrees

Convert a quaternion frame rotation to Euler angles in degrees using the 'ZYX' rotation sequence.

```
quat = quaternion([0.7071 0.7071 0 0]);
eulerAnglesDegrees = eulerd(quat, 'ZYX', 'frame')
```

```
eulerAnglesDegrees = 1×3
```

```
    0         0    90.0000
```

### Input Arguments

#### **quat** — Quaternion to convert to Euler angles

scalar | vector | matrix | multidimensional array

Quaternion to convert to Euler angles, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

#### **rotationSequence** — Rotation sequence

'ZYX' | 'YZX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX'

Rotation sequence of Euler angle representation, specified as a character vector or string.

The rotation sequence defines the order of rotations about the axes. For example, if you specify a rotation sequence of 'YZX':

- 1 The first rotation is about the y-axis.
- 2 The second rotation is about the new z-axis.
- 3 The third rotation is about the new x-axis.

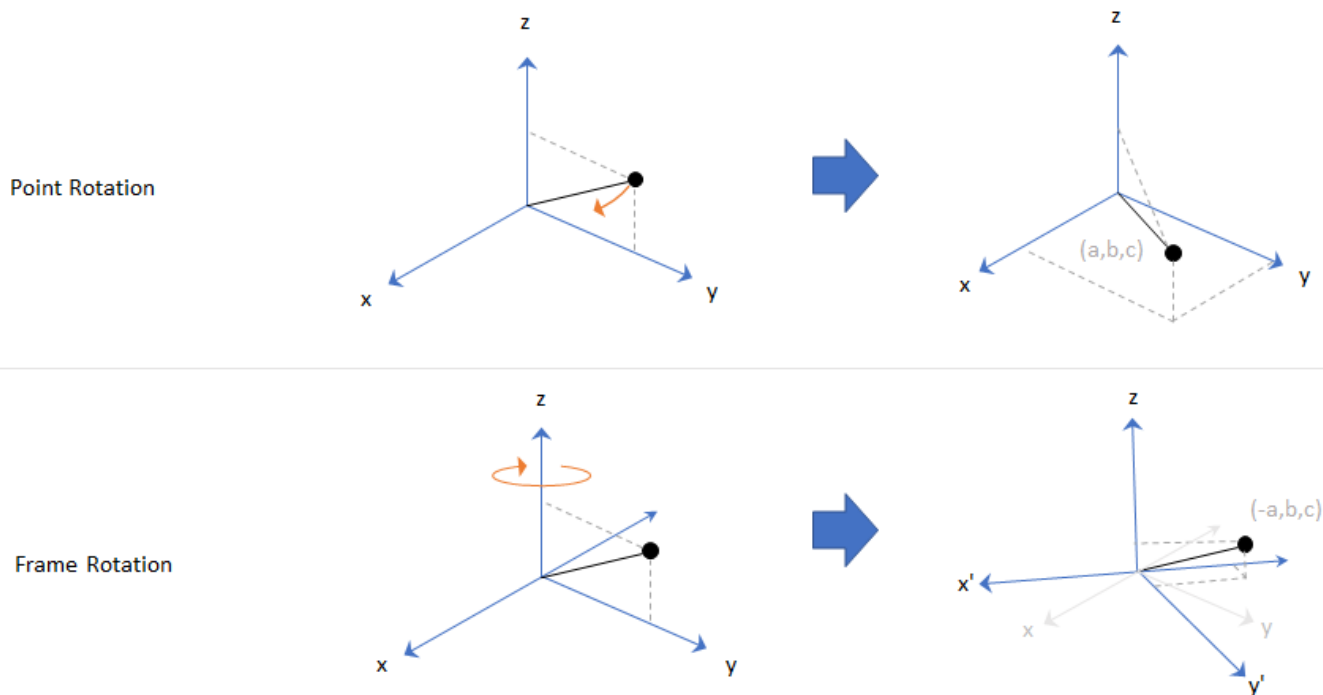
Data Types: char | string

**rotationType — Type of rotation**

'point' | 'frame'

Type of rotation, specified as 'point' or 'frame'.

In a point rotation, the frame is static and the point moves. In a frame rotation, the point is static and the frame moves. Point rotation and frame rotation define equivalent angular displacements but in opposite directions.



Data Types: char | string

**Output Arguments****eulerAngles — Euler angle representation (degrees)***N*-by-3 matrix

Euler angle representation in degrees, returned as a *N*-by-3 matrix. *N* is the number of quaternions in the `quat` argument.

For each row of `eulerAngles`, the first column corresponds to the first axis in the rotation sequence, the second column corresponds to the second axis in the rotation sequence, and the third column corresponds to the third axis in the rotation sequence.

The data type of the Euler angles representation is the same as the underlying data type of `quat`.

Data Types: single | double

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

euler | rotateframe | rotatepoint

### **Objects**

quaternion

**Introduced in R2021a**

# meanrot

Quaternion mean rotation

## Syntax

```
quatAverage = meanrot(quat)
quatAverage = meanrot(quat,dim)
quatAverage = meanrot( ____,nanflag)
```

## Description

`quatAverage = meanrot(quat)` returns the average rotation of the elements of `quat` along the first array dimension whose size not does equal 1.

- If `quat` is a vector, `meanrot(quat)` returns the average rotation of the elements.
- If `quat` is a matrix, `meanrot(quat)` returns a row vector containing the average rotation of each column.
- If `quat` is a multidimensional array, then `meanrot(quat)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes 1 while the sizes of all other dimensions remain the same.

The `meanrot` function normalizes the input quaternions, `quat`, before calculating the mean.

`quatAverage = meanrot(quat,dim)` return the average rotation along dimension `dim`. For example, if `quat` is a matrix, then `meanrot(quat,2)` is a column vector containing the mean of each row.

`quatAverage = meanrot( ____,nanflag)` specifies whether to include or omit NaN values from the calculation for any of the previous syntaxes. `meanrot(quat,'includenan')` includes all NaN values in the calculation while `mean(quat,'omitnan')` ignores them.

## Examples

### Quaternion Mean Rotation

Create a matrix of quaternions corresponding to three sets of Euler angles.

```
eulerAngles = [40 20 10; ...
               50 10 5; ...
               45 70 1];
```

```
quat = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Determine the average rotation represented by the quaternions. Convert the average rotation to Euler angles in degrees for readability.

```
quatAverage = meanrot(quat)
```

```
quatAverage = quaternion
    0.88863 - 0.062598i + 0.27822j + 0.35918k

eulerAverage = eulerd(quatAverage, 'ZYX', 'frame')

eulerAverage = 1×3
    45.7876    32.6452    6.0407
```

### Average Out Rotational Noise

Use `meanrot` over a sequence of quaternions to average out additive noise.

Create a vector of  $1e6$  quaternions whose distance, as defined by the `dist` function, from `quaternion(1,0,0,0)` is normally distributed. Plot the Euler angles corresponding to the noisy quaternion vector.

```
nrows = 1e6;
ax = 2*rand(nrows,3) - 1;
ax = ax./sqrt(sum(ax.^2,2));
ang = 0.5*randn(size(ax,1),1);
q = quaternion(ax.*ang, 'rotvec');

noisyEulerAngles = eulerd(q, 'ZYX', 'frame');

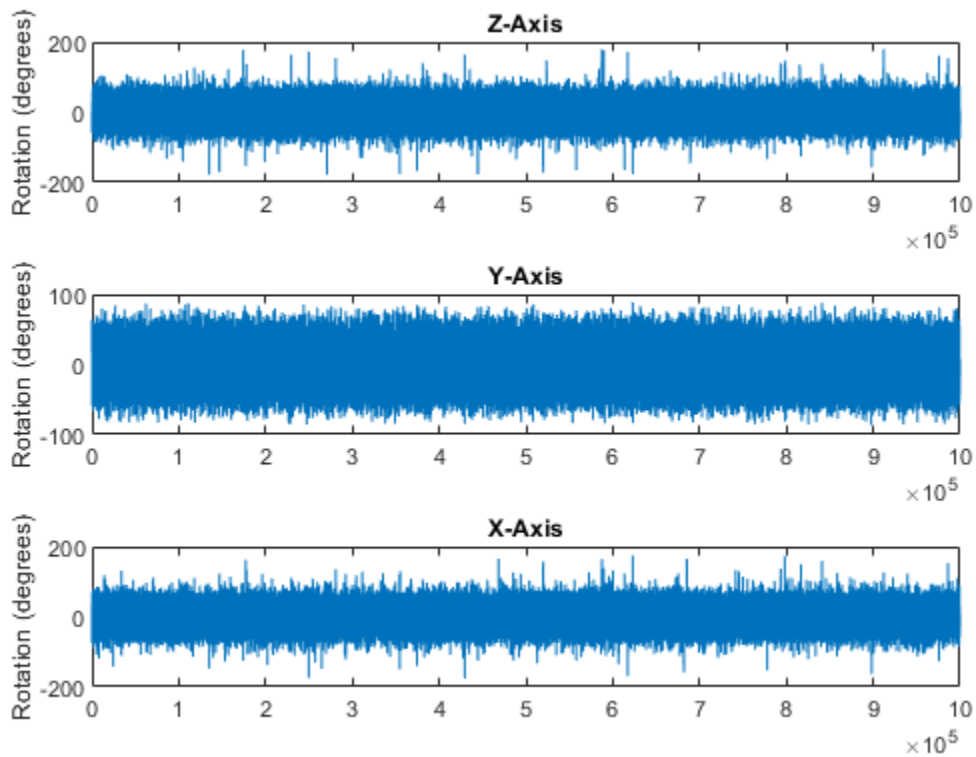
figure(1)

subplot(3,1,1)
plot(noisyEulerAngles(:,1))
title('Z-Axis')
ylabel('Rotation (degrees)')
hold on

subplot(3,1,2)
plot(noisyEulerAngles(:,2))
title('Y-Axis')
ylabel('Rotation (degrees)')
hold on

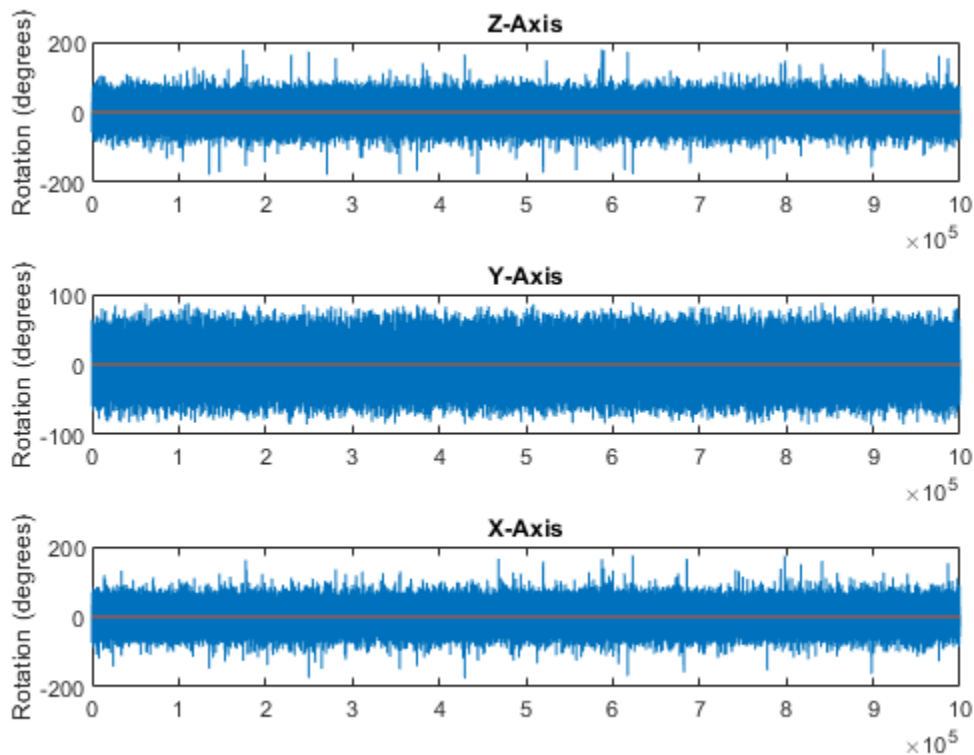
subplot(3,1,3)
plot(noisyEulerAngles(:,3))
title('X-Axis')
ylabel('Rotation (degrees)')
hold on
```





Use `meanrot` to determine the average quaternion given the vector of quaternions. Convert to Euler angles and plot the results.

```
qAverage = meanrot(q);
qAverageInEulerAngles = eulerd(qAverage, 'ZYX', 'frame');
figure(1)
subplot(3,1,1)
plot(ones(nrows,1)*qAverageInEulerAngles(:,1))
title('Z-Axis')
subplot(3,1,2)
plot(ones(nrows,1)*qAverageInEulerAngles(:,2))
title('Y-Axis')
subplot(3,1,3)
plot(ones(nrows,1)*qAverageInEulerAngles(:,3))
title('X-Axis')
```



## The meanrot Algorithm and Limitations

### The meanrot Algorithm

The `meanrot` function outputs a quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices. Consider two quaternions:

- `q0` represents no rotation.
- `q90` represents a 90 degree rotation about the x-axis.

```
q0 = quaternion([0 0 0], 'eulerd', 'ZYX', 'frame');
q90 = quaternion([0 0 90], 'eulerd', 'ZYX', 'frame');
```

Create a quaternion sweep, `qSweep`, that represents rotations from 0 to 180 degrees about the x-axis.

```
eulerSweep = (0:1:180)';
qSweep = quaternion([zeros(numel(eulerSweep),2), eulerSweep], ...
    'eulerd', 'ZYX', 'frame');
```

Convert `q0`, `q90`, and `qSweep` to rotation matrices. In a loop, calculate the metric to minimize for each member of the quaternion sweep. Plot the results and return the value of the Euler sweep that corresponds to the minimum of the metric.

```
r0 = rotmat(q0, 'frame');
r90 = rotmat(q90, 'frame');
```

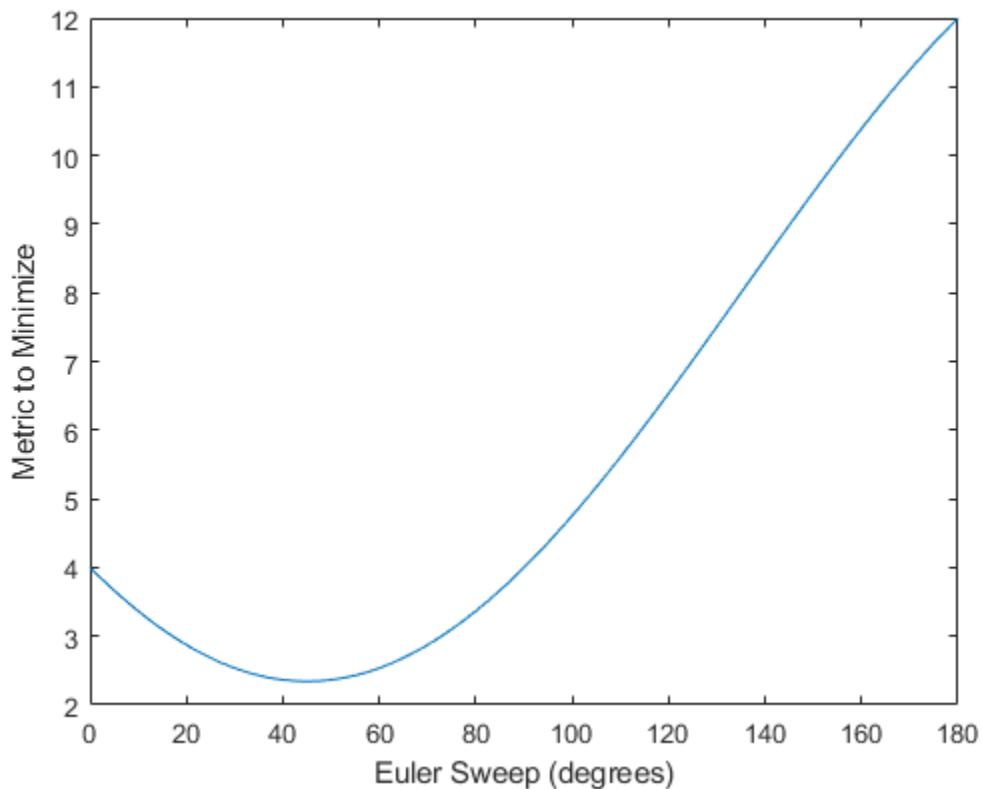
```

rSweep = rotmat(qSweep, 'frame');

metricToMinimize = zeros(size(rSweep,3),1);
for i = 1:numel(qSweep)
    metricToMinimize(i) = norm((rSweep(:,:,i) - r0), 'fro').^2 + ...
        norm((rSweep(:,:,i) - r90), 'fro').^2;
end

plot(eulerSweep,metricToMinimize)
xlabel('Euler Sweep (degrees)')
ylabel('Metric to Minimize')

```



```

[~,eulerIndex] = min(metricToMinimize);
eulerSweep(eulerIndex)

```

```
ans = 45
```

The minimum of the metric corresponds to the Euler angle sweep at 45 degrees. That is, meanrot defines the average between quaternion([0 0 0], 'ZYX', 'frame') and quaternion([0 0 90], 'ZYX', 'frame') as quaternion([0 0 45], 'ZYX', 'frame'). Call meanrot with q0 and q90 to verify the same result.

```
eulerd(meanrot([q0,q90]), 'ZYX', 'frame')
```

```
ans = 1x3
```

```
0 0 45.0000
```

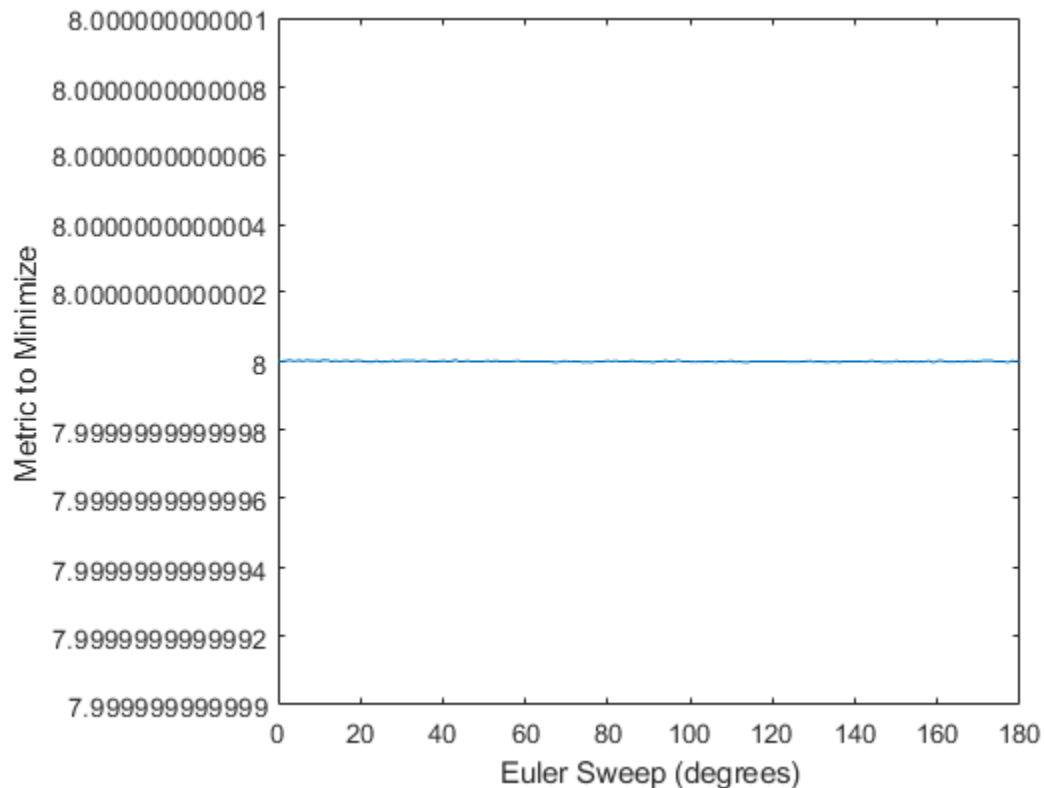
### Limitations

The metric that `meanrot` uses to determine the mean rotation is not unique for quaternions significantly far apart. Repeat the experiment above for quaternions that are separated by 180 degrees.

```
q180 = quaternion([0 0 180], 'eulerd', 'ZYX', 'frame');
r180 = rotmat(q180, 'frame');

for i = 1:numel(qSweep)
    metricToMinimize(i) = norm((rSweep(:, :, i) - r0), 'fro').^2 + ...
        norm((rSweep(:, :, i) - r180), 'fro').^2;
end

plot(eulerSweep, metricToMinimize)
xlabel('Euler Sweep (degrees)')
ylabel('Metric to Minimize')
```



```
[~,eulerIndex] = min(metricToMinimize);
eulerSweep(eulerIndex)
```

```
ans = 159
```

Quaternion means are usually calculated for rotations that are close to each other, which makes the edge case shown in this example unlikely in real-world applications. To average two quaternions that are significantly far apart, use the `slerp` function. Repeat the experiment using `slerp` and verify that the quaternion mean returned is more intuitive for large distances.

```
qMean = slerp(q0,q180,0.5);
q0_q180 = eulerd(qMean,'ZYX','frame')
```

```
q0_q180 = 1×3
          0          0  90.0000
```

## Input Arguments

### **quat** — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion for which to calculate the mean, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

### **dim** — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to 1. The `size(quatAverage,dim)` is 1, while the sizes of all other dimensions remain the same.

Data Types: double | single

### **nanflag** — NaN condition

'includenan' (default) | 'omitnan'

NaN condition, specified as one of these values:

- 'includenan' -- Include NaN values when computing the mean rotation, resulting in NaN.
- 'omitnan' -- Ignore all NaN values in the input.

Data Types: char | string

## Output Arguments

### **quatAverage** — Quaternion average rotation

scalar | vector | matrix | multidimensional array

Quaternion average rotation, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: single | double

## Algorithms

meanrot determines a quaternion mean,  $\bar{q}$ , according to [1].  $\bar{q}$  is the quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices:

$$\bar{q} = \arg \min_{q \in S^3} \sum_{i=1}^n \|A(q) - A(q_i)\|_F^2$$

## References

- [1] Markley, F. Landis, Yang Chen, John Lucas Crassidis, and Yaakov Oshman. "Average Quaternions." *Journal of Guidance, Control, and Dynamics*. Vol. 30, Issue 4, 2007, pp. 1193-1197.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

dist | slerp

### Objects

quaternion

**Introduced in R2021a**

# slerp

Spherical linear interpolation

## Syntax

```
q0 = slerp(q1,q2,T)
```

## Description

`q0 = slerp(q1,q2,T)` spherically interpolates between `q1` and `q2` by the interpolation coefficient `T`. The function always chooses the shorter interpolation path between `q1` and `q2`.

## Examples

### Interpolate Between Two Quaternions

Create two quaternions with the following interpretation:

- 1 `a` = 45 degree rotation around the z-axis
- 2 `c` = -45 degree rotation around the z-axis

```
a = quaternion([45,0,0], 'eulerd', 'ZYX', 'frame');
c = quaternion([-45,0,0], 'eulerd', 'ZYX', 'frame');
```

Call `slerp` with the quaternions `a` and `c` and specify an interpolation coefficient of 0.5.

```
interpolationCoefficient = 0.5;
b = slerp(a,c,interpolationCoefficient);
```

The output of `slerp`, `b`, represents an average rotation of `a` and `c`. To verify, convert `b` to Euler angles in degrees.

```
averageRotation = eulerd(b, 'ZYX', 'frame')
averageRotation = 1×3
```

```
    0    0    0
```

The interpolation coefficient is specified as a normalized value between 0 and 1, inclusive. An interpolation coefficient of 0 corresponds to the `a` quaternion, and an interpolation coefficient of 1 corresponds to the `c` quaternion. Call `slerp` with coefficients 0 and 1 to confirm.

```
b = slerp(a,c,[0,1]);
eulerd(b, 'ZYX', 'frame')
```

```
ans = 2×3
```

```
45.0000    0    0
```

```
-45.0000    0    0
```

You can create smooth paths between quaternions by specifying arrays of equally spaced interpolation coefficients.

```
path = 0:0.1:1;
interpolatedQuaternions = slerp(a,c,path);
```

For quaternions that represent rotation only about a single axis, specifying interpolation coefficients as equally spaced results in quaternions equally spaced in Euler angles. Convert `interpolatedQuaternions` to Euler angles and verify that the difference between the angles in the path is constant.

```
k = eulerd(interpolatedQuaternions, 'ZYX', 'frame');
abc = abs(diff(k))
```

```
abc = 10x3
    9.0000    0    0
    9.0000    0    0
    9.0000    0    0
    9.0000    0    0
    9.0000    0    0
    9.0000    0    0
    9.0000    0    0
    9.0000    0    0
    9.0000    0    0
    9.0000    0    0
```

Alternatively, you can use the `dist` function to verify that the distance between the interpolated quaternions is consistent. The `dist` function returns angular distance in radians; convert to degrees for easy comparison.

```
def = rad2deg(dist(interpolatedQuaternions(2:end),interpolatedQuaternions(1:end-1)))
def = 1x10
    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000
```

### SLERP Minimizes Great Circle Path

The SLERP algorithm interpolates along a great circle path connecting two quaternions. This example shows how the SLERP algorithm minimizes the great circle path.

Define three quaternions:

- 1 q0 - quaternion indicating no rotation from the global frame
- 2 q179 - quaternion indicating a 179 degree rotation about the z-axis
- 3 q180 - quaternion indicating a 180 degree rotation about the z-axis



4 q181 - quaternion indicating a 181 degree rotation about the z-axis

```
q0 = ones(1, 'quaternion');
q179 = quaternion([179,0,0], 'eulerd', 'ZYX', 'frame');
q180 = quaternion([180,0,0], 'eulerd', 'ZYX', 'frame');
q181 = quaternion([181,0,0], 'eulerd', 'ZYX', 'frame');
```

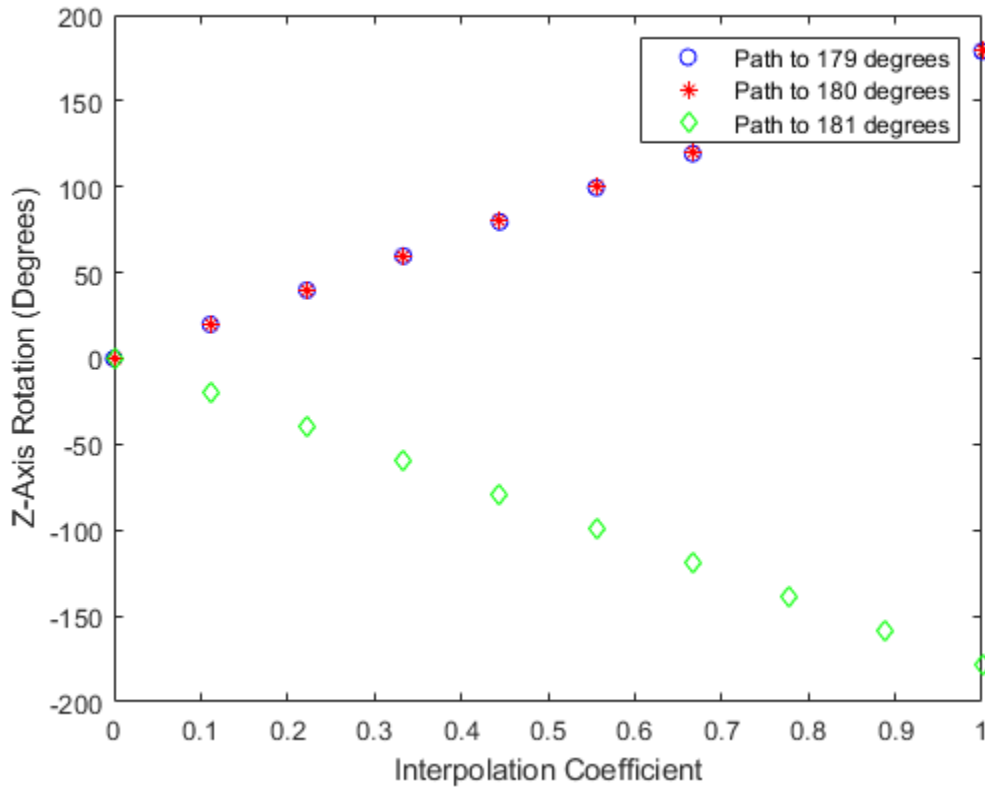
Use `slerp` to interpolate between `q0` and the three quaternion rotations. Specify that the paths are traveled in 10 steps.

```
T = linspace(0,1,10);
q179path = slerp(q0,q179,T);
q180path = slerp(q0,q180,T);
q181path = slerp(q0,q181,T);
```

Plot each path in terms of Euler angles in degrees.

```
q179pathEuler = eulerd(q179path, 'ZYX', 'frame');
q180pathEuler = eulerd(q180path, 'ZYX', 'frame');
q181pathEuler = eulerd(q181path, 'ZYX', 'frame');

plot(T,q179pathEuler(:,1), 'bo', ...
      T,q180pathEuler(:,1), 'r*', ...
      T,q181pathEuler(:,1), 'gd');
legend('Path to 179 degrees', ...
       'Path to 180 degrees', ...
       'Path to 181 degrees')
xlabel('Interpolation Coefficient')
ylabel('Z-Axis Rotation (Degrees)')
```



The path between  $q_0$  and  $q_{179}$  is clockwise to minimize the great circle distance. The path between  $q_0$  and  $q_{181}$  is counterclockwise to minimize the great circle distance. The path between  $q_0$  and  $q_{180}$  can be either clockwise or counterclockwise, depending on numerical rounding.

### Show Interpolated Quaternions on Sphere

Create two quaternions.

```
q1 = quaternion([75,-20,-10], 'eulerd', 'ZYX', 'frame');
q2 = quaternion([-45,20,30], 'eulerd', 'ZYX', 'frame');
```

Define the interpolation coefficient.

```
T = 0:0.01:1;
```

Obtain the interpolated quaternions.

```
quats = slerp(q1,q2,T);
```

Obtain the corresponding rotate points.

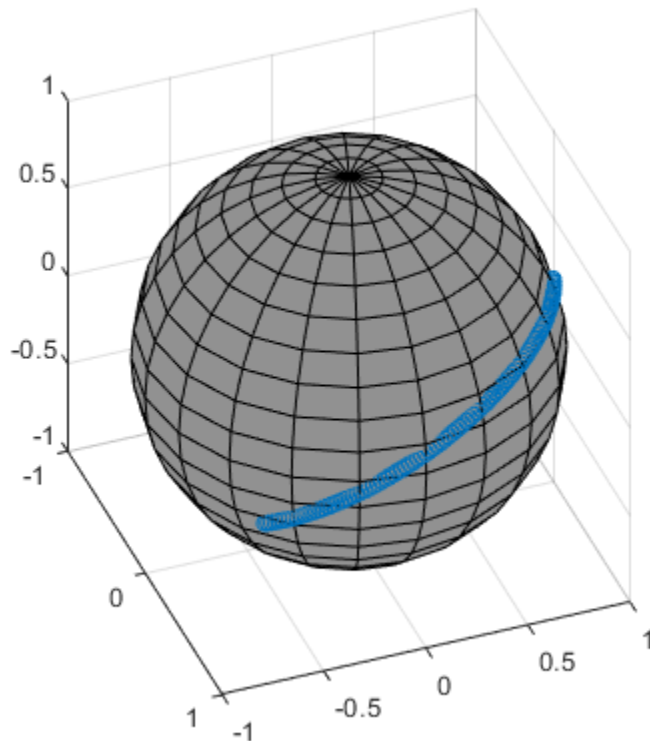
```
pts = rotatepoint(quats,[1 0 0]);
```

Show the interpolated quaternions on a unit sphere.

```
figure
[X,Y,Z] = sphere;
```

```
surf(X,Y,Z, 'FaceColor', [0.57 0.57 0.57])
hold on;

scatter3(pts(:,1),pts(:,2),pts(:,3))
view([69.23 36.60])
axis equal
```



Note that the interpolated quaternions follow the shorter path from  $q_1$  to  $q_2$ .

## Input Arguments

### **q1** – Quaternion

scalar | vector | matrix | multidimensional array

Quaternion to interpolate, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

$q_1$ ,  $q_2$ , and  $T$  must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of them is 1.

Data Types: quaternion

### **q2** – Quaternion

scalar | vector | matrix | multidimensional array

Quaternion to interpolate, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

$q_1$ ,  $q_2$ , and  $T$  must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: quaternion

### **T — Interpolation coefficient**

scalar | vector | matrix | multidimensional array

Interpolation coefficient, specified as a scalar, vector, matrix, or multidimensional array of numbers with each element in the range [0,1].

$q_1$ ,  $q_2$ , and  $T$  must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: single | double

## **Output Arguments**

### **q0 — Interpolated quaternion**

scalar | vector | matrix | multidimensional array

Interpolated quaternion, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

## **Algorithms**

Quaternion **spherical linear interpolation** (SLERP) is an extension of linear interpolation along a plane to spherical interpolation in three dimensions. The algorithm was first proposed in [1]. Given two quaternions,  $q_1$  and  $q_2$ , SLERP interpolates a new quaternion,  $q_0$ , along the great circle that connects  $q_1$  and  $q_2$ . The interpolation coefficient,  $T$ , determines how close the output quaternion is to either  $q_1$  and  $q_2$ .

The SLERP algorithm can be described in terms of sinusoids:

$$q_0 = \frac{\sin((1 - T)\theta)}{\sin(\theta)}q_1 + \frac{\sin(T\theta)}{\sin(\theta)}q_2$$

where  $q_1$  and  $q_2$  are normalized quaternions, and  $\theta$  is half the angular distance between  $q_1$  and  $q_2$ .

## **References**

[1] Shoemake, Ken. "Animating Rotation with Quaternion Curves." *ACM SIGGRAPH Computer Graphics* Vol. 19, Issue 3, 1985, pp. 345-354.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

dist | meanrot

### Objects

quaternion

**Introduced in R2021a**

## classUnderlying

Class of parts within quaternion

### Syntax

```
underlyingClass = classUnderlying(quat)
```

### Description

`underlyingClass = classUnderlying(quat)` returns the name of the class of the parts of the quaternion `quat`.

### Examples

#### Get Underlying Class of Quaternion

A quaternion is a four-part hyper-complex number used in three-dimensional representations. The four parts of the quaternion are of data type `single` or `double`.

Create two quaternions, one with an underlying data type of `single`, and one with an underlying data type of `double`. Verify the underlying data types by calling `classUnderlying` on the quaternions.

```
qSingle = quaternion(single([1,2,3,4]))
```

```
qSingle = quaternion  
1 + 2i + 3j + 4k
```

```
classUnderlying(qSingle)
```

```
ans =  
'single'
```

```
qDouble = quaternion([1,2,3,4])
```

```
qDouble = quaternion  
1 + 2i + 3j + 4k
```

```
classUnderlying(qDouble)
```

```
ans =  
'double'
```

You can separate quaternions into their parts using the `parts` function. Verify the parts of each quaternion are the correct data type. Recall that `double` is the default MATLAB® type.

```
[aS,bS,cS,dS] = parts(qSingle)
```

```
aS = single  
1
```

```

bS = single
    2

cS = single
    3

dS = single
    4

[aD,bD,cD,dD] = parts(qDouble)

aD = 1

bD = 2

cD = 3

dD = 4

```

Quaternions follow the same implicit casting rules as other data types in MATLAB. That is, a quaternion with underlying data type `single` that is combined with a quaternion with underlying data type `double` results in a quaternion with underlying data type `single`. Multiply `qDouble` and `qSingle` and verify the resulting underlying data type is `single`.

```

q = qDouble*qSingle;
classUnderlying(q)

ans =
'single'

```

## Input Arguments

### **quat** — Quaternion to investigate

scalar | vector | matrix | multi-dimensional array

Quaternion to investigate, specified as a quaternion or array of quaternions.

Data Types: quaternion

## Output Arguments

### **underlyingClass** — Underlying class of quaternion object

'single' | 'double'

Underlying class of quaternion, returned as the character vector 'single' or 'double'.

Data Types: char

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

compact | parts

### **Objects**

quaternion

**Introduced in R2021a**



## compact

Convert quaternion array to  $N$ -by-4 matrix

### Syntax

```
matrix = compact(quat)
```

### Description

`matrix = compact(quat)` converts the quaternion array, `quat`, to an  $N$ -by-4 matrix. The columns are made from the four quaternion parts. The  $i^{\text{th}}$  row of the matrix corresponds to `quat(i)`.

### Examples

#### Convert Quaternion Array to Compact Representation of Parts

Create a scalar quaternion with random parts. Convert the parts to a 1-by-4 vector using `compact`.

```
randomParts = randn(1,4)
randomParts = 1×4
    0.5377    1.8339   -2.2588    0.8622

quat = quaternion(randomParts)
quat = quaternion
    0.53767 + 1.8339i - 2.2588j + 0.86217k

quatParts = compact(quat)
quatParts = 1×4
    0.5377    1.8339   -2.2588    0.8622
```

Create a 2-by-2 array of quaternions, then convert the representation to a matrix of quaternion parts. The output rows correspond to the linear indices of the quaternion array.

```
quatArray = [quaternion([1:4;5:8]),quaternion([9:12;13:16])]
quatArray = 2×2 quaternion array
    1 + 2i + 3j + 4k    9 + 10i + 11j + 12k
    5 + 6i + 7j + 8k    13 + 14i + 15j + 16k

quatArrayParts = compact(quatArray)
quatArrayParts = 4×4
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

## Input Arguments

### **quat** — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

## Output Arguments

### **matrix** — Quaternion in matrix form

*N*-by-4 matrix

Quaternion in matrix form, returned as an *N*-by-4 matrix, where *N* = numel(quat).

Data Types: single | double

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

parts | classUnderlying

### **Objects**

quaternion

**Introduced in R2021a**

# conj

Complex conjugate of quaternion

## Syntax

```
quatConjugate = conj(quat)
```

## Description

`quatConjugate = conj(quat)` returns the complex conjugate of the quaternion, `quat`.

If  $q = a + bi + cj + dk$ , the complex conjugate of  $q$  is  $q^* = a - bi - cj - dk$ . Considered as a rotation operator, the conjugate performs the opposite rotation. For example,

```
q = quaternion(deg2rad([16 45 30]), 'rotvec');
a = q*conj(q);
rotatepoint(a,[0,1,0])
```

```
ans =
```

```
    0    1    0
```

## Examples

### Complex Conjugate of Quaternion

Create a quaternion scalar and get the complex conjugate.

```
q = normalize(quaternion([0.9 0.3 0.3 0.25]))
q = quaternion
    0.87727 + 0.29242i + 0.29242j + 0.24369k
```

```
qConj = conj(q)
```

```
qConj = quaternion
    0.87727 - 0.29242i - 0.29242j - 0.24369k
```

Verify that a quaternion multiplied by its conjugate returns a quaternion one.

```
q*qConj
```

```
ans = quaternion
    1 + 0i + 0j + 0k
```

## Input Arguments

### **quat** — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion to conjugate, specified as a scalar, vector, matrix, or array of quaternions.

Data Types: quaternion

## Output Arguments

### **quatConjugate** — Quaternion conjugate

scalar | vector | matrix | multidimensional array

Quaternion conjugate, returned as a quaternion or array of quaternions the same size as `quat`.

Data Types: quaternion

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

`norm` | `.*`, `times`

### **Objects**

quaternion

**Introduced in R2021a**

## ctranspose, '

Complex conjugate transpose of quaternion array

### Syntax

```
quatTransposed = quat'
```

### Description

quatTransposed = quat' returns the complex conjugate transpose of the quaternion, quat.

### Examples

#### Vector Complex Conjugate Transpose

Create a vector of quaternions and compute its complex conjugate transpose.

```
quat = quaternion(randn(4,4))
```

```
quat = 4x1 quaternion array
    0.53767 + 0.31877i + 3.5784j + 0.7254k
    1.8339 - 1.3077i + 2.7694j - 0.063055k
   -2.2588 - 0.43359i - 1.3499j + 0.71474k
    0.86217 + 0.34262i + 3.0349j - 0.20497k
```

```
quatTransposed = quat'
```

```
quatTransposed = 1x4 quaternion array
    0.53767 - 0.31877i - 3.5784j - 0.7254k    1.8339 + 1.3077i - 2.7694j + 0.063055k
   -2.2588 + 0.43359i + 1.3499j - 0.71474k    0.86217 - 0.34262i - 3.0349j + 0.20497k
```

#### Matrix Complex Conjugate Transpose

Create a matrix of quaternions and compute its complex conjugate transpose.

```
quat = [quaternion(randn(2,4)), quaternion(randn(2,4))]
```

```
quat = 2x2 quaternion array
    0.53767 - 2.2588i + 0.31877j - 0.43359k    3.5784 - 1.3499i + 0.7254j + 0.71474k
    1.8339 + 0.86217i - 1.3077j + 0.34262k    2.7694 + 3.0349i - 0.063055j - 0.20497k
```

```
quatTransposed = quat'
```

```
quatTransposed = 2x2 quaternion array
    0.53767 + 2.2588i - 0.31877j + 0.43359k    1.8339 - 0.86217i + 1.3077j - 0.34262k
    3.5784 + 1.3499i - 0.7254j - 0.71474k    2.7694 - 3.0349i + 0.063055j + 0.20497k
```

## Input Arguments

### **quat** — Quaternion to transpose

scalar | vector | matrix

Quaternion to transpose, specified as a vector or matrix or quaternions. The complex conjugate transpose is defined for 1-D and 2-D arrays.

Data Types: quaternion

## Output Arguments

### **quatTransposed** — Conjugate transposed quaternion

scalar | vector | matrix

Conjugate transposed quaternion, returned as an  $N$ -by- $M$  array, where `quat` was specified as an  $M$ -by- $N$  array.

Data Types: quaternion

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

`transpose`, `'`

### **Objects**

quaternion

**Introduced in R2021a**

## dist

Angular distance in radians

### Syntax

```
distance = dist(quatA,quatB)
```

### Description

`distance = dist(quatA,quatB)` returns the angular distance in radians between two quaternions, `quatA` and `quatB`.

### Examples

#### Calculate Quaternion Distance

Calculate the quaternion distance between a single quaternion and each element of a vector of quaternions. Define the quaternions using Euler angles.

```
q = quaternion([0,0,0], 'eulerd', 'zyx', 'frame')
```

```
q = quaternion
    1 + 0i + 0j + 0k
```

```
qArray = quaternion([0,45,0;0,90,0;0,180,0;0,-90,0;0,-45,0], 'eulerd', 'zyx', 'frame')
```

```
qArray = 5x1 quaternion array
    0.92388 +      0i + 0.38268j +      0k
    0.70711 +      0i + 0.70711j +      0k
    6.1232e-17 +      0i +      1j +      0k
    0.70711 +      0i - 0.70711j +      0k
    0.92388 +      0i - 0.38268j +      0k
```

```
quaternionDistance = rad2deg(dist(q,qArray))
```

```
quaternionDistance = 5x1
```

```
45.0000
90.0000
180.0000
90.0000
45.0000
```

If both arguments to `dist` are vectors, the quaternion distance is calculated between corresponding elements. Calculate the quaternion distance between two quaternion vectors.

```
angles1 = [30,0,15; ...
           30,5,15; ...
```

```

        30,10,15; ...
        30,15,15];
angles2 = [30,6,15; ...
        31,11,15; ...
        30,16,14; ...
        30.5,21,15.5];

qVector1 = quaternion(angles1,'eulerd','zyx','frame');
qVector2 = quaternion(angles2,'eulerd','zyx','frame');

rad2deg(dist(qVector1,qVector2))

ans = 4×1

    6.0000
    6.0827
    6.0827
    6.0287

```

Note that a quaternion represents the same rotation as its negative. Calculate a quaternion and its negative.

```

qPositive = quaternion([30,45,-60],'eulerd','zyx','frame')

qPositive = quaternion
    0.72332 - 0.53198i + 0.20056j + 0.3919k

qNegative = -qPositive

qNegative = quaternion
    -0.72332 + 0.53198i - 0.20056j - 0.3919k

```

Find the distance between the quaternion and its negative.

```

dist(qPositive,qNegative)

ans = 0

```

The components of a quaternion may look different from the components of its negative, but both expressions represent the same rotation.

## Input Arguments

### **quatA, quatB** — Quaternions to calculate distance between

scalar | vector | matrix | multidimensional array

Quaternions to calculate distance between, specified as comma-separated quaternions or arrays of quaternions. `quatA` and `quatB` must have compatible sizes:

- `size(quatA) == size(quatB)`, or
- `numel(quatA) == 1`, or
- `numel(quatB) == 1`, or



- if  $[A_{dim1}, \dots, A_{dimN}] = \text{size}(\text{quatA})$  and  $[B_{dim1}, \dots, B_{dimN}] = \text{size}(\text{quatB})$ , then for  $i = 1:N$ , either  $A_{dimi} == B_{dimi}$  or  $A_{dim} == 1$  or  $B_{dim} == 1$ .

If one of the quaternion arguments contains only one quaternion, then this function returns the distances between that quaternion and every quaternion in the other argument.

Data Types: quaternion

## Output Arguments

### distance — Angular distance (radians)

scalar | vector | matrix | multidimensional array

Angular distance in radians, returned as an array. The dimensions are the maximum of the union of  $\text{size}(\text{quatA})$  and  $\text{size}(\text{quatB})$ .

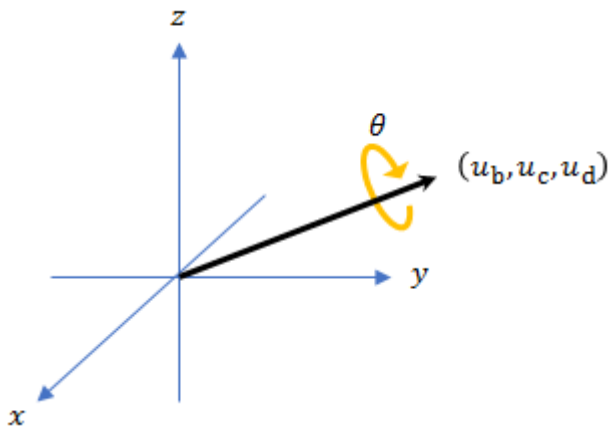
Data Types: single | double

## Algorithms

The `dist` function returns the angular distance between two quaternions.

A quaternion may be defined by an axis  $(u_b, u_c, u_d)$  and angle of rotation  $\theta_q$ :

$$q = \cos\left(\frac{\theta_q}{2}\right) + \sin\left(\frac{\theta_q}{2}\right)(u_b i + u_c j + u_d k).$$



Given a quaternion in the form,  $q = a + bi + cj + dk$ , where  $a$  is the real part, you can solve for the angle of  $q$  as  $\theta_q = 2\cos^{-1}(a)$ .

Consider two quaternions,  $p$  and  $q$ , and the product  $z = p * \text{conjugate}(q)$ . As  $p$  approaches  $q$ , the angle of  $z$  goes to 0, and  $z$  approaches the unit quaternion.

The angular distance between two quaternions can be expressed as  $\theta_z = 2\cos^{-1}(\text{real}(z))$ .

Using the quaternion data type syntax, the angular distance is calculated as:

```
angularDistance = 2*acos(abs(parts(p*conj(q))));
```

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

parts | conj

### **Objects**

quaternion

**Introduced in R2021a**

# euler

Convert quaternion to Euler angles (radians)

## Syntax

```
eulerAngles = euler(quat, rotationSequence, rotationType)
```

## Description

`eulerAngles = euler(quat, rotationSequence, rotationType)` converts the quaternion, `quat`, to an  $N$ -by-3 matrix of Euler angles.

## Examples

### Convert Quaternion to Euler Angles in Radians

Convert a quaternion frame rotation to Euler angles in radians using the 'ZYX' rotation sequence.

```
quat = quaternion([0.7071 0.7071 0 0]);
eulerAnglesRadians = euler(quat, 'ZYX', 'frame')
```

```
eulerAnglesRadians = 1×3
    0         0    1.5708
```

## Input Arguments

### quat — Quaternion to convert to Euler angles

scalar | vector | matrix | multidimensional array

Quaternion to convert to Euler angles, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

### rotationSequence — Rotation sequence

'ZYX' | 'YZX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX' | 'ZYX'

Rotation sequence of Euler representation, specified as a character vector or string.

The rotation sequence defines the order of rotations about the axes. For example, if you specify a rotation sequence of 'YZX':

- 1 The first rotation is about the y-axis.
- 2 The second rotation is about the new z-axis.
- 3 The third rotation is about the new x-axis.

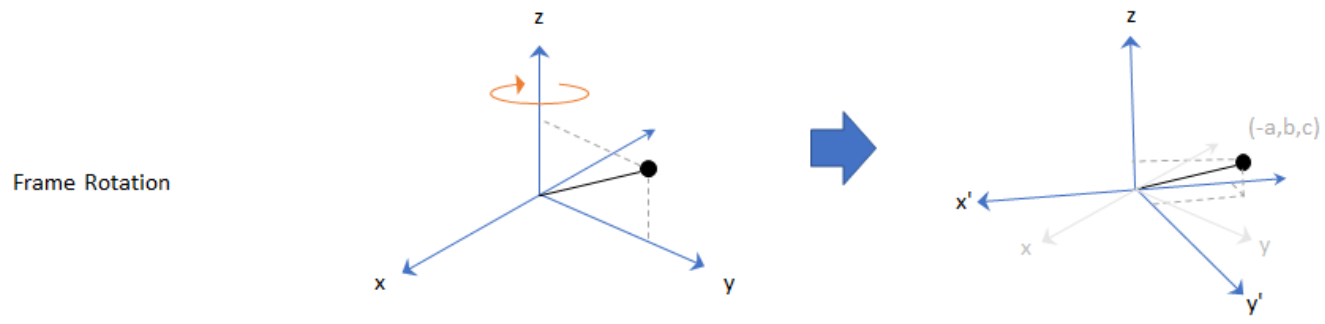
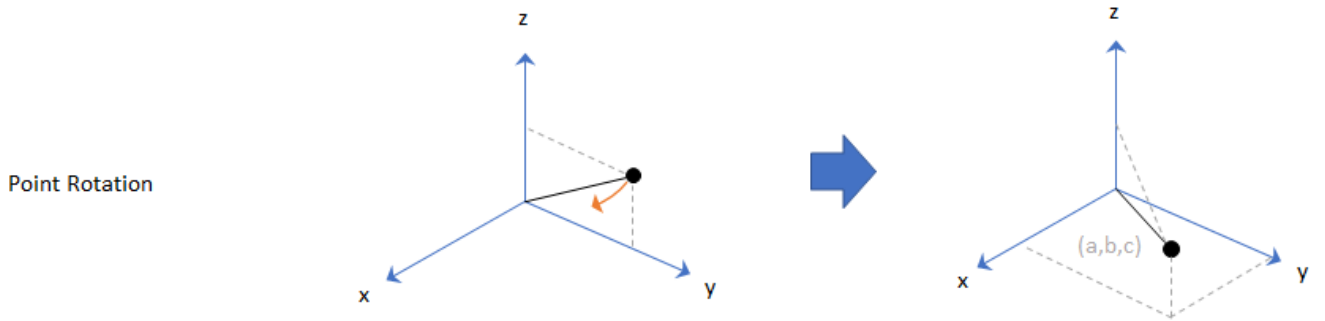
Data Types: char | string

**rotationType — Type of rotation**

'point' | 'frame'

Type of rotation, specified as 'point' or 'frame'.

In a point rotation, the frame is static and the point moves. In a frame rotation, the point is static and the frame moves. Point rotation and frame rotation define equivalent angular displacements but in opposite directions.



Data Types: char | string

**Output Arguments**

**eulerAngles — Euler angle representation (radians)**

*N*-by-3 matrix

Euler angle representation in radians, returned as a *N*-by-3 matrix. *N* is the number of quaternions in the quat argument.

For each row of `eulerAngles`, the first element corresponds to the first axis in the rotation sequence, the second element corresponds to the second axis in the rotation sequence, and the third element corresponds to the third axis in the rotation sequence.

The data type of the Euler angles representation is the same as the underlying data type of `quat`.

Data Types: single | double

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

eulerd | rotateframe | rotatepoint

### Objects

quaternion

**Introduced in R2021a**

## exp

Exponential of quaternion array

### Syntax

`B = exp(A)`

### Description

`B = exp(A)` computes the exponential of the elements of the quaternion array `A`.

### Examples

#### Exponential of Quaternion Array

Create a 4-by-1 quaternion array `A`.

```
A = quaternion(magic(4))
```

```
A = 4x1 quaternion array
    16 + 2i + 3j + 13k
     5 + 11i + 10j + 8k
     9 + 7i + 6j + 12k
     4 + 14i + 15j + 1k
```

Compute the exponential of `A`.

```
B = exp(A)
```

```
B = 4x1 quaternion array
 5.3525e+06 + 1.0516e+06i + 1.5774e+06j + 6.8352e+06k
 -57.359 - 89.189i - 81.081j - 64.865k
 -6799.1 + 2039.1i + 1747.8j + 3495.6k
 -6.66 + 36.931i + 39.569j + 2.6379k
```

### Input Arguments

#### A — Input quaternion

scalar | vector | matrix | multidimensional array

Input quaternion, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

### Output Arguments

#### B — Result

scalar | vector | matrix | multidimensional array

Result of quaternion exponential, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

## Algorithms

Given a quaternion  $A = a + bi + cj + dk = a + \bar{v}$ , the exponential is computed by

$$\exp(A) = e^a \left( \cos\|\bar{v}\| + \frac{\bar{v}}{\|\bar{v}\|} \sin\|\bar{v}\| \right)$$

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

.^, power | log

### Objects

quaternion

**Introduced in R2021a**

## ldivide, ./

Element-wise quaternion left division

### Syntax

```
C = A.\B
```

### Description

`C = A.\B` performs quaternion element-wise division by dividing each element of quaternion B by the corresponding element of quaternion A.

### Examples

#### Divide a Quaternion Array by a Real Scalar

Create a 2-by-1 quaternion array, and divide it element-by-element by a real scalar.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array
    1 + 2i + 3j + 4k
    5 + 6i + 7j + 8k
```

```
B = 2;
C = A.\B
```

```
C = 2x1 quaternion array
    0.066667 - 0.133333i - 0.2j - 0.26667k
    0.057471 - 0.068966i - 0.08046j - 0.091954k
```

#### Divide a Quaternion Array by Another Quaternion Array

Create a 2-by-2 quaternion array, and divide it element-by-element by another 2-by-2 quaternion array.

```
q1 = quaternion([1:4;2:5;4:7;5:8]);
A = reshape(q1,2,2)
```

```
A = 2x2 quaternion array
    1 + 2i + 3j + 4k    4 + 5i + 6j + 7k
    2 + 3i + 4j + 5k    5 + 6i + 7j + 8k
```

```
q2 = quaternion(magic(4));
B = reshape(q2,2,2)
```



B = 2x2 quaternion array

16 + 2i + 3j + 13k	9 + 7i + 6j + 12k
5 + 11i + 10j + 8k	4 + 14i + 15j + 1k

C = A.\B

C = 2x2 quaternion array

2.7 - 1.9i - 0.9j - 1.7k	1.5159 - 0.37302i - 0.15079j - 0.0238k
2.2778 + 0.46296i - 0.57407j + 0.092593k	1.2471 + 0.91379i - 0.33908j - 0.109k

## Input Arguments

### A — Divisor

scalar | vector | matrix | multidimensional array

Divisor, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

### B — Dividend

scalar | vector | matrix | multidimensional array

Dividend, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

## Output Arguments

### C — Result

scalar | vector | matrix | multidimensional array

Result of quaternion division, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

## Algorithms

### Quaternion Division

Given a quaternion  $A = a_1 + a_2i + a_3j + a_4k$  and a real scalar  $p$ ,

$$C = p.\backslash A = \frac{a_1}{p} + \frac{a_2}{p}i + \frac{a_3}{p}j + \frac{a_4}{p}k$$

---

**Note** For a real scalar  $p$ ,  $A./p = A.\backslash p$ .

---

### Quaternion Division by a Quaternion Scalar

Given two quaternions  $A$  and  $B$  of compatible sizes, then

$$C = A.\backslash B = A^{-1} .* B = \left( \frac{\text{conj}(A)}{\text{norm}(A)^2} \right) .* B$$

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`.*`, `times` | `conj` | `norm` | `./`, `ldivide`

### Objects

`quaternion`

**Introduced in R2021a**

# log

Natural logarithm of quaternion array

## Syntax

$B = \log(A)$

## Description

$B = \log(A)$  computes the natural logarithm of the elements of the quaternion array  $A$ .

## Examples

### Logarithmic Values of Quaternion Array

Create a 3-by-1 quaternion array  $A$ .

```
A = quaternion(randn(3,4))
```

```
A = 3x1 quaternion array
    0.53767 + 0.86217i - 0.43359j + 2.7694k
    1.8339 + 0.31877i + 0.34262j - 1.3499k
   -2.2588 - 1.3077i + 3.5784j + 3.0349k
```

Compute the logarithmic values of  $A$ .

```
B = log(A)
```

```
B = 3x1 quaternion array
    1.0925 + 0.40848i - 0.20543j + 1.3121k
    0.8436 + 0.14767i + 0.15872j - 0.62533k
    1.6807 - 0.53829i + 1.473j + 1.2493k
```

## Input Arguments

### A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

## Output Arguments

### B — Logarithm values

scalar | vector | matrix | multidimensional array

Quaternion natural logarithm values, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

## Algorithms

Given a quaternion  $A = a + \bar{v} = a + bi + cj + dk$ , the logarithm is computed by

$$\log(A) = \log\|A\| + \frac{\bar{v}}{\|\bar{v}\|} \arccos \frac{a}{\|A\|}$$

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`exp` | `.^`, power

### Objects

quaternion

**Introduced in R2021a**

# minus, -

Quaternion subtraction

## Syntax

$$C = A - B$$

## Description

$C = A - B$  subtracts quaternion  $B$  from quaternion  $A$  using quaternion subtraction. Either  $A$  or  $B$  may be a real number, in which case subtraction is performed with the real part of the quaternion argument.

## Examples

### Subtract a Quaternion from a Quaternion

Quaternion subtraction is defined as the subtraction of the corresponding parts of each quaternion. Create two quaternions and perform subtraction.

```
Q1 = quaternion([1,0,-2,7]);
Q2 = quaternion([1,2,3,4]);
```

$$Q1minusQ2 = Q1 - Q2$$

```
Q1minusQ2 = quaternion
    0 - 2i - 5j + 3k
```

### Subtract a Real Number from a Quaternion

Addition and subtraction of real numbers is defined for quaternions as acting on the real part of the quaternion. Create a quaternion and then subtract 1 from the real part.

$$Q = \text{quaternion}([1,1,1,1])$$

```
Q = quaternion
    1 + 1i + 1j + 1k
```

$$Qminus1 = Q - 1$$

```
Qminus1 = quaternion
    0 + 1i + 1j + 1k
```

## Input Arguments

### A — Input

scalar | vector | matrix | multidimensional array

Input, specified as a quaternion, array of quaternions, real number, or array of real numbers.

Data Types: quaternion | single | double

### B — Input

scalar | vector | matrix | multidimensional array

Input, specified as a quaternion, array of quaternions, real number, or array of real numbers.

Data Types: quaternion | single | double

## Output Arguments

### C — Result

scalar | vector | matrix | multidimensional array

Result of quaternion subtraction, returned as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

-, uminus | .\*, times | \*, mtimes

### Objects

quaternion

**Introduced in R2021a**

## mtimes, \*

Quaternion multiplication

### Syntax

```
quatC = A*B
```

### Description

quatC = A\*B implements quaternion multiplication if either A or B is a quaternion. Either A or B must be a scalar.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the order of the desired sequence of rotations. For example, to apply a  $p$  quaternion followed by a  $q$  quaternion, multiply in the order  $pq$ . The rotation operator becomes  $(pq)^*v(pq)$ , where  $v$  represents the object to rotate specified in quaternion form.  $*$  represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a  $p$  quaternion followed by a  $q$  quaternion, multiply in the reverse order,  $qp$ . The rotation operator becomes  $(qp)v(qp)^*$ .

### Examples

#### Multiply Quaternion Scalar and Quaternion Vector

Create a 4-by-1 column vector, A, and a scalar, b. Multiply A times b.

```
A = quaternion(randn(4,4))
```

```
A = 4x1 quaternion array
    0.53767 + 0.31877i + 3.5784j + 0.7254k
    1.8339 - 1.3077i + 2.7694j - 0.063055k
   -2.2588 - 0.43359i - 1.3499j + 0.71474k
    0.86217 + 0.34262i + 3.0349j - 0.20497k
```

```
b = quaternion(randn(1,4))
```

```
b = quaternion
   -0.12414 + 1.4897i + 1.409j + 1.4172k
```

```
C = A*b
```

```
C = 4x1 quaternion array
   -6.6117 + 4.8105i + 0.94224j - 4.2097k
   -2.0925 + 6.9079i + 3.9995j - 3.3614k
    1.8155 - 6.2313i - 1.336j - 1.89k
   -4.6033 + 5.8317i + 0.047161j - 2.791k
```

## Input Arguments

### A — Input

scalar | vector | matrix | multidimensional array

Input to multiply, specified as a quaternion, array of quaternions, real scalar, or array of real scalars.

If B is nonscalar, then A must be scalar.

Data Types: quaternion | single | double

### B — Input

scalar | vector | matrix | multidimensional array

Input to multiply, specified as a quaternion, array of quaternions, real scalar, or array of real scalars.

If A is nonscalar, then B must be scalar.

Data Types: quaternion | single | double

## Output Arguments

### quatC — Quaternion product

scalar | vector | matrix | multidimensional array

Quaternion product, returned as a quaternion or array of quaternions.

Data Types: quaternion

## Algorithms

### Quaternion Multiplication by a Real Scalar

Given a quaternion

$$q = a_q + b_q i + c_q j + d_q k,$$

the product of  $q$  and a real scalar  $\beta$  is

$$\beta q = \beta a_q + \beta b_q i + \beta c_q j + \beta d_q k$$

### Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$i^2 = j^2 = k^2 = ijk = -1,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	<b>1</b>	<b>i</b>	<b>j</b>	<b>k</b>
<b>1</b>	1	i	j	k
<b>i</b>	i	-1	k	-j



<b>j</b>	j	-k	-1	i
<b>k</b>	k	j	-i	-1

When reading the table, the rows are read first, for example:  $ij = k$  and  $ji = -k$ .

Given two quaternions,  $q = a_q + b_q i + c_q j + d_q k$ , and  $p = a_p + b_p i + c_p j + d_p k$ , the multiplication can be expanded as:

$$\begin{aligned}
 z = pq &= (a_p + b_p i + c_p j + d_p k)(a_q + b_q i + c_q j + d_q k) \\
 &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\
 &\quad + b_p a_q i + b_p b_q i^2 + b_p c_q ij + b_p d_q ik \\
 &\quad + c_p a_q j + c_p b_q ji + c_p c_q j^2 + c_p d_q jk \\
 &\quad + d_p a_q k + d_p b_q ki + d_p c_q kj + d_p d_q k^2
 \end{aligned}$$

You can simplify the equation using the quaternion multiplication table:

$$\begin{aligned}
 z = pq &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\
 &\quad + b_p a_q i - b_p b_q + b_p c_q k - b_p d_q j \\
 &\quad + c_p a_q j - c_p b_q k - c_p c_q + c_p d_q i \\
 &\quad + d_p a_q k + d_p b_q j - d_p c_q i - d_p d_q
 \end{aligned}$$

## References

- [1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

.\*, times

### Objects

quaternion

Introduced in R2021a

## norm

Quaternion norm

### Syntax

```
N = norm(quat)
```

### Description

`N = norm(quat)` returns the norm of the quaternion, `quat`.

Given a quaternion of the form  $Q = a + bi + cj + dk$ , the norm of the quaternion is defined as  $\text{norm}(Q) = \sqrt{a^2 + b^2 + c^2 + d^2}$ .

### Examples

#### Calculate Quaternion Norm

Create a scalar quaternion and calculate its norm.

```
quat = quaternion(1,2,3,4);  
norm(quat)
```

```
ans = 5.4772
```

The quaternion norm is defined as the square root of the sum of the quaternion parts squared. Calculate the quaternion norm explicitly to verify the result of the `norm` function.

```
[a,b,c,d] = parts(quat);  
sqrt(a^2+b^2+c^2+d^2)
```

```
ans = 5.4772
```

### Input Arguments

#### quat — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion for which to calculate the norm, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

### Output Arguments

#### N — Quaternion norm

scalar | vector | matrix | multidimensional array

Quaternion norm. If the input `quat` is an array, the output is returned as an array the same size as `quat`. Elements of the array are real numbers with the same data type as the underlying data type of the quaternion, `quat`.

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`normalize` | `parts` | `conj`

### Objects

`quaternion`

**Introduced in R2021a**

# normalize

Quaternion normalization

## Syntax

```
quatNormalized = normalize(quat)
```

## Description

`quatNormalized = normalize(quat)` normalizes the quaternion.

Given a quaternion of the form  $Q = a + bi + cj + dk$ , the normalized quaternion is defined as  $Q/\sqrt{a^2 + b^2 + c^2 + d^2}$ .

## Examples

### Normalize Elements of Quaternion Vector

Quaternions can represent rotations when normalized. You can use `normalize` to normalize a scalar, elements of a matrix, or elements of a multi-dimensional array of quaternions. Create a column vector of quaternions, then normalize them.

```
quatArray = quaternion([1,2,3,4; ...  
                      2,3,4,1; ...  
                      3,4,1,2]);  
quatArrayNormalized = normalize(quatArray)  
  
quatArrayNormalized = 3x1 quaternion array  
    0.18257 + 0.36515i + 0.54772j + 0.7303k  
    0.36515 + 0.54772i + 0.7303j + 0.18257k  
    0.54772 + 0.7303i + 0.18257j + 0.36515k
```

## Input Arguments

### quat — Quaternion to normalize

scalar | vector | matrix | multidimensional array

Quaternion to normalize, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: quaternion

## Output Arguments

### quatNormalized — Normalized quaternion

scalar | vector | matrix | multidimensional array

Normalized quaternion, returned as a quaternion or array of quaternions the same size as `quat`.

Data Types: `quaternion`

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`norm` | `.*`, `times` | `conj`

### Objects

`quaternion`

**Introduced in R2021a**

## ones

Create quaternion array with real parts set to one and imaginary parts set to zero

### Syntax

```
quat0nes = ones('quaternion')
quat0nes = ones(n,'quaternion')
quat0nes = ones(sz,'quaternion')
quat0nes = ones(sz1,...,szN,'quaternion')

quat0nes = ones( ____, 'like', prototype, 'quaternion')
```

### Description

`quat0nes = ones('quaternion')` returns a scalar quaternion with the real part set to 1 and the imaginary parts set to 0.

Given a quaternion of the form  $Q = a + bi + cj + dk$ , a quaternion one is defined as  $Q = 1 + 0i + 0j + 0k$ .

`quat0nes = ones(n,'quaternion')` returns an n-by-n quaternion matrix with the real parts set to 1 and the imaginary parts set to 0.

`quat0nes = ones(sz,'quaternion')` returns an array of quaternion ones where the size vector, `sz`, defines `size(q0nes)`.

Example: `ones([1,4,2],'quaternion')` returns a 1-by-4-by-2 array of quaternions with the real parts set to 1 and the imaginary parts set to 0.

`quat0nes = ones(sz1,...,szN,'quaternion')` returns a `sz1`-by-...-by-`szN` array of ones where `sz1,...,szN` indicates the size of each dimension.

`quat0nes = ones( ____, 'like', prototype, 'quaternion')` specifies the underlying class of the returned quaternion array to be the same as the underlying class of the quaternion prototype.

### Examples

#### Quaternion Scalar One

Create a quaternion scalar one.

```
quat0nes = ones('quaternion')

quat0nes = quaternion
          1 + 0i + 0j + 0k
```

## Square Matrix of Quaternion Ones

Create an n-by-n matrix of quaternion ones.

```
n = 3;
quat0nes = ones(n, 'quaternion')

quat0nes = 3x3 quaternion array
    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k
    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k
    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k
```

## Multidimensional Array of Quaternion Ones

Create a multidimensional array of quaternion ones by defining array dimensions in order. In this example, you create a 3-by-1-by-2 array. You can specify dimensions using a row vector or comma-separated integers. Specify the dimensions using a row vector and display the results:

```
dims = [3,1,2];
quat0nesSyntax1 = ones(dims, 'quaternion')
```

```
quat0nesSyntax1 = 3x1x2 quaternion array
quat0nesSyntax1(:,:,1) =
```

```
    1 + 0i + 0j + 0k
    1 + 0i + 0j + 0k
    1 + 0i + 0j + 0k
```

```
quat0nesSyntax1(:,:,2) =
```

```
    1 + 0i + 0j + 0k
    1 + 0i + 0j + 0k
    1 + 0i + 0j + 0k
```

Specify the dimensions using comma-separated integers, and then verify the equivalency of the two syntaxes:

```
quat0nesSyntax2 = ones(3,1,2, 'quaternion');
isequal(quat0nesSyntax1, quat0nesSyntax2)
```

```
ans = logical
     1
```

## Underlying Class of Quaternion Ones

A quaternion is a four-part hyper-complex number used in three-dimensional rotations and orientations. You can specify the underlying data type of the parts as `single` or `double`. The default is `double`.

Create a quaternion array of ones with the underlying data type set to `single`.

```
quatOnes = ones(2,'like',single(1),'quaternion')
```

```
quatOnes = 2x2 quaternion array
    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k
    1 + 0i + 0j + 0k    1 + 0i + 0j + 0k
```

Verify the underlying class using the `classUnderlying` function.

```
classUnderlying(quatOnes)
```

```
ans =
'single'
```

## Input Arguments

### **n** — Size of square quaternion matrix

integer value

Size of square quaternion matrix, specified as an integer value.

If `n` is zero or negative, then `quatOnes` is returned as an empty matrix.

Example: `ones(4,'quaternion')` returns a 4-by-4 matrix of quaternions with the real parts set to 1 and the imaginary parts set to 0.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **sz** — Output size

row vector of integer values

Output size, specified as a row vector of integer values. Each element of `sz` indicates the size of the corresponding dimension in `quatOnes`. If the size of any dimension is 0 or negative, then `quatOnes` is returned as an empty array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **prototype** — Quaternion prototype

variable

Quaternion prototype, specified as a variable.

Example: `ones(2,'like',quat,'quaternion')` returns a 2-by-2 matrix of quaternions with the same underlying class as the prototype quaternion, `quat`.

Data Types: `quaternion`

### **sz1, ..., szN** — Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integers. If the size of any dimension is 0 or negative, then `quatOnes` is returned as an empty array.

Example: `ones(2,3,'quaternion')` returns a 2-by-3 matrix of quaternions with the real parts set to 1 and the imaginary parts set to 0.



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### **quat0nes** — Quaternion ones

scalar | vector | matrix | multidimensional array

Quaternion ones, returned as a scalar, vector, matrix, or multidimensional array of quaternions.

Given a quaternion of the form  $Q = a + bi + cj + dk$ , a quaternion one is defined as  $Q = 1 + 0i + 0j + 0k$ .

Data Types: `quaternion`

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

`zeros`

### **Objects**

`quaternion`

**Introduced in R2021a**

## parts

Extract quaternion parts

### Syntax

```
[a,b,c,d] = parts(quat)
```

### Description

`[a,b,c,d] = parts(quat)` returns the parts of the quaternion array as arrays, each the same size as `quat`.

### Examples

#### Convert Quaternion to Matrix of Quaternion Parts

Convert a quaternion representation to parts using the `parts` function.

Create a two-element column vector of quaternions by specifying the parts.

```
quat = quaternion([1:4;5:8])  
  
quat = 2x1 quaternion array  
    1 + 2i + 3j + 4k  
    5 + 6i + 7j + 8k
```

Recover the parts from the quaternion matrix using the `parts` function. The parts are returned as separate output arguments, each the same size as the input 2-by-1 column vector of quaternions.

```
[qA,qB,qC,qD] = parts(quat)
```

```
qA = 2x1
```

```
    1  
    5
```

```
qB = 2x1
```

```
    2  
    6
```

```
qC = 2x1
```

```
    3  
    7
```

```
qD = 2x1
```

4  
8

## Input Arguments

### **quat** — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion, specified as a quaternion or array of quaternions.

Data Types: quaternion

## Output Arguments

### **[a, b, c, d]** — Quaternion parts

scalar | vector | matrix | multidimensional array

Quaternion parts, returned as four arrays: a, b, c, and d. Each part is the same size as quat.

Data Types: single | double

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

classUnderlying | compact

### **Objects**

quaternion

### **Introduced in R2021a**

## power, .^

Element-wise quaternion power

### Syntax

```
C = A.^b
```

### Description

$C = A.^b$  raises each element of  $A$  to the corresponding power in  $b$ .

### Examples

#### Raise a Quaternion to a Real Scalar Power

Create a quaternion and raise it to a real scalar power.

```
A = quaternion(1,2,3,4)
```

```
A = quaternion  
    1 + 2i + 3j + 4k
```

```
b = 3;  
C = A.^b
```

```
C = quaternion  
   -86 - 52i - 78j - 104k
```

#### Raise a Quaternion Array to Powers from a Multidimensional Array

Create a 2-by-1 quaternion array and raise it to powers from a 2-D array.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array  
    1 + 2i + 3j + 4k  
    5 + 6i + 7j + 8k
```

```
b = [1 0 2; 3 2 1]
```

```
b = 2x3
```

```
    1    0    2  
    3    2    1
```

```
C = A.^b
```

$C = 2 \times 3$  quaternion array

1 +	2i +	3j +	4k	1 +	0i +	0j +	0k	-28 +	4i +	6j +
-2110 -	444i -	518j -	592k	-124 +	60i +	70j +	80k	5 +	6i +	7j +

## Input Arguments

### A — Base

scalar | vector | matrix | multidimensional array

Base, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion | single | double

### b — Exponent

scalar | vector | matrix | multidimensional array

Exponent, specified as a real scalar, vector, matrix, or multidimensional array.

Data Types: single | double

## Output Arguments

### C — Result

scalar | vector | matrix | multidimensional array

Each element of quaternion  $A$  raised to the corresponding power in  $b$ , returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

## Algorithms

The polar representation of a quaternion  $A = a + bi + cj + dk$  is given by

$$A = \|A\|(\cos\theta + \hat{u}\sin\theta)$$

where  $\theta$  is the angle of rotation, and  $\hat{u}$  is the unit quaternion.

Quaternion  $A$  raised by a real exponent  $b$  is given by

$$P = A.^b = \|A\|^b(\cos(b\theta) + \hat{u}\sin(b\theta))$$

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

log | exp

**Objects**

quaternion

**Introduced in R2021a**

## prod

Product of a quaternion array

### Syntax

```
quatProd = prod(quat)
quatProd = prod(quat,dim)
```

### Description

`quatProd = prod(quat)` returns the quaternion product of the elements of the array.

`quatProd = prod(quat,dim)` calculates the quaternion product along dimension `dim`.

### Examples

#### Product of Quaternions in Each Column

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A = reshape(quaternion(randn(9,4)),3,3)
```

```
A = 3x3 quaternion array
    0.53767 + 2.7694i + 1.409j - 0.30344k    0.86217 + 0.7254i - 1.2075j + 0.888
    1.8339 - 1.3499i + 1.4172j + 0.29387k    0.31877 - 0.063055i + 0.71724j - 1.147
    -2.2588 + 3.0349i + 0.6715j - 0.78728k    -1.3077 + 0.71474i + 1.6302j - 1.068
```

Find the product of the quaternions in each column. The length of the first dimension is 1, and the length of the second dimension matches `size(A,2)`.

```
B = prod(A)
```

```
B = 1x3 quaternion array
    -19.837 - 9.1521i + 15.813j - 19.918k    -5.4708 - 0.28535i + 3.077j - 1.2295k
```

#### Product of Specified Dimension of Quaternion Array

You can specify which dimension of a quaternion array to take the product of.

Create a 2-by-2-by-2 quaternion array.

```
A = reshape(quaternion(randn(8,4)),2,2,2);
```

Find the product of the elements in each page of the array. The length of the first dimension matches `size(A,1)`, the length of the second dimension matches `size(A,2)`, and the length of the third dimension is 1.

```
dim = 3;
B = prod(A,dim)

B = 2x2 quaternion array
    -2.4847 + 1.1659i - 0.37547j + 2.8068k    0.28786 - 0.29876i - 0.51231j - 4.2972k
    0.38986 - 3.6606i - 2.0474j - 6.047k    -1.741 - 0.26782i + 5.4346j + 4.1452k
```

## Input Arguments

### **quat** — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Example: `qProd = prod(quat)` calculates the quaternion product along the first non-singleton dimension of `quat`.

Data Types: quaternion

### **dim** — Dimension

first non-singleton dimension (default) | positive integer

Dimension along which to calculate the quaternion product, specified as a positive integer. If `dim` is not specified, `prod` operates along the first non-singleton dimension of `quat`.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Output Arguments

### **quatProd** — Quaternion product

positive integer

Quaternion product, returned as quaternion array with one less non-singleton dimension than `quat`.

For example, if `quat` is a 2-by-2-by-5 array,

- `prod(quat,1)` returns a 1-by-2-by-5 array.
- `prod(quat,2)` returns a 2-by-1-by-5 array.
- `prod(quat,3)` returns a 2-by-2 array.

Data Types: quaternion

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

`mtimes` | `.*`, `times`



**Objects**

quaternion

**Introduced in R2021a**

## rdivide, ./

Element-wise quaternion right division

### Syntax

```
C = A./B
```

### Description

`C = A./B` performs quaternion element-wise division by dividing each element of quaternion `A` by the corresponding element of quaternion `B`.

### Examples

#### Divide a Quaternion Array by a Real Scalar

Create a 2-by-1 quaternion array, and divide it element-by-element by a real scalar.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array
    1 + 2i + 3j + 4k
    5 + 6i + 7j + 8k
```

```
B = 2;
C = A./B
```

```
C = 2x1 quaternion array
    0.5 + 1i + 1.5j + 2k
    2.5 + 3i + 3.5j + 4k
```

#### Divide a Quaternion Array by Another Quaternion Array

Create a 2-by-2 quaternion array, and divide it element-by-element by another 2-by-2 quaternion array.

```
q1 = quaternion(magic(4));
A = reshape(q1,2,2)
```

```
A = 2x2 quaternion array
    16 + 2i + 3j + 13k    9 + 7i + 6j + 12k
    5 + 11i + 10j + 8k   4 + 14i + 15j + 1k
```

```
q2 = quaternion([1:4;3:6;2:5;4:7]);
B = reshape(q2,2,2)
```

B = 2x2 quaternion array

$$\begin{array}{cc} 1 + 2i + 3j + 4k & 2 + 3i + 4j + 5k \\ 3 + 4i + 5j + 6k & 4 + 5i + 6j + 7k \end{array}$$

C = A./B

C = 2x2 quaternion array

$$\begin{array}{cccc} 2.7 - & 0.1i - & 2.1j - & 1.7k \\ 1.8256 - 0.081395i + & 0.45349j - & 0.24419k & 2.2778 + 0.092593i - 0.46296j - 0.5740 \\ & & & 1.4524 - 0.5i + 1.0238j - 0.2619 \end{array}$$

## Input Arguments

### A — Dividend

scalar | vector | matrix | multidimensional array

Dividend, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

### B — Divisor

scalar | vector | matrix | multidimensional array

Divisor, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of the dimensions is 1.

Data Types: quaternion | single | double

## Output Arguments

### C — Result

scalar | vector | matrix | multidimensional array

Result of quaternion division, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

## Algorithms

### Quaternion Division

Given a quaternion  $A = a_1 + a_2i + a_3j + a_4k$  and a real scalar  $p$ ,

$$C = A ./ p = \frac{a_1}{p} + \frac{a_2}{p}i + \frac{a_3}{p}j + \frac{a_4}{p}k$$

---

**Note** For a real scalar  $p$ ,  $A./p = A.\backslash p$ .

---

### Quaternion Division by a Quaternion Scalar

Given two quaternions  $A$  and  $B$  of compatible sizes,

$$C = A ./ B = A .* B^{-1} = A .* \left( \frac{\text{conj}(B)}{\text{norm}(B)^2} \right)$$

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`conj` | `./`, `ldivide` | `norm` | `.*`, `times`

### Objects

`quaternion`

**Introduced in R2021a**

# rotateframe

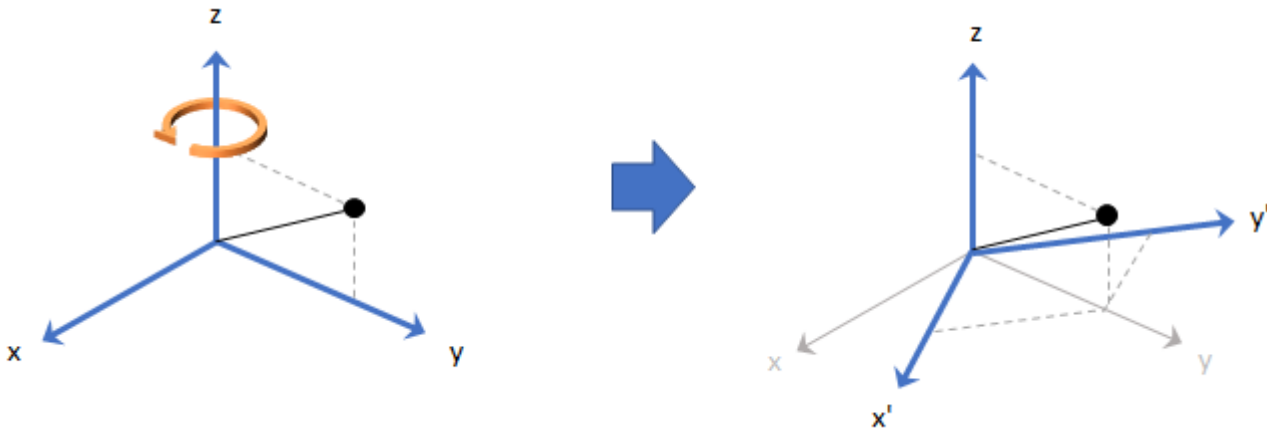
Quaternion frame rotation

## Syntax

```
rotationResult = rotateframe(quat, cartesianPoints)
```

## Description

`rotationResult = rotateframe(quat, cartesianPoints)` rotates the frame of reference for the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.

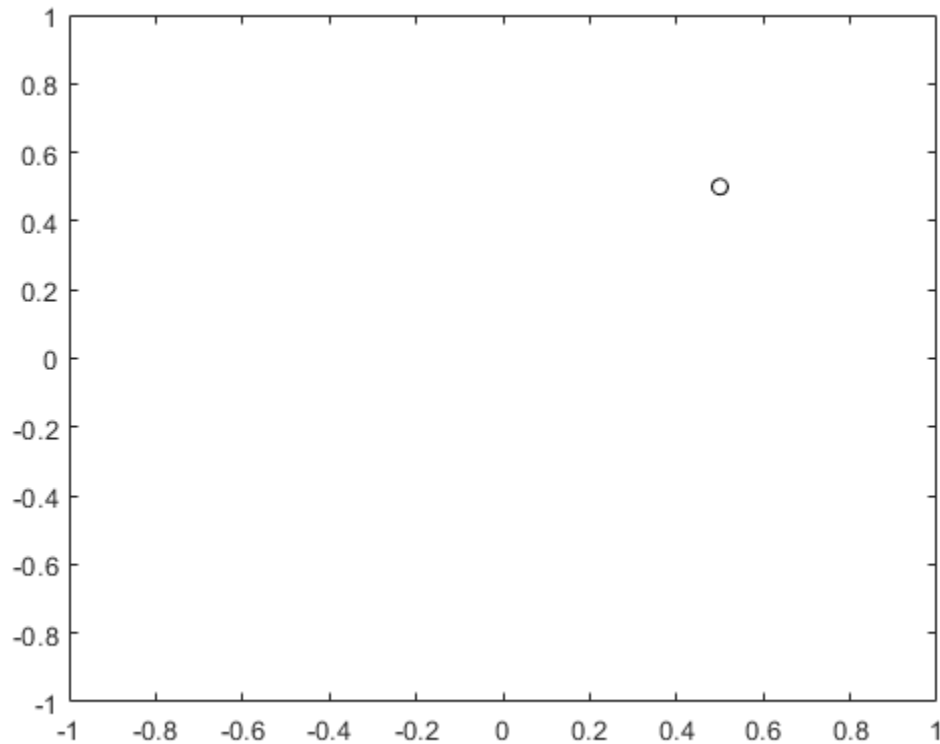


## Examples

### Rotate Frame Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in the order  $x$ ,  $y$ , and  $z$ . For convenient visualization, define the point on the  $x$ - $y$  plane.

```
x = 0.5;
y = 0.5;
z = 0;
plot(x,y, 'ko')
hold on
axis([-1 1 -1 1])
```



Create a quaternion vector specifying two separate rotations, one to rotate the frame 45 degrees and another to rotate the point -90 degrees about the z-axis. Use `rotateframe` to perform the rotations.

```
quat = quaternion([0,0,pi/4; ...
                  0,0,-pi/2], 'euler', 'XYZ', 'frame');
```

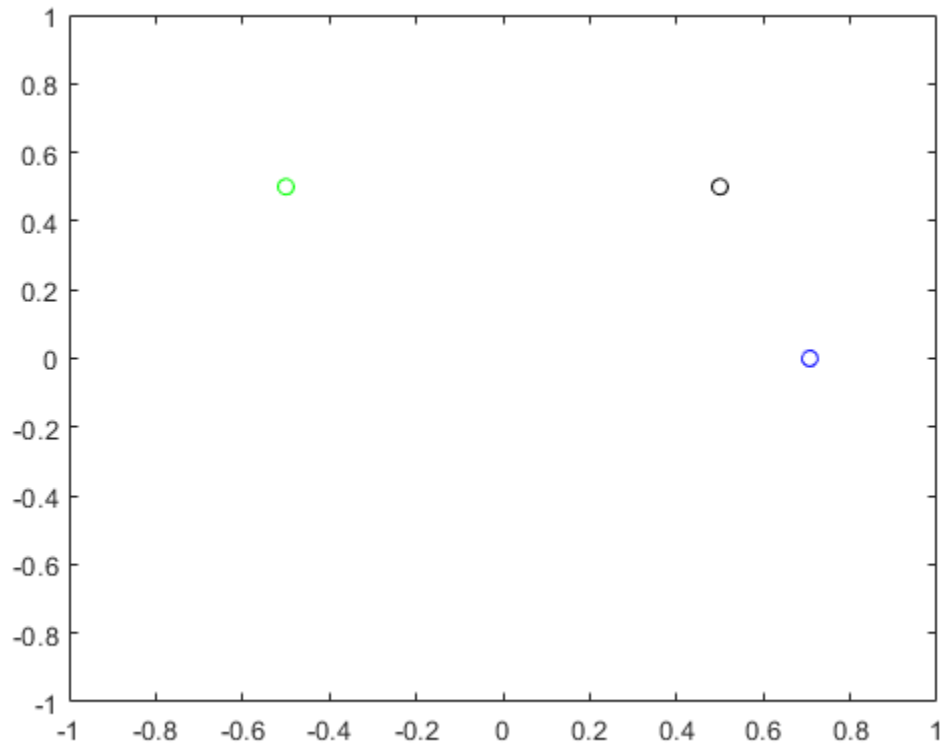
```
rereferencedPoint = rotateframe(quat, [x,y,z])
```

```
rereferencedPoint = 2x3
```

```
    0.7071    -0.0000         0
   -0.5000     0.5000         0
```

Plot the rereferenced points.

```
plot(rereferencedPoint(1,1),rereferencedPoint(1,2),'bo')
plot(rereferencedPoint(2,1),rereferencedPoint(2,2),'go')
```



### Rereference Group of Points using Quaternion

Define two points in three-dimensional space. Define a quaternion to rereference the points by first rotating the reference frame about the z-axis 30 degrees and then about the new y-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0], 'eulerd', 'ZYX', 'point');
```

Use rotateframe to reference both points using the quaternion rotation operator. Display the result.

```
rP = rotateframe(quat, [a;b])
rP = 2x3
    0.6124    -0.3536    0.7071
    0.5000     0.8660   -0.0000
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3), 'bo');
hold on
```

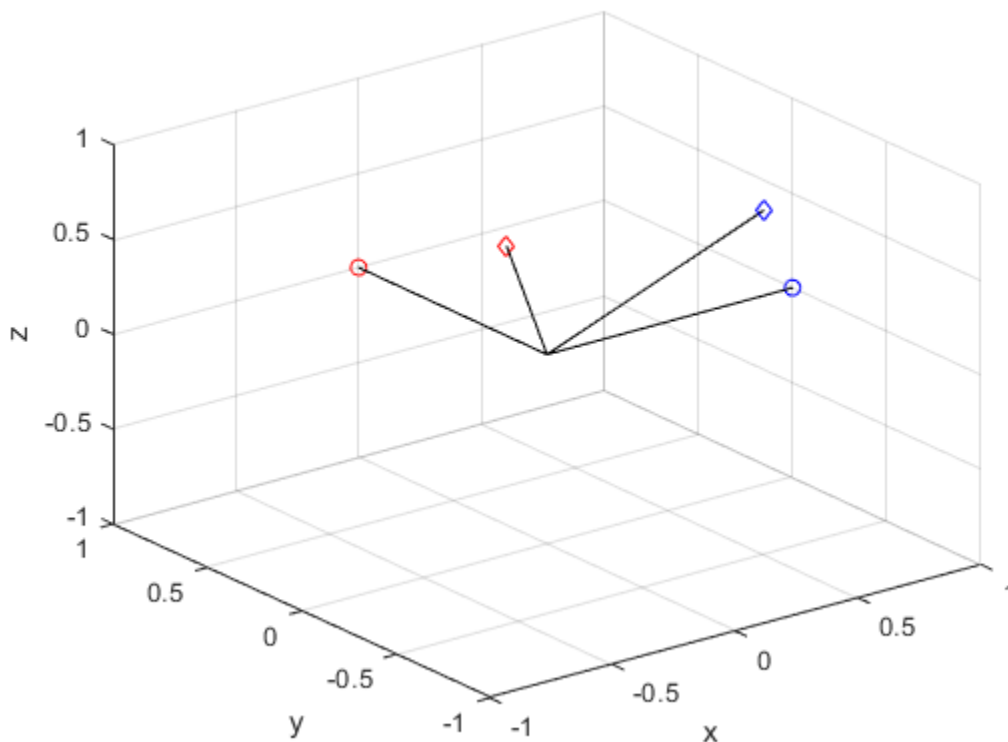
```

grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3), 'ro');
plot3(rP(1,1),rP(1,2),rP(1,3), 'bd')
plot3(rP(2,1),rP(2,2),rP(2,3), 'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)], 'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)], 'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)], 'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)], 'k')

```



## Input Arguments

### quat — Quaternion that defines rotation

scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion or vector of quaternions.

Data Types: quaternion

### cartesianPoints — Three-dimensional Cartesian points

1-by-3 vector |  $N$ -by-3 matrix



Three-dimensional Cartesian points, specified as a 1-by-3 vector or  $N$ -by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **rotationResult** — Re-referenced Cartesian points

vector | matrix

Cartesian points defined in reference to rotated reference frame, returned as a vector or matrix the same size as `cartesianPoints`.

The data type of the re-referenced Cartesian points is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

## Algorithms

Quaternion frame rotation re-references a point specified in  $\mathbf{R}^3$  by rotating the original frame of reference according to a specified quaternion:

$$L_q(u) = q*uq$$

where  $q$  is the quaternion,  $*$  represents conjugation, and  $u$  is the point to rotate, specified as a quaternion.

For convenience, the `rotateframe` function takes a point in  $\mathbf{R}^3$  and returns a point in  $\mathbf{R}^3$ . Given a function call with some arbitrary quaternion,  $q = a + bi + cj + dk$ , and arbitrary coordinate,  $[x,y,z]$ ,

```
point = [x,y,z];
rereferencedPoint = rotateframe(q,point)
```

the `rotateframe` function performs the following operations:

- 1 Converts point  $[x,y,z]$  to a quaternion:

$$u_q = 0 + xi + yj + zk$$

- 2 Normalizes the quaternion,  $q$ :

$$q_n = \frac{q}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

- 3 Applies the rotation:

$$v_q = q*u_qq$$

- 4 Converts the quaternion output,  $v_q$ , back to  $\mathbf{R}^3$

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

rotatepoint

### **Objects**

quaternion

**Introduced in R2021a**

# rotatepoint

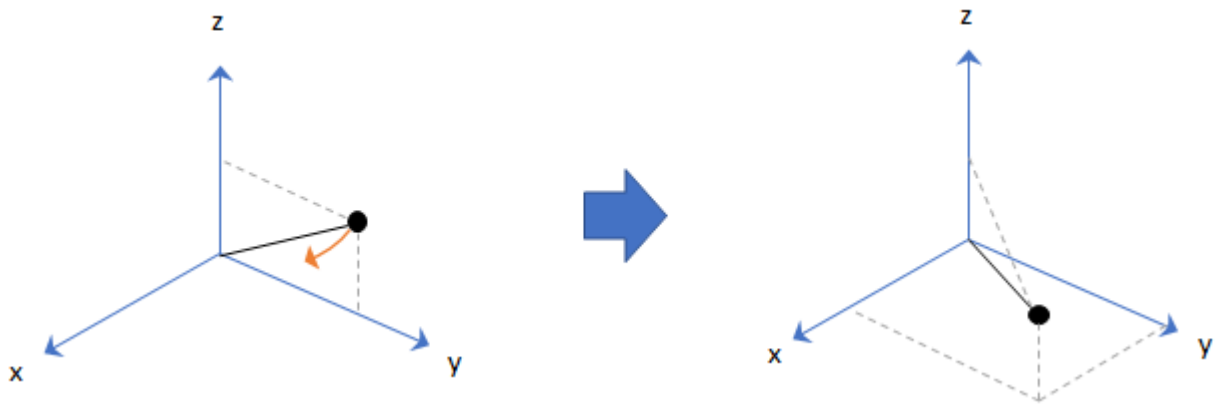
Quaternion point rotation

## Syntax

```
rotationResult = rotatepoint(quat, cartesianPoints)
```

## Description

`rotationResult = rotatepoint(quat, cartesianPoints)` rotates the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.

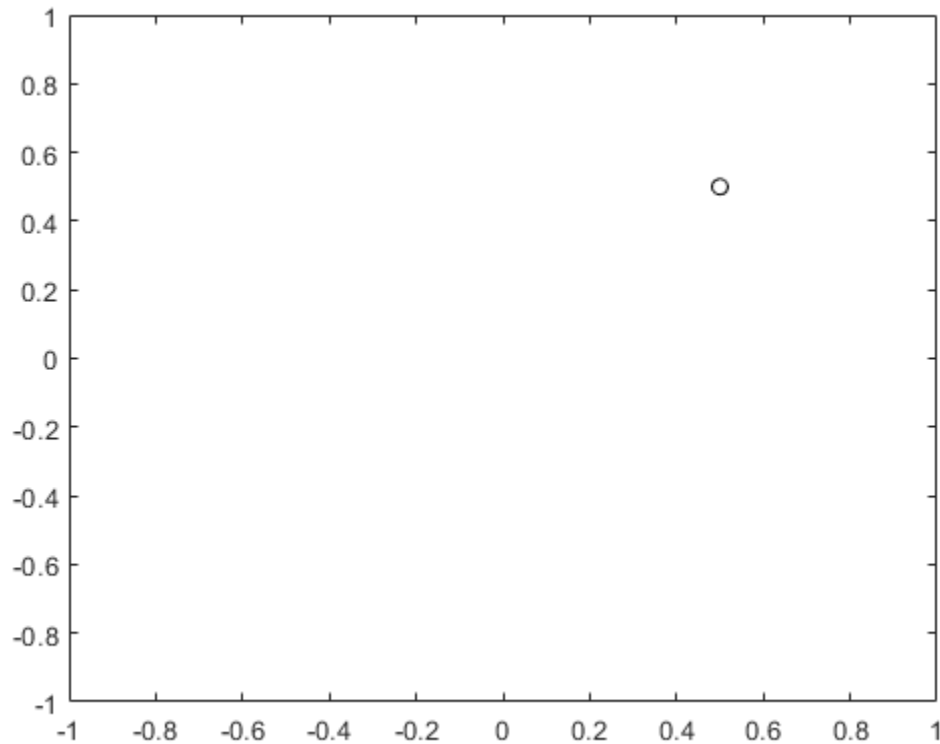


## Examples

### Rotate Point Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in order `x`, `y`, `z`. For convenient visualization, define the point on the `x-y` plane.

```
x = 0.5;  
y = 0.5;  
z = 0;  
  
plot(x,y, 'ko')  
hold on  
axis([-1 1 -1 1])
```



Create a quaternion vector specifying two separate rotations, one to rotate the point 45 and another to rotate the point -90 degrees about the z-axis. Use `rotatepoint` to perform the rotation.

```
quat = quaternion([0,0,pi/4; ...
                  0,0,-pi/2], 'euler', 'XYZ', 'point');
```

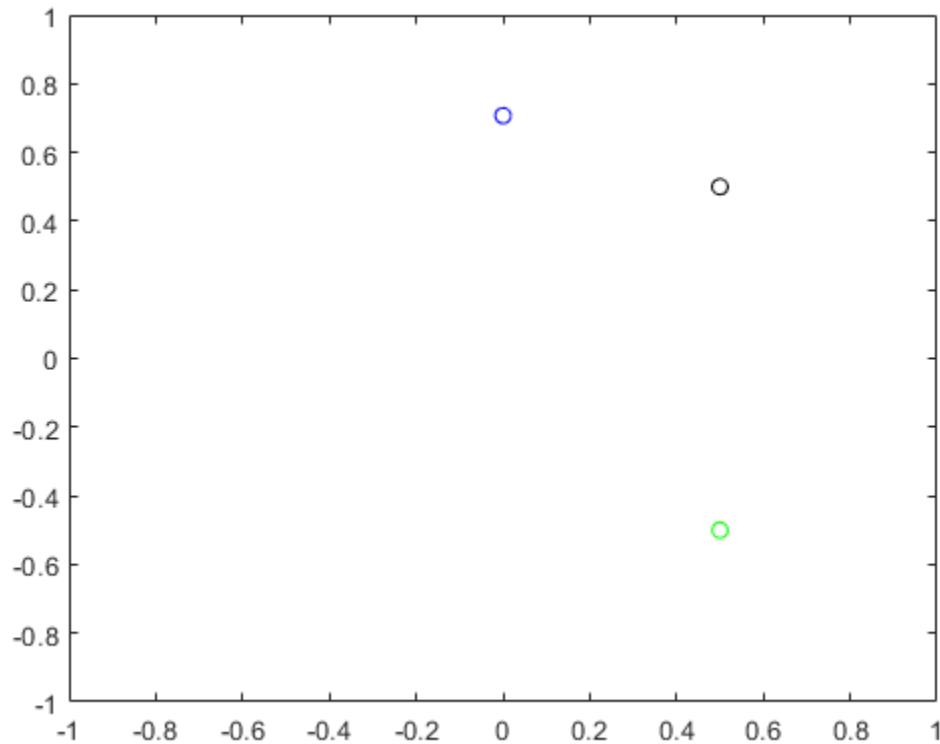
```
rotatedPoint = rotatepoint(quat,[x,y,z])
```

```
rotatedPoint = 2×3
```

```
-0.0000    0.7071    0
 0.5000   -0.5000    0
```

Plot the rotated points.

```
plot(rotatedPoint(1,1),rotatedPoint(1,2), 'bo')
plot(rotatedPoint(2,1),rotatedPoint(2,2), 'go')
```



### Rotate Group of Points Using Quaternion

Define two points in three-dimensional space. Define a quaternion to rotate the point by first rotating about the z-axis 30 degrees and then about the new y-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0], 'eulerd', 'ZYX', 'point');
```

Use rotatepoint to rotate both points using the quaternion rotation operator. Display the result.

```
rP = rotatepoint(quat,[a;b])
rP = 2×3
    0.6124    0.5000   -0.6124
   -0.3536    0.8660    0.3536
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3), 'bo');
hold on
```

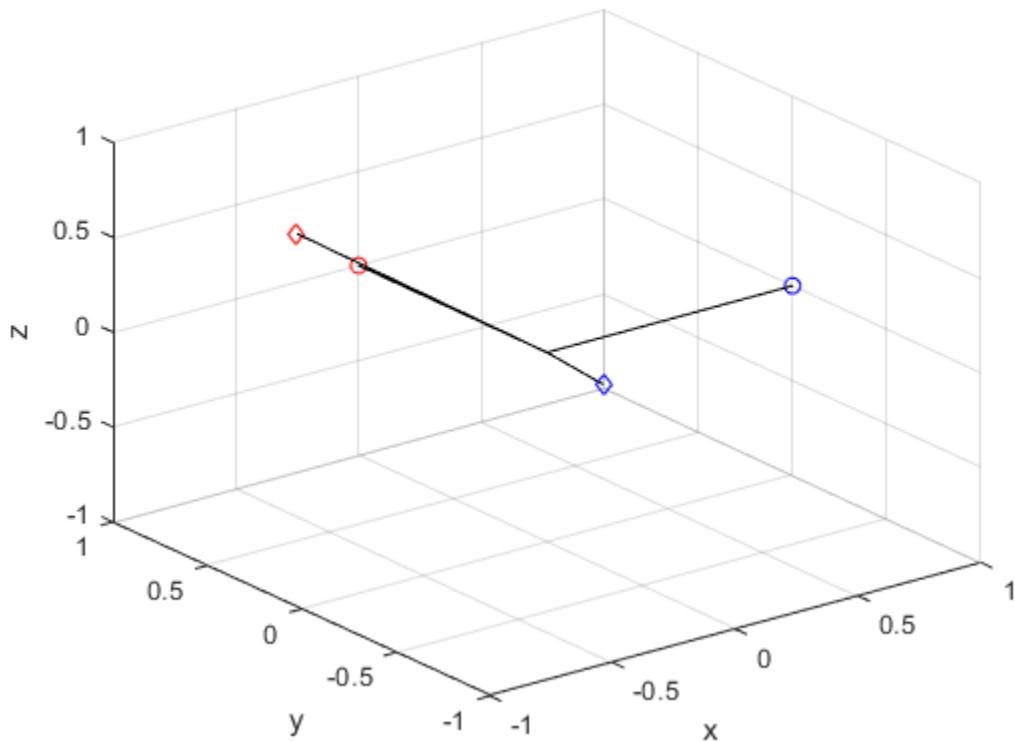
```

grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3), 'ro');
plot3(rP(1,1),rP(1,2),rP(1,3), 'bd')
plot3(rP(2,1),rP(2,2),rP(2,3), 'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)], 'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)], 'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)], 'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)], 'k')

```



## Input Arguments

### quat — Quaternion that defines rotation

scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion, row vector of quaternions, or column vector of quaternions.

Data Types: quaternion

**cartesianPoints — Three-dimensional Cartesian points**1-by-3 vector |  $N$ -by-3 matrixThree-dimensional Cartesian points, specified as a 1-by-3 vector or  $N$ -by-3 matrix.

Data Types: single | double

**Output Arguments****rotationResult — Repositioned Cartesian points**

vector | matrix

Rotated Cartesian points defined using the quaternion rotation, returned as a vector or matrix the same size as `cartesianPoints`.

Data Types: single | double

**Algorithms**Quaternion point rotation rotates a point specified in  $\mathbf{R}^3$  according to a specified quaternion:

$$L_q(u) = quq^*$$

where  $q$  is the quaternion,  $*$  represents conjugation, and  $u$  is the point to rotate, specified as a quaternion.For convenience, the `rotatepoint` function takes in a point in  $\mathbf{R}^3$  and returns a point in  $\mathbf{R}^3$ . Given a function call with some arbitrary quaternion,  $q = a + bi + cj + dk$ , and arbitrary coordinate,  $[x,y,z]$ , for example,

```
rereferencedPoint = rotatepoint(q,[x,y,z])
```

the `rotatepoint` function performs the following operations:

- 1 Converts point  $[x,y,z]$  to a quaternion:

$$u_q = 0 + xi + yj + zk$$

- 2 Normalizes the quaternion,  $q$ :

$$q_n = \frac{q}{\sqrt{a^2 + b^2 + c^2 + d^2}}$$

- 3 Applies the rotation:

$$v_q = qu_qq^*$$

- 4 Converts the quaternion output,  $v_q$ , back to  $\mathbf{R}^3$

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

rotateframe

### **Objects**

quaternion

**Introduced in R2021a**



# rotmat

Convert quaternion to rotation matrix

## Syntax

```
rotationMatrix = rotmat(quat,rotationType)
```

## Description

`rotationMatrix = rotmat(quat,rotationType)` converts the quaternion, `quat`, to an equivalent rotation matrix representation.

## Examples

### Convert Quaternion to Rotation Matrix for Point Rotation

Define a quaternion for use in point rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma], 'eulerd', 'ZYX', 'point')

quat = quaternion
      0.8924 + 0.23912i + 0.36964j + 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat, 'point')

rotationMatrix = 3×3

    0.7071    -0.0000    0.7071
    0.3536     0.8660   -0.3536
   -0.6124     0.5000     0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the *y*- and *x*-axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)  0          sind(theta) ; ...
      0            1          0           ; ...
      -sind(theta) 0          cosd(theta)];

rx = [1           0          0          ; ...
      0           cosd(gamma) -sind(gamma) ; ...
      0           sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry
```

```
rotationMatrixVerification = 3×3

    0.7071         0    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

### Convert Quaternion to Rotation Matrix for Frame Rotation

Define a quaternion for use in frame rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma], 'eulerd', 'ZYX', 'frame')

quat = quaternion
    0.8924 + 0.23912i + 0.36964j - 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat, 'frame')

rotationMatrix = 3×3

    0.7071   -0.0000   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the y- and x-axes. Multiply the rotation matrices and compare to the output of rotmat.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)  0          -sind(theta) ; ...
      0            1           0          ; ...
      sind(theta)  0          cosd(theta)];

rx = [1           0           0           ; ...
      0           cosd(gamma) sind(gamma) ; ...
      0           -sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry

rotationMatrixVerification = 3×3

    0.7071         0   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

## Convert Quaternion Vector to Rotation Matrices

Create a 3-by-1 normalized quaternion vector.

```
qVec = normalize( quaternion( randn(3,4) ) );
```

Convert the quaternion array to rotation matrices. The pages of `rotmatArray` correspond to the linear index of `qVec`.

```
rotmatArray = rotmat( qVec, 'frame' );
```

Assume `qVec` and `rotmatArray` correspond to a sequence of rotations. Combine the quaternion rotations into a single representation, then apply the quaternion rotation to arbitrarily initialized Cartesian points.

```
loc = normalize( randn(1,3) );
quat = prod( qVec );
rotateframe( quat, loc)
```

```
ans = 1×3
```

```
    0.9524    0.5297    0.9013
```

Combine the rotation matrices into a single representation, then apply the rotation matrix to the same initial Cartesian points. Verify the quaternion rotation and rotation matrix result in the same orientation.

```
totalRotMat = eye(3);
for i = 1:size( rotmatArray, 3 )
    totalRotMat = rotmatArray( :, :, i ) * totalRotMat;
end
totalRotMat * loc'
```

```
ans = 3×1
```

```
    0.9524
    0.5297
    0.9013
```

## Input Arguments

### **quat** — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

### **rotationType** — Type or rotation

'frame' | 'point'

Type of rotation represented by the `rotationMatrix` output, specified as 'frame' or 'point'.

Data Types: char | string

## Output Arguments

### rotationMatrix — Rotation matrix representation

3-by-3 matrix | 3-by-3-by-*N* multidimensional array

Rotation matrix representation, returned as a 3-by-3 matrix or 3-by-3-by-*N* multidimensional array.

- If `quat` is a scalar, `rotationMatrix` is returned as a 3-by-3 matrix.
- If `quat` is non-scalar, `rotationMatrix` is returned as a 3-by-3-by-*N* multidimensional array, where `rotationMatrix(:, :, i)` is the rotation matrix corresponding to `quat(i)`.

The data type of the rotation matrix is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

## Algorithms

Given a quaternion of the form

$$q = a + bi + cj + dk,$$

the equivalent rotation matrix for frame rotation is defined as

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc + 2ad & 2bd - 2ac \\ 2bc - 2ad & 2a^2 - 1 + 2c^2 & 2cd + 2ab \\ 2bd + 2ac & 2cd - 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

The equivalent rotation matrix for point rotation is the transpose of the frame rotation matrix:

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc - 2ad & 2bd + 2ac \\ 2bc + 2ad & 2a^2 - 1 + 2c^2 & 2cd - 2ab \\ 2bd - 2ac & 2cd + 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

## References

- [1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`rotvec` | `rotvecd` | `euler` | `eulerd`

### Objects

`quaternion`

**Introduced in R2021a**

## rotvec

Convert quaternion to rotation vector (radians)

### Syntax

```
rotationVector = rotvec(quat)
```

### Description

`rotationVector = rotvec(quat)` converts the quaternion array, `quat`, to an  $N$ -by-3 matrix of equivalent rotation vectors in radians. The elements of `quat` are normalized before conversion.

### Examples

#### Convert Quaternion to Rotation Vector in Radians

Convert a random quaternion scalar to a rotation vector in radians

```
quat = quaternion(randn(1,4));  
rotvec(quat)
```

```
ans = 1×3
```

```
    1.6866   -2.0774    0.7929
```

### Input Arguments

#### **quat** — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar quaternion, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

### Output Arguments

#### **rotationVector** — Rotation vector (radians)

$N$ -by-3 matrix

Rotation vector representation, returned as an  $N$ -by-3 matrix of rotations vectors, where each row represents the [X Y Z] angles of the rotation vectors in radians. The  $i$ th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

## Algorithms

All rotations in 3-D can be represented by a three-element axis of rotation and a rotation angle, for a total of four elements. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

$$q = \cos(\theta/2) + \sin(\theta/2)(xi + yj + zk),$$

where  $\theta$  is the angle of rotation and  $[x,y,z]$  represent the axis of rotation.

Given a quaternion of the form

$$q = a + bi + cj + dk,$$

you can solve for the rotation angle using the axis-angle form of quaternions:

$$\theta = 2\cos^{-1}(a).$$

Assuming a normalized axis, you can rewrite the quaternion as a rotation vector without loss of information by distributing  $\theta$  over the parts  $b$ ,  $c$ , and  $d$ . The rotation vector representation of  $q$  is

$$q_{rv} = \frac{\theta}{\sin(\theta/2)}[b, c, d].$$

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

rotvecd | euler | eulerd

### Objects

quaternion

**Introduced in R2021a**

## times, .\*

Element-wise quaternion multiplication

### Syntax

```
quatC = A.*B
```

### Description

`quatC = A.*B` returns the element-by-element quaternion multiplication of quaternion arrays.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the same order as the desired sequence of rotations. For example, to apply a  $p$  quaternion followed by a  $q$  quaternion, multiply in the order  $pq$ . The rotation operator becomes  $(pq)^*v(pq)$ , where  $v$  represents the object to rotate in quaternion form. `*` represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a  $p$  quaternion followed by a  $q$  quaternion, multiply in the reverse order,  $qp$ . The rotation operator becomes  $(qp)v(qp)^*$ .

### Examples

#### Multiply Two Quaternion Vectors

Create two vectors, A and B, and multiply them element by element.

```
A = quaternion([1:4;5:8]);
B = A;
C = A.*B
```

```
C = 2x1 quaternion array
    -28 + 4i + 6j + 8k
   -124 + 60i + 70j + 80k
```

#### Multiply Two Quaternion Arrays

Create two 3-by-3 arrays, A and B, and multiply them element by element.

```
A = reshape(quaternion(randn(9,4)),3,3);
B = reshape(quaternion(randn(9,4)),3,3);
C = A.*B
```

```
C = 3x3 quaternion array
    0.60169 + 2.4332i - 2.5844j + 0.51646k   -0.49513 + 1.1722i + 4.4401j - 1.217k
   -4.2329 + 2.4547i + 3.7768j + 0.77484k   -0.65232 - 0.43112i - 1.4645j - 0.90073k
```



```
-4.4159 + 2.1926i + 1.9037j - 4.0303k -2.0232 + 0.4205i - 0.17288j + 3.8529k
```

Note that quaternion multiplication is not commutative:

```
isequal(C,B.*A)
```

```
ans = logical
      0
```

## Multiply Quaternion Row and Column Vectors

Create a row vector **a** and a column vector **b**, then multiply them. The 1-by-3 row vector and 4-by-1 column vector combine to produce a 4-by-3 matrix with all combinations of elements multiplied.

```
a = [zeros('quaternion'),ones('quaternion'),quaternion(randn(1,4))]
```

```
a = 1x3 quaternion array
      0 + 0i + 0j + 0k      1 + 0i + 0j + 0k      0
```

```
b = quaternion(randn(4,4))
```

```
b = 4x1 quaternion array
      0.31877 + 3.5784i + 0.7254j - 0.12414k
      -1.3077 + 2.7694i - 0.063055j + 1.4897k
      -0.43359 - 1.3499i + 0.71474j + 1.409k
      0.34262 + 3.0349i - 0.20497j + 1.4172k
```

```
a.*b
```

```
ans = 4x3 quaternion array
      0 + 0i + 0j + 0k      0.31877 + 3.5784i + 0.7254j - 0.12414k
      0 + 0i + 0j + 0k      -1.3077 + 2.7694i - 0.063055j + 1.4897k
      0 + 0i + 0j + 0k      -0.43359 - 1.3499i + 0.71474j + 1.409k
      0 + 0i + 0j + 0k      0.34262 + 3.0349i - 0.20497j + 1.4172k
```

## Input Arguments

### A — Array to multiply

scalar | vector | matrix | multidimensional array

Array to multiply, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of them is 1.

Data Types: quaternion | single | double

### B – Array to multiply

scalar | vector | matrix | multidimensional array

Array to multiply, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of them is 1.

Data Types: quaternion | single | double

## Output Arguments

### quatC – Quaternion product

scalar | vector | matrix | multidimensional array

Quaternion product, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

## Algorithms

### Quaternion Multiplication by a Real Scalar

Given a quaternion,

$$q = a_q + b_q i + c_q j + d_q k,$$

the product of  $q$  and a real scalar  $\beta$  is

$$\beta q = \beta a_q + \beta b_q i + \beta c_q j + \beta d_q k$$

### Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$i^2 = j^2 = k^2 = ijk = -1,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	<b>1</b>	<b>i</b>	<b>j</b>	<b>k</b>
<b>1</b>	1	i	j	k
<b>i</b>	i	-1	k	-j
<b>j</b>	j	-k	-1	i
<b>k</b>	k	j	-i	-1

When reading the table, the rows are read first, for example:  $ij = k$  and  $ji = -k$ .

Given two quaternions,  $q = a_q + b_q i + c_q j + d_q k$ , and  $p = a_p + b_p i + c_p j + d_p k$ , the multiplication can be expanded as:

$$\begin{aligned}
z = pq &= (a_p + b_p i + c_p j + d_p k)(a_q + b_q i + c_q j + d_q k) \\
&= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\
&\quad + b_p a_q i + b_p b_q i^2 + b_p c_q ij + b_p d_q ik \\
&\quad + c_p a_q j + c_p b_q ji + c_p c_q j^2 + c_p d_q jk \\
&\quad + d_p a_q k + d_p b_q ki + d_p c_q kj + d_p d_q k^2
\end{aligned}$$

You can simplify the equation using the quaternion multiplication table.

$$\begin{aligned}
z = pq &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\
&\quad + b_p a_q i - b_p b_q + b_p c_q k - b_p d_q j \\
&\quad + c_p a_q j - c_p b_q k - c_p c_q + c_p d_q i \\
&\quad + d_p a_q k + d_p b_q j - d_p c_q i - d_p d_q
\end{aligned}$$

## References

- [1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

prod | mtimes, \*

### Objects

quaternion

**Introduced in R2021a**

## transpose, .'

Transpose a quaternion array

### Syntax

`Y = quat.'`

### Description

`Y = quat.'` returns the non-conjugate transpose of the quaternion array, `quat`.

### Examples

#### Vector Transpose

Create a vector of quaternions and compute its nonconjugate transpose.

```
quat = quaternion(randn(4,4))
```

```
quat = 4x1 quaternion array
    0.53767 + 0.31877i + 3.5784j + 0.7254k
    1.8339 - 1.3077i + 2.7694j - 0.063055k
   -2.2588 - 0.43359i - 1.3499j + 0.71474k
    0.86217 + 0.34262i + 3.0349j - 0.20497k
```

```
quatTransposed = quat.'
```

```
quatTransposed = 1x4 quaternion array
    0.53767 + 0.31877i + 3.5784j + 0.7254k    1.8339 - 1.3077i + 2.7694j - 0.063055k
```

#### Matrix Transpose

Create a matrix of quaternions and compute its nonconjugate transpose.

```
quat = [quaternion(randn(2,4)), quaternion(randn(2,4))]
```

```
quat = 2x2 quaternion array
    0.53767 - 2.2588i + 0.31877j - 0.43359k    3.5784 - 1.3499i + 0.7254j + 0.71474k
    1.8339 + 0.86217i - 1.3077j + 0.34262k    2.7694 + 3.0349i - 0.063055j - 0.20497k
```

```
quatTransposed = quat.'
```

```
quatTransposed = 2x2 quaternion array
    0.53767 - 2.2588i + 0.31877j - 0.43359k    1.8339 + 0.86217i - 1.3077j + 0.34262k
    3.5784 - 1.3499i + 0.7254j + 0.71474k    2.7694 + 3.0349i - 0.063055j - 0.20497k
```

## Input Arguments

### **quat** — Quaternion array to transpose

vector | matrix

Quaternion array to transpose, specified as a vector or matrix of quaternions. `transpose` is defined for 1-D and 2-D arrays. For higher-order arrays, use `permute`.

Data Types: quaternion

## Output Arguments

### **Y** — Transposed quaternion array

vector | matrix

Transposed quaternion array, returned as an  $N$ -by- $M$  array, where `quat` was specified as an  $M$ -by- $N$  array.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

`ctranspose`, '

### **Objects**

quaternion

**Introduced in R2021a**

## uminus, -

Quaternion unary minus

### Syntax

```
mQuat = -quat
```

### Description

mQuat = -quat negates the elements of quat and stores the result in mQuat.

### Examples

#### Negate Elements of Quaternion Matrix

Unary minus negates each part of a the quaternion. Create a 2-by-2 matrix, Q.

```
Q = quaternion(randn(2),randn(2),randn(2),randn(2))
```

Q = 2x2 quaternion array

0.53767 +	0.31877i +	3.5784j +	0.7254k	-2.2588 -	0.43359i -	1.3499j +	0.7147k
1.8339 -	1.3077i +	2.7694j -	0.063055k	0.86217 +	0.34262i +	3.0349j -	0.2049k

Negate the parts of each quaternion in Q.

```
R = -Q
```

R = 2x2 quaternion array

-0.53767 -	0.31877i -	3.5784j -	0.7254k	2.2588 +	0.43359i +	1.3499j -	0.7147k
-1.8339 +	1.3077i -	2.7694j +	0.063055k	-0.86217 -	0.34262i -	3.0349j +	0.2049k

### Input Arguments

#### quat — Quaternion array

scalar | vector | matrix | multidimensional array

Quaternion array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: quaternion

### Output Arguments

#### mQuat — Negated quaternion array

scalar | vector | matrix | multidimensional array

Negated quaternion array, returned as the same size as quat.

Data Types: quaternion

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

minus, -

### Objects

quaternion

**Introduced in R2021a**

## zeros

Create quaternion array with all parts set to zero

### Syntax

```
quatZeros = zeros('quaternion')
quatZeros = zeros(n,'quaternion')
quatZeros = zeros(sz,'quaternion')
quatZeros = zeros(sz1,...,szN,'quaternion')

quatZeros = zeros(___, 'like', prototype, 'quaternion')
```

### Description

`quatZeros = zeros('quaternion')` returns a scalar quaternion with all parts set to zero.

`quatZeros = zeros(n,'quaternion')` returns an n-by-n matrix of quaternions.

`quatZeros = zeros(sz,'quaternion')` returns an array of quaternions where the size vector, `sz`, defines `size(quatZeros)`.

`quatZeros = zeros(sz1,...,szN,'quaternion')` returns a `sz1`-by-...-by-`szN` array of quaternions where `sz1`, ..., `szN` indicates the size of each dimension.

`quatZeros = zeros(___, 'like', prototype, 'quaternion')` specifies the underlying class of the returned quaternion array to be the same as the underlying class of the quaternion prototype.

### Examples

#### Quaternion Scalar Zero

Create a quaternion scalar zero.

```
quatZeros = zeros('quaternion')

quatZeros = quaternion
           0 + 0i + 0j + 0k
```

#### Square Matrix of Quaternions

Create an n-by-n array of quaternion zeros.

```
n = 3;
quatZeros = zeros(n,'quaternion')

quatZeros = 3x3 quaternion array
           0 + 0i + 0j + 0k      0 + 0i + 0j + 0k      0 + 0i + 0j + 0k
```



```

0 + 0i + 0j + 0k    0 + 0i + 0j + 0k    0 + 0i + 0j + 0k
0 + 0i + 0j + 0k    0 + 0i + 0j + 0k    0 + 0i + 0j + 0k

```

### Multidimensional Array of Quaternion Zeros

Create a multidimensional array of quaternion zeros by defining array dimensions in order. In this example, you create a 3-by-1-by-2 array. You can specify dimensions using a row vector or comma-separated integers.

Specify the dimensions using a row vector and display the results:

```

dims = [3,1,2];
quatZerosSyntax1 = zeros(dims, 'quaternion')

```

```

quatZerosSyntax1 = 3x1x2 quaternion array
quatZerosSyntax1(:,:,1) =

```

```

0 + 0i + 0j + 0k
0 + 0i + 0j + 0k
0 + 0i + 0j + 0k

```

```

quatZerosSyntax1(:,:,2) =

```

```

0 + 0i + 0j + 0k
0 + 0i + 0j + 0k
0 + 0i + 0j + 0k

```

Specify the dimensions using comma-separated integers, and then verify the equivalence of the two syntaxes:

```

quatZerosSyntax2 = zeros(3,1,2, 'quaternion');
isequal(quatZerosSyntax1, quatZerosSyntax2)

```

```

ans = logical
     1

```

### Underlying Class of Quaternion Zeros

A quaternion is a four-part hyper-complex number used in three-dimensional representations. You can specify the underlying data type of the parts as `single` or `double`. The default is `double`.

Create a quaternion array of zeros with the underlying data type set to `single`.

```

quatZeros = zeros(2, 'like', single(1), 'quaternion')

```

```

quatZeros = 2x2 quaternion array
0 + 0i + 0j + 0k    0 + 0i + 0j + 0k
0 + 0i + 0j + 0k    0 + 0i + 0j + 0k

```

Verify the underlying class using the `classUnderlying` function.

```
classUnderlying(quatZeros)

ans =
'single'
```

## Input Arguments

### **n** — Size of square quaternion matrix

integer value

Size of square quaternion matrix, specified as an integer value. If `n` is 0 or negative, then `quatZeros` is returned as an empty matrix.

Example: `zeros(4, 'quaternion')` returns a 4-by-4 matrix of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **sz** — Output size

row vector of integer values

Output size, specified as a row vector of integer values. Each element of `sz` indicates the size of the corresponding dimension in `quatZeros`. If the size of any dimension is 0 or negative, then `quatZeros` is returned as an empty array.

Example: `zeros([1,4,2], 'quaternion')` returns a 1-by-4-by-2 array of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### **prototype** — Quaternion prototype

variable

Quaternion prototype, specified as a variable.

Example: `zeros(2, 'like', quat, 'quaternion')` returns a 2-by-2 matrix of quaternions with the same underlying class as the prototype quaternion, `quat`.

Data Types: `quaternion`

### **sz1, ..., szN** — Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integers.

- If the size of any dimension is 0, then `quatZeros` is returned as an empty array.
- If the size of any dimension is negative, then it is treated as 0.

Example: `zeros(2,3, 'quaternion')` returns a 2-by-3 matrix of quaternion zeros.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### **quatZeros** — Quaternion zeros

scalar | vector | matrix | multidimensional array

Quaternion zeros, returned as a quaternion or array of quaternions.

Given a quaternion of the form  $Q = a + bi + cj + dk$ , a quaternion zero is defined as  $Q = 0 + 0i + 0j + 0k$ .

Data Types: quaternion

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

ones

### Objects

quaternion

**Introduced in R2021a**

## getTrackPositions

Returns updated track positions and position covariance matrix

### Syntax

```
position = getTrackPositions(tracks,positionSelector)
[position,positionCovariances] = getTrackPositions(tracks,positionSelector)
```

### Description

`position = getTrackPositions(tracks,positionSelector)` returns a matrix of track positions. Each row contains the position of a tracked object.

`[position,positionCovariances] = getTrackPositions(tracks,positionSelector)` returns a matrix of track positions.

### Examples

#### Find Position of 3-D Constant-Acceleration Object

Create an extended Kalman filter tracker for 3-D constant-acceleration motion.

```
tracker = radarTracker('FilterInitializationFcn',@initcaekf);
```

Update the tracker with a single detection and get the tracks output.

```
detection = objectDetection(0,[10;-20;4],'ObjectClassID',3);
tracks = tracker(detection,0)
```

```
tracks =
  objectTrack with properties:
    TrackID: 1
    BranchID: 0
    SourceIndex: 0
    UpdateTime: 0
    Age: 1
    State: [9x1 double]
    StateCovariance: [9x9 double]
    StateParameters: [1x1 struct]
    ObjectClassID: 3
    TrackLogic: 'History'
    TrackLogicState: [1 0 0 0 0]
    IsConfirmed: 1
    IsCoasted: 0
    IsSelfReported: 1
    ObjectAttributes: [1x1 struct]
```

Obtain the position vector from the track state.

```
positionSelector = [1 0 0 0 0 0 0 0; 0 0 0 1 0 0 0 0; 0 0 0 0 0 0 1 0];
position = getTrackPositions(tracks, positionSelector)
```

```
position = 1×3
```

```
    10    -20     4
```

### Find Position and Covariance of 3-D Constant-Velocity Object

Create an extended Kalman filter tracker for 3-D constant-velocity motion.

```
tracker = radarTracker('FilterInitializationFcn',@initcvekf);
```

Update the tracker with a single detection and get the tracks output.

```
detection = objectDetection(0,[10;3;-7], 'ObjectClassID',3);
tracks = tracker(detection,0)
```

```
tracks =
```

```
  objectTrack with properties:
```

```
      TrackID: 1
      BranchID: 0
      SourceIndex: 0
      UpdateTime: 0
      Age: 1
      State: [6x1 double]
      StateCovariance: [6x6 double]
      StateParameters: [1x1 struct]
      ObjectClassID: 3
      TrackLogic: 'History'
      TrackLogicState: [1 0 0 0 0]
      IsConfirmed: 1
      IsCoasted: 0
      IsSelfReported: 1
      ObjectAttributes: [1x1 struct]
```

Obtain the position vector and position covariance for that track.

```
positionSelector = [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 1 0];
[position,positionCovariance] = getTrackPositions(tracks,positionSelector)
```

```
position = 1×3
```

```
    10     3    -7
```

```
positionCovariance = 3×3
```

```
     1     0     0
     0     1     0
     0     0     1
```

## Input Arguments

### **tracks — Object tracks**

array of `objectTrack` objects | array of structures

Object tracks, specified as an array of `objectTrack` objects or an array of structures containing sufficient information to obtain the track position information. At a minimum, these structures must contain a `State` column vector field and a positive-definite `StateCovariance` matrix field. For a sample track structure, see `toStruct`.

### **positionSelector — Position selection matrix**

$D$ -by- $N$  real-valued matrix.

Position selector, specified as a  $D$ -by- $N$  real-valued matrix of ones and zeros.  $D$  is the number of dimensions of the tracker.  $N$  is the size of the state vector. Using this matrix, the function extracts track positions from the state vector. Multiply the state vector by position selector matrix returns positions. The same selector is applied to all object tracks.

## Output Arguments

### **position — Positions of tracked objects**

real-valued  $M$ -by- $D$  matrix

Positions of tracked objects at last update time, returned as a real-valued  $M$ -by- $D$  matrix.  $D$  represents the number of position elements.  $M$  represents the number of tracks.

### **positionCovariances — Position covariance matrices of tracked objects**

real-valued  $D$ -by- $D$ - $M$  array

Position covariance matrices of tracked objects, returned as a real-valued  $D$ -by- $D$ - $M$  array.  $D$  represents the number of position elements.  $M$  represents the number of tracks. Each  $D$ -by- $D$  submatrix is a position covariance matrix for a track.

## More About

### **Position Selector for 2-Dimensional Motion**

Show the position selection matrix for two-dimensional motion when the state consists of the position and velocity.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

### **Position Selector for 3-Dimensional Motion**

Show the position selection matrix for three-dimensional motion when the state consists of the position and velocity.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

## Position Selector for 3-Dimensional Motion with Acceleration

Show the position selection matrix for three-dimensional motion when the state consists of the position, velocity, and acceleration.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

[getTrackVelocities](#) | [initcaekf](#) | [initcakf](#) | [initcaukf](#) | [initctekf](#) | [initctukf](#) | [initcvkf](#) | [initcvukf](#)

### Objects

[objectDetection](#) | [radarTracker](#)

## Introduced in R2021a

## getTrackVelocities

Obtain updated track velocities and velocity covariance matrix

### Syntax

```
velocity = getTrackVelocities(tracks,velocitySelector)  
[velocity,velocityCovariances] = getTrackVelocities(tracks,velocitySelector)
```

### Description

`velocity = getTrackVelocities(tracks,velocitySelector)` returns velocities of tracked objects.

`[velocity,velocityCovariances] = getTrackVelocities(tracks,velocitySelector)` also returns the track velocity covariance matrices.

### Examples

#### Find Velocity of 3-D Constant-Acceleration Object

Create an extended Kalman filter tracker for 3-D constant-acceleration motion.

```
tracker = radarTracker('FilterInitializationFcn',@initcaekf);
```

Initialize the tracker with one detection.

```
detection = objectDetection(0,[10;-20;4],'ObjectClassID',3);  
tracks = tracker(detection,0);
```

Add a second detection at a later time and at a different position.

```
detection = objectDetection(0.1,[10.3;-20.2;4],'ObjectClassID',3);  
tracks = tracker(detection,0.2);
```

Obtain the velocity vector from the track state.

```
velocitySelector = [0 1 0 0 0 0 0 0 0; 0 0 0 0 1 0 0 0 0; 0 0 0 0 0 0 0 1 0];  
velocity = getTrackVelocities(tracks,velocitySelector)
```

```
velocity = 1×3
```

```
    1.0093    -0.6728         0
```

#### Velocity and Covariance of 3-D Constant-Acceleration Object

Create an extended Kalman filter tracker for 3-D constant-acceleration motion.

```
tracker = radarTracker('FilterInitializationFcn',@initcaekf);
```



Initialize the tracker with one detection.

```
detection = objectDetection(0,[10;-20;4], 'ObjectClassID',3);
tracks = step(tracker,detection,0);
```

Add a second detection at a later time and at a different position.

```
detection = objectDetection(0.1,[10.3;-20.2;4.3], 'ObjectClassID',3);
tracks = step(tracker,detection,0.2);
```

Obtain the velocity vector from the track state.

```
velocitySelector = [0 1 0 0 0 0 0 0 0; 0 0 0 0 1 0 0 0 0; 0 0 0 0 0 0 0 0 1 0];
[velocity,velocityCovariance] = getTrackVelocities(tracks,velocitySelector)
```

velocity = 1×3

```
    1.0093    -0.6728     1.0093
```

velocityCovariance = 3×3

```
    70.0685         0         0
         0    70.0685         0
         0         0    70.0685
```

## Input Arguments

### tracks — Object tracks

array of `objectTrack` objects | array of structures

Object tracks, specified as an array of `objectTrack` objects or an array of structures containing sufficient information to obtain the track velocity information. At a minimum, these structures must contain a `State` column vector field and a positive-definite `StateCovariance` matrix field. For a sample track structure, see `toStruct`.

### velocitySelector — Velocity selection matrix

$D$ -by- $N$  real-valued matrix.

Velocity selector, specified as a  $D$ -by- $N$  real-valued matrix of ones and zeros.  $D$  is the number of dimensions of the tracker.  $N$  is the size of the state vector. Using this matrix, the function extracts track velocities from the state vector. Multiply the state vector by velocity selector matrix returns velocities. The same selector is applied to all object tracks.

## Output Arguments

### velocity — Velocities of tracked objects

real-valued  $1$ -by- $D$  vector | real-valued  $M$ -by- $D$  matrix

Velocities of tracked objects at last update time, returned as a  $1$ -by- $D$  vector or a real-valued  $M$ -by- $D$  matrix.  $D$  represents the number of velocity elements.  $M$  represents the number of tracks.

### velocityCovariances — Velocity covariance matrices of tracked objects

real-valued  $D$ -by- $D$ -matrix | real-valued  $D$ -by- $D$ -by- $M$  array

Velocity covariance matrices of tracked objects, returned as a real-valued  $D$ -by- $D$ -matrix or a real-valued  $D$ -by- $D$ -by- $M$  array.  $D$  represents the number of velocity elements.  $M$  represents the number of tracks. Each  $D$ -by- $D$  submatrix is a velocity covariance matrix for a track.

## More About

### Velocity Selector for 2-Dimensional Motion

Show the velocity selection matrix for two-dimensional motion when the state consists of the position and velocity.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### Velocity Selector for 3-Dimensional Motion

Show the velocity selection matrix for three-dimensional motion when the state consists of the position and velocity.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

### Velocity Selector for 3-Dimensional Motion with Acceleration

Show the velocity selection matrix for three-dimensional motion when the state consists of the position, velocity, and acceleration.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`getTrackPositions` | `initcaekf` | `initcakf` | `initcaukf` | `initctekf` | `initctukf` | `initcvkf` | `initcvukf`

### Objects

`objectDetection` | `radarTracker`

**Introduced in R2021a**

## cameas

Measurement function for constant-acceleration motion

### Syntax

```
measurement = cameas(state)
measurement = cameas(state, frame)
measurement = cameas(state, frame, sensorpos)
measurement = cameas(state, frame, sensorpos, sensorvel)
measurement = cameas(state, frame, sensorpos, sensorvel, laxes)
measurement = cameas(state, measurementParameters)
[measurement, bounds] = cameas( ___ )
```

### Description

`measurement = cameas(state)` returns the measurement, for the constant-acceleration Kalman filter motion model in rectangular coordinates. The `state` argument specifies the current state of the filter.

`measurement = cameas(state, frame)` also specifies the measurement coordinate system, `frame`.

`measurement = cameas(state, frame, sensorpos)` also specifies the sensor position, `sensorpos`.

`measurement = cameas(state, frame, sensorpos, sensorvel)` also specifies the sensor velocity, `sensorvel`.

`measurement = cameas(state, frame, sensorpos, sensorvel, laxes)` also specifies the local sensor axes orientation, `laxes`.

`measurement = cameas(state, measurementParameters)` specifies the measurement parameters, `measurementParameters`.

`[measurement, bounds] = cameas( ___ )` returns the measurement bounds, used by a tracking filter (`trackingEKF` or `trackingUKF`) in residual calculations. See the `HasMeasurementWrapping` of the filter object for more details.

### Examples

#### Create Measurement from Accelerating Object in Rectangular Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. The measurements are in rectangular coordinates.

```
state = [1,10,3,2,20,0.5].';
measurement = cameas(state)
```

```
measurement = 3×1
```

```
1  
2  
0
```

The measurement is returned in three-dimensions with the z-component set to zero.

### Create Measurement from Accelerating Object in Spherical Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. The measurements are in spherical coordinates.

```
state = [1,10,3,2,20,5].';  
measurement = cameas(state, 'spherical')
```

```
measurement = 4×1
```

```
63.4349  
0  
2.2361  
22.3607
```

The elevation of the measurement is zero and the range rate is positive. These results indicate that the object is moving away from the sensor.

### Create Measurement from Accelerating Object in Translated Spherical Frame

Define the state of an object moving in 2-D constant-acceleration motion. The state consists of position, velocity, and acceleration in each dimension. The measurements are in spherical coordinates with respect to a frame located at (20;40;0) meters from the origin.

```
state = [1,10,3,2,20,5].';  
measurement = cameas(state, 'spherical', [20;40;0])
```

```
measurement = 4×1
```

```
-116.5651  
0  
42.4853  
-22.3607
```

The elevation of the measurement is zero and the range rate is negative indicating that the object is moving toward the sensor.

## Create Measurement from Constant-Accelerating Object Using Measurement Parameters

Define the state of an object moving in 2-D constant-acceleration motion. The state consists of position, velocity, and acceleration in each dimension. The measurements are in spherical coordinates with respect to a frame located at (20;40;0) meters from the origin.

```
state2d = [1,10,3,2,20,5].';
```

The elevation of the measurement is zero and the range rate is negative indicating that the object is moving toward the sensor.

```
frame = 'spherical';
sensorpos = [20;40;0];
sensorvel = [0;5;0];
laxes = eye(3);
measurement = cameas(state2d,'spherical',sensorpos,sensorvel,laxes)
```

```
measurement = 4×1
```

```
-116.5651
         0
  42.4853
 -17.8885
```

The elevation of the measurement is zero and the range rate is negative. These results indicate that the object is moving toward the sensor.

Put the measurement parameters in a structure and use the alternative syntax.

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel, ...
    'Orientation',laxes);
measurement = cameas(state2d,measparm)
```

```
measurement = 4×1
```

```
-116.5651
         0
  42.4853
 -17.8885
```

## Display Residual Wrapping Bounds for cameas

Specify a 2-D state and specify a measurement structure such that the function outputs azimuth, range, and range-rate measurements.

```
state = [10 1 0.1 10 1 0.1]'; % [x vx ax y vy ay]'
mp = struct("Frame","Spherical", ...
    "HasAzimuth",true, ...
    "HasElevation",false, ...
    "HasRange",true, ...
    "HasVelocity",false);
```

Output the measurement and wrapping bounds using the cameas function.

```
[measure,bounds] = cameas(state,mp)
```

```
measure = 2×1
```

```
45.0000
14.1421
```

```
bounds = 2×2
```

```
-180 180
-Inf Inf
```

## Input Arguments

### state — Kalman filter state

real-valued  $3D$ -by $N$  matrix

Kalman filter state for constant-acceleration motion, specified as a real-valued  $3D$ -by $N$  matrix.  $D$  is the number of spatial degrees of freedom of motion and  $N$  is the number states. For each spatial degree of motion, the state vector, as a column of the `state` matrix, takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	[x;vx;ax]
2-D	[x;vx;ax;y;vy;ay]
3-D	[x;vx;ax;y;vy;ay;z;vz;az]

For example,  $x$  represents the  $x$ -coordinate,  $vx$  represents the velocity in the  $x$ -direction, and  $ax$  represents the acceleration in the  $x$ -direction. If the motion model is in one-dimensional space, the  $y$ - and  $z$ -axes are assumed to be zero. If the motion model is in two-dimensional space, values along the  $z$ -axis are assumed to be zero. Position coordinates are in meters. Velocity coordinates are in meters/second. Acceleration coordinates are in meters/second<sup>2</sup>.

Example: [5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]

Data Types: double

### frame — Measurement output frame

'rectangular' (default) | 'spherical'

Measurement output frame, specified as 'rectangular' or 'spherical'. When the frame is 'rectangular', a measurement consists of  $x$ ,  $y$ , and  $z$  Cartesian coordinates. When specified as 'spherical', a measurement consists of azimuth, elevation, range, and range rate.

Data Types: char

### sensorpos — Sensor position

[0;0;0] (default) | real-valued 3-by-1 column vector

Sensor position with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in meters.

Data Types: double

**sensorvel — Sensor velocity**

[0;0;0] (default) | real-valued 3-by-1 column vector

Sensor velocity with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in m/s.

Data Types: double

**laxes — Local sensor coordinate axes**

[1,0,0;0,1,0;0,0,1] (default) | 3-by-3 orthogonal matrix

Local sensor coordinate axes, specified as a 3-by-3 orthogonal matrix. Each column specifies the direction of the local x-, y-, and z-axes, respectively, with respect to the navigation frame. That is, the matrix is the rotation matrix from the global frame to the sensor frame.

Data Types: double

**measurementParameters — Measurement parameters**

structure | array of structure

Measurement parameters, specified as a structure or an array of structures. The fields of the structure are:

Field	Description	Example
Frame	Frame used to report measurements, specified as one of these values: <ul style="list-style-type: none"> <li>'rectangular' — Detections are reported in rectangular coordinates.</li> <li>'spherical' — Detections are reported in spherical coordinates.</li> </ul>	'spherical'
OriginPosition	Position offset of the origin of the frame relative to the parent frame, specified as an [x y z] real-valued vector.	[0 0 0]
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, specified as a [vx vy vz] real-valued vector.	[0 0 0]
Orientation	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	[1 0 0; 0 1 0; 0 0 1]
HasAzimuth	Logical scalar indicating if azimuth is included in the measurement.	1

Field	Description	Example
HasElevation	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if HasElevation is false, the reported measurements assume 0 degrees of elevation.	1
HasRange	Logical scalar indicating if range is included in the measurement.	1
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if Orientation performs a frame rotation from the parent coordinate frame to the child coordinate frame. When IsParentToChild is false, then Orientation performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

If you only want to perform one coordinate transformation, such as a transformation from the body frame to the sensor frame, you only need to specify a measurement parameter structure. If you want to perform multiple coordinate transformations, you need to specify an array of measurement parameter structures. To learn how to perform multiple transformations, see the “Convert Detections to objectDetection Format” (Sensor Fusion and Tracking Toolbox) example.

Data Types: struct

## Output Arguments

### measurement — Measurement vector

real-valued  $M$ -by- $N$  matrix

Measurement vector, returned as an  $M$ -by- $N$  matrix.  $M$  is the dimension of the measurement and  $N$ , the number of measurement, is the same as the number of states. The form of each measurement depends upon which syntax you use.



- When the syntax does not use the `measurementParameters` argument, the measurement vector is  $[x, y, z]$  when the `frame` input argument is set to 'rectangular' and  $[az;el;r;rr]$  when the frame is set to 'spherical'.
- When the syntax uses the `measurementParameters` argument, the size of the measurement vector depends on the values of the `frame`, `HasVelocity`, and `HasElevation` fields in the `measurementParameters` structure.

frame	measurement															
'spherical'	<p>Specifies the azimuth angle, <math>az</math>, elevation angle, <math>el</math>, range, <math>r</math>, and range rate, <math>rr</math>, of the object with respect to the local ego vehicle coordinate system. Positive values for range rate indicate that an object is moving away from the sensor.</p> <p><b>Spherical measurements</b></p> <table border="1"> <thead> <tr> <th></th> <th></th> <th colspan="2">HasElevation</th> </tr> <tr> <th></th> <th></th> <th>false</th> <th>true</th> </tr> </thead> <tbody> <tr> <td rowspan="2"><b>HasVelocity</b></td> <td>false</td> <td><math>[az;r]</math></td> <td><math>[az;el;r]</math></td> </tr> <tr> <td>true</td> <td><math>[az;r;rr]</math></td> <td><math>[az;el;r;rr]</math></td> </tr> </tbody> </table> <p>Angle units are in degrees, range units are in meters, and range rate units are in m/s.</p>			HasElevation				false	true	<b>HasVelocity</b>	false	$[az;r]$	$[az;el;r]$	true	$[az;r;rr]$	$[az;el;r;rr]$
		HasElevation														
		false	true													
<b>HasVelocity</b>	false	$[az;r]$	$[az;el;r]$													
	true	$[az;r;rr]$	$[az;el;r;rr]$													
'rectangular'	<p>Specifies the Cartesian position and velocity coordinates of the tracked object with respect to the ego vehicle coordinate system.</p> <p><b>Rectangular measurements</b></p> <table border="1"> <thead> <tr> <th rowspan="2"><b>HasVelocity</b></th> <th>false</th> <th><math>[x;y;y]</math></th> </tr> </thead> <tbody> <tr> <th>true</th> <th><math>[x;y;z;vx;vy;vz]</math></th> </tr> </tbody> </table> <p>Position units are in meters and velocity units are in m/s.</p>	<b>HasVelocity</b>	false	$[x;y;y]$	true	$[x;y;z;vx;vy;vz]$										
<b>HasVelocity</b>	false		$[x;y;y]$													
	true	$[x;y;z;vx;vy;vz]$														

Data Types: double

**bounds — Measurement residual wrapping bounds**

*M*-by-2 real-valued matrix

Measurement residual wrapping bounds, returned as an *M*-by-2 real-valued matrix, where *M* is the dimension of the measurement. Each row of the matrix corresponds to the lower and upper bounds for the specific dimension in the measurement output.

The function returns different bound values based on the `frame` input.

- If the `frame` input is specified as 'Rectangular', each row of the matrix is  $[-Inf \ Inf]$ , indicating the filter does not wrap the measurement residual in the filter.

- If the `frame` input is specified as `'Spherical'`, the returned `bounds` contains the bounds for specific measurement dimension based on the following:
  - When `HasAzimuth = true`, the matrix includes a row of `[-180 180]`, indicating the filter wraps the azimuth residual in the range of `[-180 180]` in degrees.
  - When `HasElevation = true`, the matrix includes a row of `[-90 90]`, indicating the filter wraps the elevation residual in the range of `[-90 90]` in degrees.
  - When `HasRange = true`, the matrix includes a row of `[-Inf Inf]`, indicating the filter does not wrap the range residual.
  - When `HasVelocity = true`, the matrix includes a row of `[-Inf Inf]`, indicating the filter does not wrap the range rate residual.

If you specify any of the options as `false`, the returned `bounds` does not contain the corresponding row. For example, if `HasAzimuth = true`, `HasElevation = false`, `HasRange = true`, `HasVelocity = true`, then `bounds` is returned as

```
-180    180
-Inf    Inf
-Inf    Inf
```

The filter wraps the measuring residuals based on this equation:

$$x_{wrap} = \text{mod}\left(x - \frac{a-b}{2}, b-a\right) + \frac{a-b}{2}$$

where  $x$  is the residual to wrap,  $a$  is the lower bound,  $b$  is the upper bound,  $\text{mod}$  is the modules after division function, and  $x_{wrap}$  is the wrapped residual.

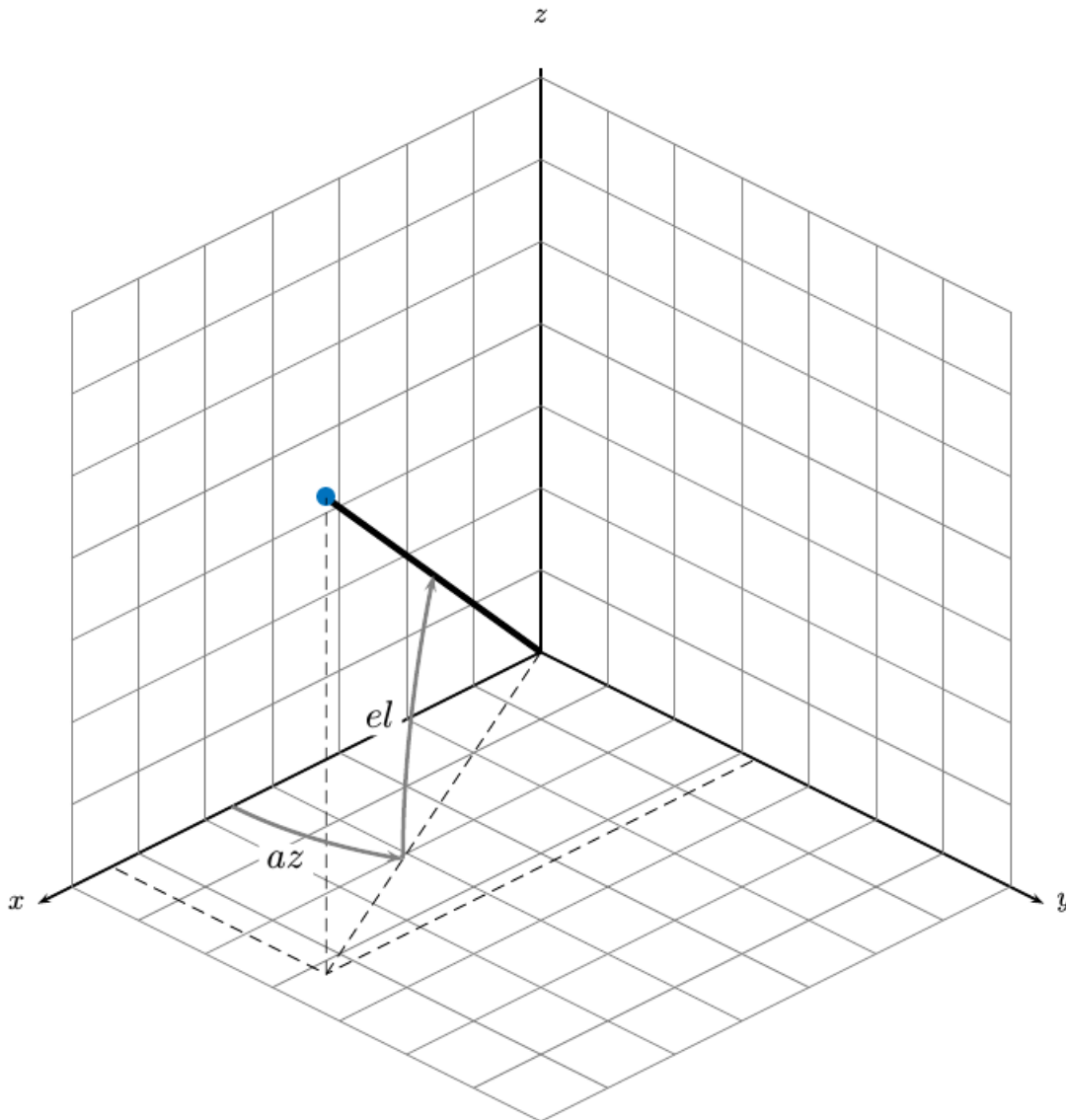
Data Types: `single` | `double`

## More About

### Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in the toolbox.

The azimuth angle of a vector is the angle between the  $x$ -axis and its orthogonal projection onto the  $xy$  plane. The angle is positive in going from the  $x$  axis toward the  $y$  axis. Azimuth angles lie between  $-180$  and  $180$  degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going toward the positive  $z$ -axis from the  $xy$  plane.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

constacc | constaccjac | cameasjac | constturn | constturnjac | ctmeas | ctmeasjac | constvel | constveljac | cvmeas | cvmeasjac

### Objects

trackingKF | trackingEKF | trackingUKF

**Introduced in R2021a**

# cameasjac

Jacobian of measurement function for constant-acceleration motion

## Syntax

```
measurementjac = cameasjac(state)
measurementjac = cameasjac(state, frame)
measurementjac = cameasjac(state, frame, sensorpos)
measurementjac = cameasjac(state, frame, sensorpos, sensorvel)
measurementjac = cameasjac(state, frame, sensorpos, sensorvel, laxes)
measurementjac = cameasjac(state, measurementParameters)
```

## Description

`measurementjac = cameasjac(state)` returns the measurement Jacobian, for constant-acceleration Kalman filter motion model in rectangular coordinates. The `state` argument specifies the current state of the filter.

`measurementjac = cameasjac(state, frame)` also specifies the measurement coordinate system, `frame`.

`measurementjac = cameasjac(state, frame, sensorpos)` also specifies the sensor position, `sensorpos`.

`measurementjac = cameasjac(state, frame, sensorpos, sensorvel)` also specifies the sensor velocity, `sensorvel`.

`measurementjac = cameasjac(state, frame, sensorpos, sensorvel, laxes)` also specifies the local sensor axes orientation, `laxes`.

`measurementjac = cameasjac(state, measurementParameters)` specifies the measurement parameters, `measurementParameters`.

## Examples

### Measurement Jacobian of Accelerating Object in Rectangular Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. Construct the measurement Jacobian in rectangular coordinates.

```
state = [1,10,3,2,20,5].';
jacobian = cameasjac(state)
```

```
jacobian = 3×6
```

```

     1     0     0     0     0     0
     0     0     0     1     0     0
     0     0     0     0     0     0
```

### Measurement Jacobian of Accelerating Object in Spherical Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. Compute the measurement Jacobian in spherical coordinates.

```
state = [1;10;3;2;20;5];
measurementjac = cameasjac(state, 'spherical')
```

```
measurementjac = 4×6
```

```
-22.9183      0      0      11.4592      0      0
      0      0      0      0      0      0
      0.4472      0      0      0.8944      0      0
      0.0000      0.4472      0      0.0000      0.8944      0
```

### Measurement Jacobian of Accelerating Object in Translated Spherical Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. Compute the measurement Jacobian in spherical coordinates with respect to an origin at (5;-20;0) meters.

```
state = [1,10,3,2,20,5].';
sensorpos = [5,-20,0].';
measurementjac = cameasjac(state, 'spherical', sensorpos)
```

```
measurementjac = 4×6
```

```
-2.5210      0      0     -0.4584      0      0
      0      0      0      0      0      0
-0.1789      0      0      0.9839      0      0
      0.5903     -0.1789      0      0.1073      0.9839      0
```

### Create Measurement Jacobian of Accelerating Object Using Measurement Parameters

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. Compute the measurement Jacobian in spherical coordinates with respect to an origin at (5;-20;0) meters.

```
state2d = [1,10,3,2,20,5].';
sensorpos = [5,-20,0].';
frame = 'spherical';
sensorvel = [0;8;0];
laxes = eye(3);
measurementjac = cameasjac(state2d, frame, sensorpos, sensorvel, laxes)
```

```
measurementjac = 4×6
```

```
-2.5210      0      0     -0.4584      0      0
```

```

      0      0      0      0      0      0
-0.1789    0      0      0.9839    0      0
 0.5274  -0.1789    0      0.0959    0.9839    0

```

Put the measurement parameters in a structure and use the alternative syntax.

```

measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel, ...
    'Orientation',axes);
measurementjac = cameasjac(state2d,measparm)

```

```

measurementjac = 4x6

```

```

-2.5210      0      0      -0.4584      0      0
      0      0      0      0      0      0
-0.1789      0      0      0.9839      0      0
 0.5274  -0.1789    0      0.0959    0.9839    0

```

## Input Arguments

### state — Kalman filter state vector

real-valued  $3N$ -element vector

Kalman filter state vector for constant-acceleration motion, specified as a real-valued  $3N$ -element vector.  $N$  is the number of spatial degrees of freedom of motion. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	[x;vx;ax]
2-D	[x;vx;ax;y;vy;ay]
3-D	[x;vx;ax;y;vy;ay;z;vz;az]

For example,  $x$  represents the  $x$ -coordinate,  $v_x$  represents the velocity in the  $x$ -direction, and  $a_x$  represents the acceleration in the  $x$ -direction. If the motion model is in one-dimensional space, the  $y$ - and  $z$ -axes are assumed to be zero. If the motion model is in two-dimensional space, values along the  $z$ -axis are assumed to be zero. Position coordinates are in meters. Velocity coordinates are in meters/second. Acceleration coordinates are in meters/second<sup>2</sup>.

Example: [5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]

Data Types: double

### frame — Measurement output frame

'rectangular' (default) | 'spherical'

Measurement output frame, specified as 'rectangular' or 'spherical'. When the frame is 'rectangular', a measurement consists of  $x$ ,  $y$ , and  $z$  Cartesian coordinates. When specified as 'spherical', a measurement consists of azimuth, elevation, range, and range rate.

Data Types: char

### sensorpos — Sensor position

[0;0;0] (default) | real-valued 3-by-1 column vector

Sensor position with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in meters.

Data Types: double

**sensorvel — Sensor velocity**

[0;0;0] (default) | real-valued 3-by-1 column vector

Sensor velocity with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in m/s.

Data Types: double

**laxes — Local sensor coordinate axes**

[1,0,0;0,1,0;0,0,1] (default) | 3-by-3 orthogonal matrix

Local sensor coordinate axes, specified as a 3-by-3 orthogonal matrix. Each column specifies the direction of the local x-, y-, and z-axes, respectively, with respect to the navigation frame. That is, the matrix is the rotation matrix from the global frame to the sensor frame.

Data Types: double

**measurementParameters — Measurement parameters**

structure | array of structure

Measurement parameters, specified as a structure or an array of structures. The fields of the structure are:

Field	Description	Example
Frame	Frame used to report measurements, specified as one of these values: <ul style="list-style-type: none"> <li>'rectangular' — Detections are reported in rectangular coordinates.</li> <li>'spherical' — Detections are reported in spherical coordinates.</li> </ul>	'spherical'
OriginPosition	Position offset of the origin of the frame relative to the parent frame, specified as an [x y z] real-valued vector.	[0 0 0]
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, specified as a [vx vy vz] real-valued vector.	[0 0 0]
Orientation	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	[1 0 0; 0 1 0; 0 0 1]



Field	Description	Example
HasAzimuth	Logical scalar indicating if azimuth is included in the measurement.	1
HasElevation	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if HasElevation is false, the reported measurements assume 0 degrees of elevation.	1
HasRange	Logical scalar indicating if range is included in the measurement.	1
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if Orientation performs a frame rotation from the parent coordinate frame to the child coordinate frame. When IsParentToChild is false, then Orientation performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

If you only want to perform one coordinate transformation, such as a transformation from the body frame to the sensor frame, you only need to specify a measurement parameter structure. If you want to perform multiple coordinate transformations, you need to specify an array of measurement parameter structures. To learn how to perform multiple transformations, see the “Convert Detections to objectDetection Format” (Sensor Fusion and Tracking Toolbox) example.

Data Types: struct

## Output Arguments

### measurementjac — Measurement Jacobian

real-valued 3-by- $N$  matrix | real-valued 4-by- $N$  matrix

Measurement Jacobian, specified as a real-valued 3-by- $N$  or 4-by- $N$  matrix.  $N$  is the dimension of the state vector. The interpretation of the rows and columns depends on the `frame` argument, as described in this table.

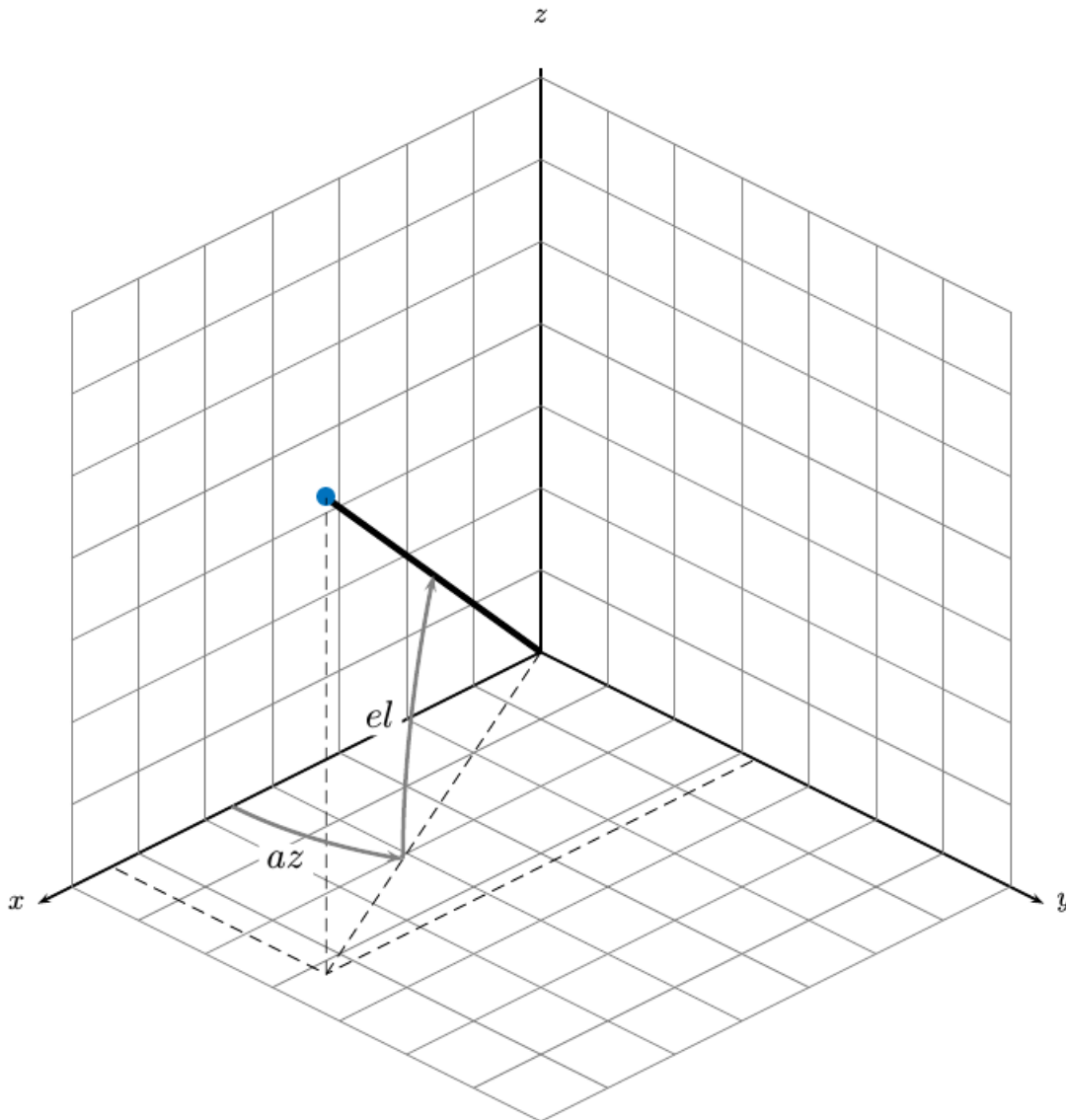
Frame	Measurement Jacobian
'rectangular'	Jacobian of the measurements $[x; y; z]$ with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
'spherical'	Jacobian of the measurement vector $[az; el; r; rr]$ with respect to the state vector. Measurement vector components specify the azimuth angle, elevation angle, range, and range rate of the object with respect to the local sensor coordinate system. Angle units are in degrees. Range units are in meters and range rate units are in meters/second.

## More About

### Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in the toolbox.

The azimuth angle of a vector is the angle between the  $x$ -axis and its orthogonal projection onto the  $xy$  plane. The angle is positive in going from the  $x$  axis toward the  $y$  axis. Azimuth angles lie between  $-180$  and  $180$  degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going toward the positive  $z$ -axis from the  $xy$  plane.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

constacc | constaccjac | cameas | constturn | constturnjac | ctmeas | ctmeasjac | constvel | constveljac | cvmeas | cvmeasjac

### Objects

trackingKF | trackingEKF | trackingUKF

**Introduced in R2021a**

## constacc

Constant-acceleration motion model

### Syntax

```
updatedstate = constacc(state)
updatedstate = constacc(state,dt)
updatedstate = constacc(state,w,dt)
```

### Description

`updatedstate = constacc(state)` returns the updated state, `state`, of a constant acceleration Kalman filter motion model for a step time of one second.

`updatedstate = constacc(state,dt)` specifies the time step, `dt`.

`updatedstate = constacc(state,w,dt)` also specifies the state noise, `w`.

### Examples

#### Predict State for Constant-Acceleration Motion

Define an initial state for 2-D constant-acceleration motion.

```
state = [1;1;1;2;1;0];
```

Predict the state 1 second later.

```
state = constacc(state)
```

```
state = 6×1
```

```
 2.5000
 2.0000
 1.0000
 3.0000
 1.0000
 0
```

#### Predict State for Constant-Acceleration Motion With Specified Time Step

Define an initial state for 2-D constant-acceleration motion.

```
state = [1;1;1;2;1;0];
```

Predict the state 0.5 s later.

```
state = constacc(state,0.5)
```

```
state = 6×1

    1.6250
    1.5000
    1.0000
    2.5000
    1.0000
    0
```

## Input Arguments

### state — Kalman filter state

real-valued  $3D$ -by- $N$  matrix

Kalman filter state for constant-acceleration motion, specified as a real-valued  $3D$ -by- $N$  matrix.  $D$  is the number of spatial degrees of freedom of motion and  $N$  is the number states. For each spatial degree of motion, the state vector, as a column of the `state` matrix, takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	[x;vx;ax]
2-D	[x;vx;ax;y;vy;ay]
3-D	[x;vx;ax;y;vy;ay;z;vz;az]

For example,  $x$  represents the  $x$ -coordinate,  $vx$  represents the velocity in the  $x$ -direction, and  $ax$  represents the acceleration in the  $x$ -direction. If the motion model is in one-dimensional space, the  $y$ - and  $z$ -axes are assumed to be zero. If the motion model is in two-dimensional space, values along the  $z$ -axis are assumed to be zero. Position coordinates are in meters. Velocity coordinates are in meters/second. Acceleration coordinates are in meters/second<sup>2</sup>.

Example: [5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]

Data Types: double

### dt — Time step interval of filter

1.0 (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: 0.5

Data Types: single | double

### w — State noise

scalar | real-valued  $D$ -by- $N$  matrix

State noise, specified as a scalar or real-valued  $D$ -by- $N$  matrix.  $D$  is the number of spatial degrees of freedom of motion and  $N$  is the number of state vectors. If specified as a scalar, the scalar value is expanded to a  $D$ -by- $N$  matrix.

Data Types: single | double

## Output Arguments

### updatedstate — Updated state vector

real-valued column or row vector | real-valued matrix

Updated state vector, returned as a real-valued vector or real-valued matrix with same number of elements and dimensions as the input state vector.

## Algorithms

For a two-dimensional constant-acceleration process, the state transition matrix after a time step,  $T$ , is block diagonal:

$$\begin{bmatrix} x_{k+1} \\ vx_{k+1} \\ ax_{k+1} \\ y_{k+1} \\ vy_{k+1} \\ ay_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 \\ 0 & 1 & T & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 \\ 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ vx_k \\ ax_k \\ y_k \\ vy_k \\ ay_k \end{bmatrix}$$

The block for each spatial dimension has this form:

$$\begin{bmatrix} 1 & T & \frac{1}{2}T^2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix}$$

For each additional spatial dimension, add an identical block.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

constaccjac | cameas | cameasjac | constturn | constturnjac | ctmeas | ctmeasjac | constvel | constveljac | cvmeas | cvmeasjac

### Objects

trackingKF | trackingEKF | trackingUKF

Introduced in R2021a

## constaccjac

Jacobian for constant-acceleration motion

### Syntax

```
jacobian = constaccjac(state)
jacobian = constaccjac(state,dt)
[jacobian,noisejacobian] = constaccjac(state,w,dt)
```

### Description

`jacobian = constaccjac(state)` returns the updated Jacobian , `jacobian`, for a constant-acceleration Kalman filter motion model. The step time is one second. The `state` argument specifies the current state of the filter.

`jacobian = constaccjac(state,dt)` also specifies the time step, `dt`.

`[jacobian,noisejacobian] = constaccjac(state,w,dt)` specifies the state noise, `w`, and returns the Jacobian, `noisejacobian`, of the state with respect to the noise.

### Examples

#### Compute State Jacobian for Constant-Acceleration Motion

Compute the state Jacobian for two-dimensional constant-acceleration motion.

Define an initial state and compute the state Jacobian for a one second update time.

```
state = [1,1,1,2,1,0];
jacobian = constaccjac(state)
```

```
jacobian = 6×6
```

```
    1.0000    1.0000    0.5000         0         0         0
         0    1.0000    1.0000         0         0         0
         0         0    1.0000         0         0         0
         0         0         0    1.0000    1.0000    0.5000
         0         0         0         0    1.0000    1.0000
         0         0         0         0         0    1.0000
```

#### Compute State Jacobian for Constant-Acceleration Motion with Specified Time Step

Compute the state Jacobian for two-dimensional constant-acceleration motion. Set the step time to 0.5 seconds.

```
state = [1,1,1,2,1,0].';
jacobian = constaccjac(state,0.5)
```



jacobian = 6×6

```

1.0000    0.5000    0.1250    0    0    0
0    1.0000    0.5000    0    0    0
0    0    1.0000    0    0    0
0    0    0    1.0000    0.5000    0.1250
0    0    0    0    1.0000    0.5000
0    0    0    0    0    1.0000

```

## Input Arguments

### state — Kalman filter state vector

real-valued  $3N$ -element vector

Kalman filter state vector for constant-acceleration motion, specified as a real-valued  $3N$ -element vector.  $N$  is the number of spatial degrees of freedom of motion. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	[x; vx; ax]
2-D	[x; vx; ax; y; vy; ay]
3-D	[x; vx; ax; y; vy; ay; z; vz; az]

For example,  $x$  represents the  $x$ -coordinate,  $vx$  represents the velocity in the  $x$ -direction, and  $ax$  represents the acceleration in the  $x$ -direction. If the motion model is in one-dimensional space, the  $y$ - and  $z$ -axes are assumed to be zero. If the motion model is in two-dimensional space, values along the  $z$ -axis are assumed to be zero. Position coordinates are in meters. Velocity coordinates are in meters/second. Acceleration coordinates are in meters/second<sup>2</sup>.

Example: [5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]

Data Types: double

### dt — Time step interval of filter

1.0 (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: 0.5

Data Types: single | double

### w — State noise

scalar | real-valued  $N$ -by-1 vector

State noise, specified as a scalar or real-valued real valued  $N$ -by-1 vector.  $N$  is the number of motion dimensions. For example,  $N = 2$  for the 2-D motion. If specified as a scalar, the scalar value is expanded to a  $N$ -by-1 vector.

Data Types: single | double

## Output Arguments

### **jacobian** — Constant-acceleration motion Jacobian

real-valued  $3N$ -by- $3N$  matrix

Constant-acceleration motion Jacobian, returned as a real-valued  $3N$ -by- $3N$  matrix.

### **noisejacobian** — Constant acceleration motion noise Jacobian

real-valued  $3N$ -by- $N$  matrix

Constant acceleration motion noise Jacobian, returned as a real-valued  $3N$ -by- $N$  matrix.  $N$  is the number of spatial degrees of motion. For example,  $N = 2$  for the 2-D motion. The Jacobian is constructed from the partial derivatives of the state at the updated time step with respect to the noise components.

## Algorithms

For a two-dimensional constant-acceleration process, the Jacobian matrix after a time step,  $T$ , is block diagonal:

$$\begin{bmatrix} 1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 \\ 0 & 1 & T & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 \\ 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The block for each spatial dimension has this form:

$$\begin{bmatrix} 1 & T & \frac{1}{2}T^2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix}$$

For each additional spatial dimension, add an identical block.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

constacc | cameas | cameasjac | constturn | constturnjac | ctmeas | ctmeasjac | constvel | constveljac | cvmeas | cvmeasjac

### **Objects**

trackingKF | trackingEKF | trackingUKF

**Introduced in R2021a**

## constturn

Constant turn-rate motion model

### Syntax

```
updatedstate = constturn(state)
updatedstate = constturn(state,dt)
updatedstate = constturn(state,w,dt)
```

### Description

`updatedstate = constturn(state)` returns the updated state, `updatedstate`, obtained from the previous state, `state`, after a one-second step time for motion modelled as constant turn rate. Constant turn rate means that motion in the x-y plane follows a constant angular velocity and motion in the vertical z directions follows a constant velocity model.

`updatedstate = constturn(state,dt)` also specifies the time step, `dt`.

`updatedstate = constturn(state,w,dt)` also specifies noise, `w`.

### Examples

#### Update State for Constant Turn-Rate Motion

Define an initial state for 2-D constant turn-rate motion. The turn rate is 12 degrees per second. Update the state to one second later.

```
state = [500,0,0,100,12].';
state = constturn(state)
```

```
state = 5×1
```

```
489.5662
-20.7912
99.2705
97.8148
12.0000
```

#### Update State for Constant Turn-Rate Motion with Specified Time Step

Define an initial state for 2-D constant turn-rate motion. The turn rate is 12 degrees per second. Update the state to 0.1 seconds later.

```
state = [500,0,0,100,12].';
state = constturn(state,0.1)
```

```
state = 5×1
```

```

499.8953
-2.0942
 9.9993
99.9781
12.0000

```

## Input Arguments

### state — State vector

real-valued 5-element vector | real-valued 7-element vector | 5-by- $N$  real-valued matrix | 7-by- $N$  real-valued matrix

State vector for a constant turn-rate motion model in two or three spatial dimensions, specified as a real-valued vector or matrix.

- When specified as a 5-element vector, the state vector describes 2-D motion in the  $x$ - $y$  plane. You can specify the state vector as a row or column vector. The components of the state vector are  $[x; vx; y; vy; \omega]$  where  $x$  represents the  $x$ -coordinate and  $vx$  represents the velocity in the  $x$ -direction.  $y$  represents the  $y$ -coordinate and  $vy$  represents the velocity in the  $y$ -direction.  $\omega$  represents the turn rate.

When specified as a 5-by- $N$  matrix, each column represents a different state vector.  $N$  represents the number of states.

- When specified as a 7-element vector, the state vector describes 3-D motion. You can specify the state vector as a row or column vector. The components of the state vector are  $[x; vx; y; vy; \omega; z; vz]$  where  $x$  represents the  $x$ -coordinate and  $vx$  represents the velocity in the  $x$ -direction.  $y$  represents the  $y$ -coordinate and  $vy$  represents the velocity in the  $y$ -direction.  $\omega$  represents the turn rate.  $z$  represents the  $z$ -coordinate and  $vz$  represents the velocity in the  $z$ -direction.

When specified as a 7-by- $N$  matrix, each column represents a different state vector.  $N$  represents the number of states.

Position coordinates are in meters. Velocity coordinates are in meters/second. Turn rate is in degrees/second.

Example: `[5;0.1;4;-0.2;0.01]`

Data Types: `double`

### dt — Time step interval of filter

1.0 (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: `0.5`

Data Types: `single` | `double`

### w — State noise

scalar | real-valued  $(D+1)$ -by- $N$  matrix

State noise, specified as a scalar or real-valued  $(D+1)$ -length -by- $N$  matrix.  $D$  is the number of motion dimensions and  $N$  is the number of state vectors. The components are each columns are

[ax;ay;alpha] for 2-D motion or [ax;ay;alpha;az] for 3-D motion. ax, ay, and az are the linear acceleration noise values in the x-, y-, and z-axes, respectively, and alpha is the angular acceleration noise value. If specified as a scalar, the value expands to a  $(D+1)$ -by- $N$  matrix.

Data Types: `single` | `double`

## Output Arguments

### **updatedstate** — Updated state vector

real-valued column or row vector | real-valued matrix

Updated state vector, returned as a real-valued vector or real-valued matrix with same number of elements and dimensions as the input state vector.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

`constacc` | `constaccjac` | `cameas` | `cameasjac` | `constturnjac` | `ctmeas` | `ctmeasjac` | `constvel` | `constveljac` | `cvmeas` | `cvmeasjac` | `initctekf` | `initctukf`

### **Objects**

`trackingEKF` | `trackingUKF`

**Introduced in R2021a**

# constturnjac

Jacobian for constant turn-rate motion

## Syntax

```
jacobian = constturnjac(state)
jacobian = constturnjac(state,dt)
[jacobian,noisejacobian] = constturnjac(state,w,dt)
```

## Description

`jacobian = constturnjac(state)` returns the updated Jacobian, `jacobian`, for constant turn-rate Kalman filter motion model for a one-second step time. The `state` argument specifies the current state of the filter. Constant turn rate means that motion in the x-y plane follows a constant angular velocity and motion in the vertical z directions follows a constant velocity model.

`jacobian = constturnjac(state,dt)` specifies the time step, `dt`.

`[jacobian,noisejacobian] = constturnjac(state,w,dt)` also specifies noise, `w`, and returns the Jacobian, `noisejacobian`, of the state with respect to the noise.

## Examples

### Compute State Jacobian for Constant Turn-Rate Motion

Compute the Jacobian for a constant turn-rate motion state. Assume the turn rate is 12 degrees/second. The time step is one second.

```
state = [500,0,0,100,12];
jacobian = constturnjac(state)
```

`jacobian = 5×5`

```

1.0000    0.9927         0   -0.1043   -0.8631
         0    0.9781         0   -0.2079   -1.7072
         0    0.1043    1.0000    0.9927   -0.1213
         0    0.2079         0    0.9781   -0.3629
         0         0         0         0    1.0000
```

### Compute State Jacobian for Constant Turn-Rate Motion with Specified Time Step

Compute the Jacobian for a constant turn-rate motion state. Assume the turn rate is 12 degrees/second. The time step is 0.1 second.

```
state = [500,0,0,100,12];
jacobian = constturnjac(state,0.1)
```

```
jacobian = 5x5
    1.0000    0.1000         0   -0.0010   -0.0087
         0    0.9998         0   -0.0209   -0.1745
         0    0.0010    1.0000    0.1000   -0.0001
         0    0.0209         0    0.9998   -0.0037
         0         0         0         0    1.0000
```

## Input Arguments

### state — State vector

real-valued 5-element vector | real-valued 7-element vector

State vector for a constant turn-rate motion model in two or three spatial dimensions, specified as a real-valued vector.

- When specified as a 5-element vector, the state vector describes 2-D motion in the x-y plane. You can specify the state vector as a row or column vector. The components of the state vector are `[x; vx; y; vy; omega]` where `x` represents the x-coordinate and `vx` represents the velocity in the x-direction. `y` represents the y-coordinate and `vy` represents the velocity in the y-direction. `omega` represents the turn rate.
- When specified as a 7-element vector, the state vector describes 3-D motion. You can specify the state vector as a row or column vector. The components of the state vector are `[x; vx; y; vy; omega; z; vz]` where `x` represents the x-coordinate and `vx` represents the velocity in the x-direction. `y` represents the y-coordinate and `vy` represents the velocity in the y-direction. `omega` represents the turn rate. `z` represents the z-coordinate and `vz` represents the velocity in the z-direction.

Position coordinates are in meters. Velocity coordinates are in meters/second. Turn rate is in degrees/second.

Example: `[5;0.1;4;-0.2;0.01]`

Data Types: `double`

### dt — Time step interval of filter

1.0 (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: `0.5`

Data Types: `single` | `double`

### w — State noise

scalar | real-valued  $(D+1)$  vector

State noise, specified as a scalar or real-valued  $M$ -by- $(D+1)$ -length vector.  $D$  is the number of motion dimensions.  $D$  is two for 2-D motion and  $D$  is three for 3-D motion. The vector components are `[ax; ay; alpha]` for 2-D motion or `[ax; ay; alpha; az]` for 3-D motion. `ax`, `ay`, and `az` are the linear acceleration noise values in the x-, y-, and z-axes, respectively, and `alpha` is the angular acceleration noise value. If specified as a scalar, the value expands to a  $(D+1)$  vector.

Data Types: `single` | `double`



## Output Arguments

### **jacobian** — Constant turn-rate motion Jacobian

real-valued 5-by-5 matrix | real-valued 7-by-7 matrix

Constant turn-rate motion Jacobian, returned as a real-valued 5-by-5 matrix or 7-by-7 matrix depending on the size of the `state` vector. The Jacobian is constructed from the partial derivatives of the state at the updated time step with respect to the state at the previous time step.

### **noisejacobian** — Constant turn-rate motion noise Jacobian

real-valued 5-by-5 matrix | real-valued 7-by-7 matrix

Constant turn-rate motion noise Jacobian, returned as a real-valued 5-by- $(D+1)$  matrix where  $D$  is two for 2-D motion or a real-valued 7-by- $(D+1)$  matrix where  $D$  is three for 3-D motion. The Jacobian is constructed from the partial derivatives of the state at the updated time step with respect to the noise components.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

`constacc` | `constaccjac` | `cameas` | `cameasjac` | `constturn` | `ctmeas` | `ctmeasjac` | `constvel` | `constveljac` | `cvmeas` | `cvmeasjac` | `initctekf`

### **Objects**

`trackingEKF`

**Introduced in R2021a**

## constvel

Constant velocity state update

### Syntax

```
updatedstate = constvel(state)
updatedstate = constvel(state,dt)
updatedstate = constvel(state,w,dt)
```

### Description

`updatedstate = constvel(state)` returns the updated state, `state`, of a constant-velocity Kalman filter motion model after a one-second time step.

`updatedstate = constvel(state,dt)` specifies the time step, `dt`.

`updatedstate = constvel(state,w,dt)` also specifies state noise, `w`.

### Examples

#### Update State for Constant-Velocity Motion

Update the state of two-dimensional constant-velocity motion for a time interval of one second.

```
state = [1;1;2;1];
state = constvel(state)
```

```
state = 4×1
```

```
    2
    1
    3
    1
```

#### Update State for Constant-Velocity Motion with Specified Time Step

Update the state of two-dimensional constant-velocity motion for a time interval of 1.5 seconds.

```
state = [1;1;2;1];
state = constvel(state,1.5)
```

```
state = 4×1
```

```
    2.5000
    1.0000
    3.5000
    1.0000
```

## Input Arguments

### state — Kalman filter state

real-valued  $2D$ -by- $N$  matrix

Kalman filter state for constant-velocity motion, specified as a real-valued  $2D$ -by- $N$  matrix.  $D$  is the number of spatial degrees of freedom of motion and  $N$  is the number states. The `state` is expected to be Cartesian state. For each spatial degree of motion, the state vector, as a column of the `state` matrix, takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	[x; vx]
2-D	[x; vx; y; vy]
3-D	[x; vx; y; vy; z; vz]

For example,  $x$  represents the  $x$ -coordinate and  $v_x$  represents the velocity in the  $x$ -direction. If the motion model is 1-D, values along the  $y$  and  $z$  axes are assumed to be zero. If the motion model is 2-D, values along the  $z$  axis are assumed to be zero. Position coordinates are in meters and velocity coordinates are in meters/sec.

Example: [5; .1; 0; -.2; -3; .05]

Data Types: `single` | `double`

### dt — Time step interval of filter

1.0 (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: 0.5

Data Types: `single` | `double`

### w — State noise

scalar | real-valued  $D$ -by- $N$  matrix

State noise, specified as a scalar or real-valued  $D$ -by- $N$  matrix.  $D$  is the number of spatial degrees of freedom of motion and  $N$  is the number of state vectors. For example,  $D = 2$  for the 2-D motion. If specified as a scalar, the scalar value is expanded to a  $D$ -by- $N$  matrix.

Data Types: `single` | `double`

## Output Arguments

### updatedstate — Updated state vector

real-valued column or row vector | real-valued matrix

Updated state vector, returned as a real-valued vector or real-valued matrix with same number of elements and dimensions as the input state vector.

## Algorithms

For a two-dimensional constant-velocity process, the state transition matrix after a time step,  $T$ , is block diagonal as shown here.

$$\begin{bmatrix} x_{k+1} \\ v_{x,k+1} \\ y_{k+1} \\ v_{y,k+1} \end{bmatrix} = \begin{bmatrix} 1 & T & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ vx_k \\ y_k \\ vy_k \end{bmatrix}$$

The block for each spatial dimension is:

$$\begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}$$

For each additional spatial dimension, add an identical block.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

constacc | constaccjac | cameas | cameasjac | constturn | constturnjac | ctmeas | ctmeasjac | constveljac | cvmeas | cvmeasjac

### Objects

trackingKF | trackingEKF | trackingUKF

### Introduced in R2021a

# constveljac

Jacobian for constant-velocity motion

## Syntax

```
jacobian = constveljac(state)
jacobian = constveljac(state,dt)
[jacobian,noisejacobian] = constveljac(state,w,dt)
```

## Description

`jacobian = constveljac(state)` returns the updated Jacobian , `jacobian`, for a constant-velocity Kalman filter motion model for a step time of one second. The `state` argument specifies the current state of the filter.

`jacobian = constveljac(state,dt)` specifies the time step, `dt`.

`[jacobian,noisejacobian] = constveljac(state,w,dt)` specifies the state noise, `w`, and returns the Jacobian, `noisejacobian`, of the state with respect to the noise.

## Examples

### Compute State Jacobian for Constant-Velocity Motion

Compute the state Jacobian for a two-dimensional constant-velocity motion model for a one second update time.

```
state = [1,1,2,1].';
jacobian = constveljac(state)
```

```
jacobian = 4×4
```

```
    1    1    0    0
    0    1    0    0
    0    0    1    1
    0    0    0    1
```

### Compute State Jacobian for Constant-Velocity Motion with Specified Time Step

Compute the state Jacobian for a two-dimensional constant-velocity motion model for a half-second update time.

```
state = [1;1;2;1];
```

Compute the state update Jacobian for 0.5 second.

```
jacobian = constveljac(state,0.5)
```

```
jacobian = 4x4
    1.0000    0.5000         0         0
         0    1.0000         0         0
         0         0    1.0000    0.5000
         0         0         0    1.0000
```

## Input Arguments

### state — Kalman filter state vector

real-valued  $2N$ -element vector

Kalman filter state vector for constant-velocity motion, specified as a real-valued  $2N$ -element column vector where  $N$  is the number of spatial degrees of freedom of motion. The `state` is expected to be Cartesian state. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	[x; vx]
2-D	[x; vx; y; vy]
3-D	[x; vx; y; vy; z; vz]

For example, `x` represents the  $x$ -coordinate and `vx` represents the velocity in the  $x$ -direction. If the motion model is 1-D, values along the  $y$  and  $z$  axes are assumed to be zero. If the motion model is 2-D, values along the  $z$  axis are assumed to be zero. Position coordinates are in meters and velocity coordinates are in meters/sec.

Example: [5; .1; 0; -.2; -3; .05]

Data Types: single | double

### dt — Time step interval of filter

1.0 (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: 0.5

Data Types: single | double

### w — State noise

scalar | real-valued  $N$ -by-1 vector

State noise, specified as a scalar or real-valued real valued  $N$ -by-1 vector.  $N$  is the number of motion dimensions. For example,  $N = 2$  for the 2-D motion. If specified as a scalar, the scalar value is expanded to an  $N$ -by-1 vector.

Data Types: single | double

## Output Arguments

### jacobian — Constant-velocity motion Jacobian

real-valued  $2N$ -by- $2N$  matrix

Constant-velocity motion Jacobian, returned as a real-valued  $2N$ -by- $2N$  matrix.  $N$  is the number of spatial degrees of motion.

### **noisejacobian — Constant velocity motion noise Jacobian**

real-valued  $2N$ -by- $N$  matrix

Constant velocity motion noise Jacobian, returned as a real-valued  $2N$ -by- $N$  matrix.  $N$  is the number of spatial degrees of motion. The Jacobian is constructed from the partial derivatives of the state at the updated time step with respect to the noise components.

## **Algorithms**

For a two-dimensional constant-velocity motion, the Jacobian matrix for a time step,  $T$ , is block diagonal:

$$\begin{bmatrix} 1 & T & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The block for each spatial dimension has this form:

$$\begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}$$

For each additional spatial dimension, add an identical block.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Functions**

constacc | constaccjac | cameas | cameasjac | constturn | constturnjac | ctmeas | ctmeasjac | constvel | cvmeas | cvmeasjac

### **Objects**

trackingKF | trackingEKF | trackingUKF

**Introduced in R2021a**

## ctmeas

Measurement function for constant turn-rate motion

### Syntax

```
measurement = ctmeas(state)
measurement = ctmeas(state, frame)
measurement = ctmeas(state, frame, sensorpos)
measurement = ctmeas(state, frame, sensorpos, sensorvel)
measurement = ctmeas(state, frame, sensorpos, sensorvel, laxes)
measurement = ctmeas(state, measurementParameters)
[measurement, bounds] = ctmeas( ___ )
```

### Description

`measurement = ctmeas(state)` returns the measurement for a constant turn-rate Kalman filter motion model in rectangular coordinates. The `state` argument specifies the current state of the filter.

`measurement = ctmeas(state, frame)` also specifies the measurement coordinate system, `frame`.

`measurement = ctmeas(state, frame, sensorpos)` also specifies the sensor position, `sensorpos`.

`measurement = ctmeas(state, frame, sensorpos, sensorvel)` also specifies the sensor velocity, `sensorvel`.

`measurement = ctmeas(state, frame, sensorpos, sensorvel, laxes)` also specifies the local sensor axes orientation, `laxes`.

`measurement = ctmeas(state, measurementParameters)` specifies the measurement parameters, `measurementParameters`.

`[measurement, bounds] = ctmeas( ___ )` returns the measurement bounds, used by a tracking filter (`trackingEKF` or `trackingUKF`) in residual calculations. See the `HasMeasurementWrapping` of the filter object for more details.

### Examples

#### Create Measurement from Constant Turn-Rate Motion in Rectangular Frame

Create a measurement from an object undergoing constant turn-rate motion. The state is the position and velocity in each dimension and the turn-rate. The measurements are in rectangular coordinates.

```
state = [1;10;2;20;5];
measurement = ctmeas(state)
```

```
measurement = 3×1
```



```
1
2
0
```

The z-component of the measurement is zero.

### Create Measurement from Constant Turn-Rate Motion in Spherical Frame

Define the state of an object in 2-D constant turn-rate motion. The state is the position and velocity in each dimension, and the turn rate. The measurements are in spherical coordinates.

```
state = [1;10;2;20;5];
measurement = ctmeas(state, 'spherical')
```

```
measurement = 4×1
```

```
63.4349
      0
  2.2361
 22.3607
```

The elevation of the measurement is zero and the range rate is positive indicating that the object is moving away from the sensor.

### Create Measurement from Constant Turn-Rate Motion in Translated Spherical Frame

Define the state of an object moving in 2-D constant turn-rate motion. The state consists of position and velocity, and the turn rate. The measurements are in spherical coordinates with respect to a frame located at [20;40;0].

```
state = [1;10;2;20;5];
measurement = ctmeas(state, 'spherical', [20;40;0])
```

```
measurement = 4×1
```

```
-116.5651
      0
  42.4853
 -22.3607
```

The elevation of the measurement is zero and the range rate is negative indicating that the object is moving toward the sensor.

### Create Measurement from Constant Turn-Rate Motion using Measurement Parameters

Define the state of an object moving in 2-D constant turn-rate motion. The state consists of position and velocity, and the turn rate. The measurements are in spherical coordinates with respect to a frame located at [20;40;0].

```
state2d = [1;10;2;20;5];
frame = 'spherical';
sensorpos = [20;40;0];
sensorvel = [0;5;0];
laxes = eye(3);
measurement = ctmeas(state2d,frame,sensorpos,sensorvel,laxes)
```

```
measurement = 4×1
```

```
-116.5651
      0
  42.4853
 -17.8885
```

The elevation of the measurement is zero and the range rate is negative indicating that the object is moving toward the sensor.

Put the measurement parameters in a structure and use the alternative syntax.

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes);
measurement = ctmeas(state2d,measparm)
```

```
measurement = 4×1
```

```
-116.5651
      0
  42.4853
 -17.8885
```

### Display Residual Wrapping Bounds for ctmeas

Specify a 2-D state and specify a measurement structure such that the function outputs azimuth, range, and range-rate measurements.

```
state = [10 1 10 1 0.5]'; % [x vx y vy omega]'
mp = struct("Frame","Spherical", ...
    "HasAzimuth",true, ...
    "HasElevation",false, ...
    "HasRange",true, ...
    "HasVelocity",false);
```

Output the measurement and wrapping bounds using the ctmeas function.

```
[measure,bounds] = ctmeas(state,mp)
```

```
measure = 2×1
```

```
45.0000
14.1421
```

```
bounds = 2x2
```

```
-180    180
-Inf    Inf
```

## Input Arguments

### state — State vector

real-valued 5-element vector | real-valued 7-element vector | 5-by- $N$  real-valued matrix | 7-by- $N$  real-valued matrix

State vector for a constant turn-rate motion model in two or three spatial dimensions, specified as a real-valued vector or matrix.

- When specified as a 5-element vector, the state vector describes 2-D motion in the  $x$ - $y$  plane. You can specify the state vector as a row or column vector. The components of the state vector are  $[x; vx; y; vy; \omega]$  where  $x$  represents the  $x$ -coordinate and  $vx$  represents the velocity in the  $x$ -direction.  $y$  represents the  $y$ -coordinate and  $vy$  represents the velocity in the  $y$ -direction.  $\omega$  represents the turn rate.

When specified as a 5-by- $N$  matrix, each column represents a different state vector.  $N$  represents the number of states.

- When specified as a 7-element vector, the state vector describes 3-D motion. You can specify the state vector as a row or column vector. The components of the state vector are  $[x; vx; y; vy; \omega; z; vz]$  where  $x$  represents the  $x$ -coordinate and  $vx$  represents the velocity in the  $x$ -direction.  $y$  represents the  $y$ -coordinate and  $vy$  represents the velocity in the  $y$ -direction.  $\omega$  represents the turn rate.  $z$  represents the  $z$ -coordinate and  $vz$  represents the velocity in the  $z$ -direction.

When specified as a 7-by- $N$  matrix, each column represents a different state vector.  $N$  represents the number of states.

Position coordinates are in meters. Velocity coordinates are in meters/second. Turn rate is in degrees/second.

Example: `[5;0.1;4;-0.2;0.01]`

Data Types: double

### frame — Measurement output frame

'rectangular' (default) | 'spherical'

Measurement output frame, specified as 'rectangular' or 'spherical'. When the frame is 'rectangular', a measurement consists of  $x$ ,  $y$ , and  $z$  Cartesian coordinates. When specified as 'spherical', a measurement consists of azimuth, elevation, range, and range rate.

Data Types: char

### sensorpos — Sensor position

`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in meters.

Data Types: double

**sensorvel — Sensor velocity**

[0;0;0] (default) | real-valued 3-by-1 column vector

Sensor velocity with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in m/s.

Data Types: double

**laxes — Local sensor coordinate axes**

[1,0,0;0,1,0;0,0,1] (default) | 3-by-3 orthogonal matrix

Local sensor coordinate axes, specified as a 3-by-3 orthogonal matrix. Each column specifies the direction of the local x-, y-, and z-axes, respectively, with respect to the navigation frame. That is, the matrix is the rotation matrix from the global frame to the sensor frame.

Data Types: double

**measurementParameters — Measurement parameters**

structure | array of structure

Measurement parameters, specified as a structure or an array of structures. The fields of the structure are:

Field	Description	Example
Frame	Frame used to report measurements, specified as one of these values: <ul style="list-style-type: none"> <li>'rectangular' — Detections are reported in rectangular coordinates.</li> <li>'spherical' — Detections are reported in spherical coordinates.</li> </ul>	'spherical'
OriginPosition	Position offset of the origin of the frame relative to the parent frame, specified as an [x y z] real-valued vector.	[0 0 0]
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, specified as a [vx vy vz] real-valued vector.	[0 0 0]
Orientation	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	[1 0 0; 0 1 0; 0 0 1]

Field	Description	Example
HasAzimuth	Logical scalar indicating if azimuth is included in the measurement.	1
HasElevation	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if HasElevation is false, the reported measurements assume 0 degrees of elevation.	1
HasRange	Logical scalar indicating if range is included in the measurement.	1
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if Orientation performs a frame rotation from the parent coordinate frame to the child coordinate frame. When IsParentToChild is false, then Orientation performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

If you only want to perform one coordinate transformation, such as a transformation from the body frame to the sensor frame, you only need to specify a measurement parameter structure. If you want to perform multiple coordinate transformations, you need to specify an array of measurement parameter structures. To learn how to perform multiple transformations, see the “Convert Detections to objectDetection Format” (Sensor Fusion and Tracking Toolbox) example.

Data Types: struct

## Output Arguments

**measurement** — Measurement vector

real-valued  $M$ -by- $N$  matrix

Measurement vector, returned as an  $M$ -by- $N$  matrix.  $M$  is the dimension of the measurement and  $N$ , the number of measurement, is the same as the number of states. The form of each measurement depends upon which syntax you use.

- When the syntax does not use the `measurementParameters` argument, the measurement vector is  $[x, y, z]$  when the `frame` input argument is set to 'rectangular' and  $[az; el; r; rr]$  when the frame is set to 'spherical'.
- When the syntax uses the `measurementParameters` argument, the size of the measurement vector depends on the values of the `frame`, `HasVelocity`, and `HasElevation` fields in the `measurementParameters` structure.

frame	measurement		
'spherical'	Specifies the azimuth angle, $az$ , elevation angle, $el$ , range, $r$ , and range rate, $rr$ , of the object with respect to the local ego vehicle coordinate system. Positive values for range rate indicate that an object is moving away from the sensor.		
	<b>Spherical measurements</b>		
		HasElevation	
		false	true
	HasVelocity	false	$[az; r]$
		true	$[az; el; r; rr]$
	Angle units are in degrees, range units are in meters, and range rate units are in m/s.		
'rectangular'	Specifies the Cartesian position and velocity coordinates of the tracked object with respect to the ego vehicle coordinate system.		
	<b>Rectangular measurements</b>		
	HasVelocity	false	$[x; y; y]$
		true	$[x; y; z; vx; vy; vz]$
	Position units are in meters and velocity units are in m/s.		

Data Types: double

**bounds — Measurement residual wrapping bounds**

$M$ -by-2 real-valued matrix

Measurement residual wrapping bounds, returned as an  $M$ -by-2 real-valued matrix, where  $M$  is the dimension of the measurement. Each row of the matrix corresponds to the lower and upper bounds for the specific dimension in the measurement output.

The function returns different bound values based on the frame input.

- If the frame input is specified as 'Rectangular', each row of the matrix is [ -Inf Inf ], indicating the filter does not wrap the measurement residual in the filter.
- If the frame input is specified as 'Spherical', the returned bounds contains the bounds for specific measurement dimension based on the following:
  - When HasAzimuth = true, the matrix includes a row of [ -180 180 ], indicating the filter wraps the azimuth residual in the range of [ -180 180 ] in degrees.
  - When HasElevation = true, the matrix includes a row of [ -90 90 ], indicating the filter wraps the elevation residual in the range of [ -90 90 ] in degrees.
  - When HasRange = true, the matrix includes a row of [ -Inf Inf ], indicating the filter does not wrap the range residual.
  - When HasVelocity = true, the matrix includes a row of [ -Inf Inf ], indicating the filter does not wrap the range rate residual.

If you specify any of the options as false, the returned bounds does not contain the corresponding row. For example, if HasAzimuth = true, HasElevation = false, HasRange = true, HasVelocity = true, then bounds is returned as

```
-180  180
-Inf  Inf
-Inf  Inf
```

The filter wraps the measuring residuals based on this equation:

$$x_{wrap} = \text{mod}\left(x - \frac{a-b}{2}, b-a\right) + \frac{a-b}{2}$$

where  $x$  is the residual to wrap,  $a$  is the lower bound,  $b$  is the upper bound,  $\text{mod}$  is the modules after division function, and  $x_{wrap}$  is the wrapped residual.

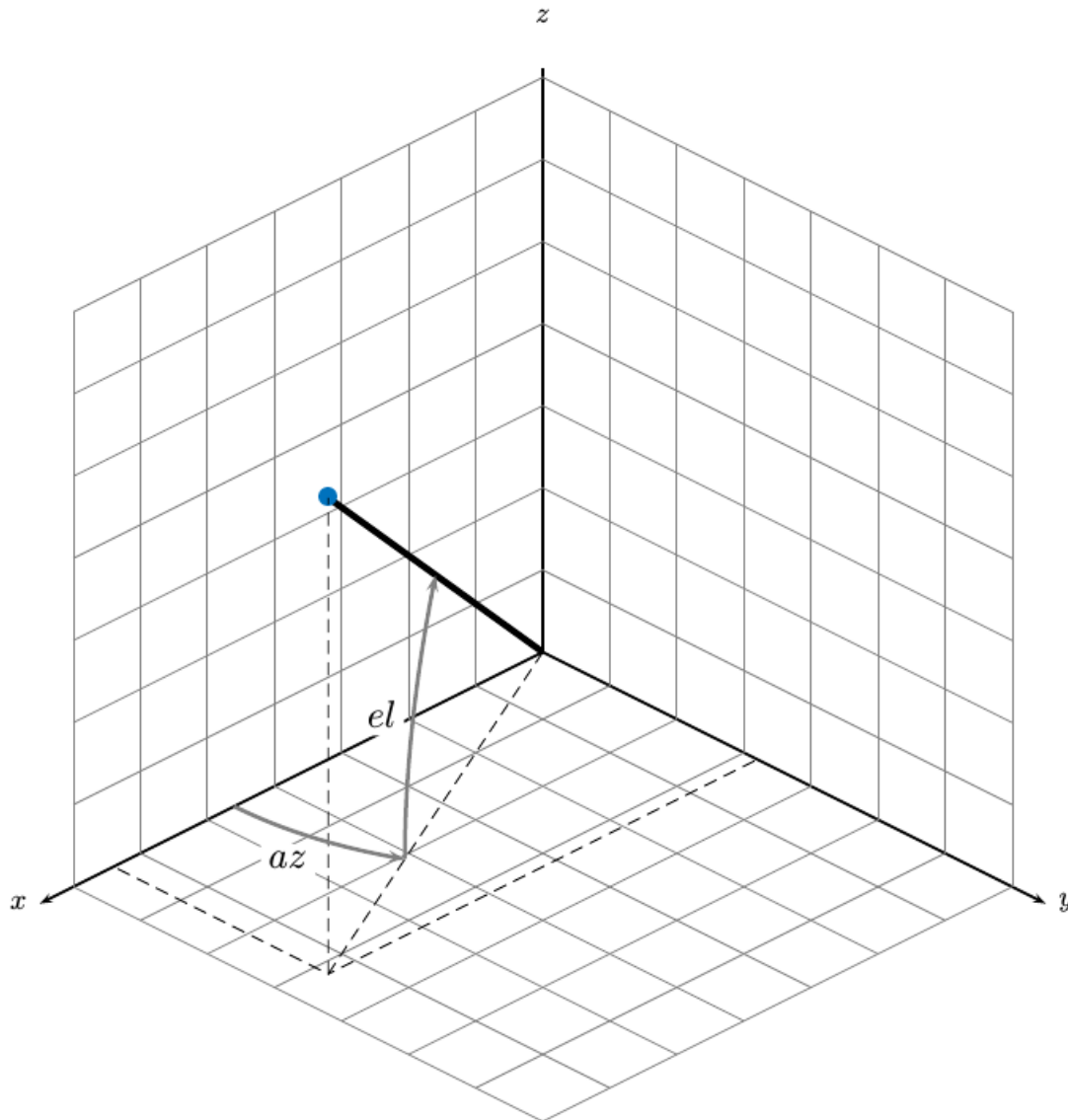
Data Types: single | double

## More About

### Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in the toolbox.

The azimuth angle of a vector is the angle between the  $x$ -axis and its orthogonal projection onto the  $xy$  plane. The angle is positive in going from the  $x$  axis toward the  $y$  axis. Azimuth angles lie between -180 and 180 degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going toward the positive  $z$ -axis from the  $xy$  plane.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

constacc | constaccjac | cameas | cameasjac | constturn | constturnjac | ctmeasjac | constvel | constveljac | cvmeas | cvmeasjac

### Objects

trackingKF | trackingEKF | trackingUKF



**Introduced in R2021a**

## ctmeasjac

Jacobian of measurement function for constant turn-rate motion

### Syntax

```
measurementjac = ctmeasjac(state)
measurementjac = ctmeasjac(state, frame)
measurementjac = ctmeasjac(state, frame, sensorpos)
measurementjac = ctmeasjac(state, frame, sensorpos, sensorvel)
measurementjac = ctmeasjac(state, frame, sensorpos, sensorvel, laxes)
measurementjac = ctmeasjac(state, measurementParameters)
```

### Description

`measurementjac = ctmeasjac(state)` returns the measurement Jacobian, `measurementjac`, for a constant turn-rate Kalman filter motion model in rectangular coordinates. `state` specifies the current state of the track.

`measurementjac = ctmeasjac(state, frame)` also specifies the measurement coordinate system, `frame`.

`measurementjac = ctmeasjac(state, frame, sensorpos)` also specifies the sensor position, `sensorpos`.

`measurementjac = ctmeasjac(state, frame, sensorpos, sensorvel)` also specifies the sensor velocity, `sensorvel`.

`measurementjac = ctmeasjac(state, frame, sensorpos, sensorvel, laxes)` also specifies the local sensor axes orientation, `laxes`.

`measurementjac = ctmeasjac(state, measurementParameters)` specifies the measurement parameters, `measurementParameters`.

### Examples

#### Measurement Jacobian of Constant Turn-Rate Motion in Rectangular Frame

Define the state of an object in 2-D constant turn-rate motion. The state is the position and velocity in each dimension, and the turn rate. Construct the measurement Jacobian in rectangular coordinates.

```
state = [1;10;2;20;5];
jacobian = ctmeasjac(state)
```

```
jacobian = 3×5
```

```
    1    0    0    0    0
    0    0    1    0    0
    0    0    0    0    0
```

### Measurement Jacobian of Constant Turn-Rate Motion in Spherical Frame

Define the state of an object in 2-D constant turn-rate motion. The state is the position and velocity in each dimension, and the turn rate. Compute the measurement Jacobian with respect to spherical coordinates.

```
state = [1;10;2;20;5];
measurementjac = ctmeasjac(state, 'spherical')

measurementjac = 4x5

-22.9183         0    11.4592         0         0
         0         0         0         0         0
  0.4472         0    0.8944         0         0
  0.0000    0.4472    0.0000    0.8944         0
```

### Measurement Jacobian of Constant Turn-Rate Object in Translated Spherical Frame

Define the state of an object in 2-D constant turn-rate motion. The state is the position and velocity in each dimension, and the turn rate. Compute the measurement Jacobian with respect to spherical coordinates centered at [5; -20; 0].

```
state = [1;10;2;20;5];
sensorpos = [5;-20;0];
measurementjac = ctmeasjac(state, 'spherical', sensorpos)

measurementjac = 4x5

-2.5210         0   -0.4584         0         0
         0         0         0         0         0
-0.1789         0    0.9839         0         0
 0.5903   -0.1789    0.1073    0.9839         0
```

### Measurement Jacobian of Constant Turn-Rate Object Using Measurement Parameters

Define the state of an object in 2-D constant turn-rate motion. The state is the position and velocity in each dimension, and the turn rate. Compute the measurement Jacobian with respect to spherical coordinates centered at [25; -40; 0].

```
state2d = [1;10;2;20;5];
sensorpos = [25,-40,0].';
frame = 'spherical';
sensorvel = [0;5;0];
laxes = eye(3);
measurementjac = ctmeasjac(state2d, frame, sensorpos, sensorvel, laxes)

measurementjac = 4x5
```

```

-1.0284      0    -0.5876      0      0
      0      0      0      0      0
-0.4961      0    0.8682      0      0
0.2894    -0.4961    0.1654    0.8682    0

```

Put the measurement parameters in a structure and use the alternative syntax.

```

measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel, ...
    'Orientation',laxes);
measurementjac = ctmeasjac(state2d,measparm)

```

```
measurementjac = 4x5
```

```

-1.0284      0    -0.5876      0      0
      0      0      0      0      0
-0.4961      0    0.8682      0      0
0.2894    -0.4961    0.1654    0.8682    0

```

## Input Arguments

### state — State vector

real-valued 5-element vector | real-valued 7-element vector | 5-by- $N$  real-valued matrix | 7-by- $N$  real-valued matrix

State vector for a constant turn-rate motion model in two or three spatial dimensions, specified as a real-valued vector or matrix.

- When specified as a 5-element vector, the state vector describes 2-D motion in the  $x$ - $y$  plane. You can specify the state vector as a row or column vector. The components of the state vector are  $[x; vx; y; vy; \omega]$  where  $x$  represents the  $x$ -coordinate and  $vx$  represents the velocity in the  $x$ -direction.  $y$  represents the  $y$ -coordinate and  $vy$  represents the velocity in the  $y$ -direction.  $\omega$  represents the turn rate.

When specified as a 5-by- $N$  matrix, each column represents a different state vector  $N$  represents the number of states.

- When specified as a 7-element vector, the state vector describes 3-D motion. You can specify the state vector as a row or column vector. The components of the state vector are  $[x; vx; y; vy; \omega; z; vz]$  where  $x$  represents the  $x$ -coordinate and  $vx$  represents the velocity in the  $x$ -direction.  $y$  represents the  $y$ -coordinate and  $vy$  represents the velocity in the  $y$ -direction.  $\omega$  represents the turn rate.  $z$  represents the  $z$ -coordinate and  $vz$  represents the velocity in the  $z$ -direction.

When specified as a 7-by- $N$  matrix, each column represents a different state vector.  $N$  represents the number of states.

Position coordinates are in meters. Velocity coordinates are in meters/second. Turn rate is in degrees/second.

Example: `[5;0.1;4;-0.2;0.01]`

Data Types: `double`

**frame — Measurement output frame**

'rectangular' (default) | 'spherical'

Measurement output frame, specified as 'rectangular' or 'spherical'. When the frame is 'rectangular', a measurement consists of  $x$ ,  $y$ , and  $z$  Cartesian coordinates. When specified as 'spherical', a measurement consists of azimuth, elevation, range, and range rate.

Data Types: char

**sensorpos — Sensor position**

[0;0;0] (default) | real-valued 3-by-1 column vector

Sensor position with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in meters.

Data Types: double

**sensorvel — Sensor velocity**

[0;0;0] (default) | real-valued 3-by-1 column vector

Sensor velocity with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in m/s.

Data Types: double

**laxes — Local sensor coordinate axes**

[1,0,0;0,1,0;0,0,1] (default) | 3-by-3 orthogonal matrix

Local sensor coordinate axes, specified as a 3-by-3 orthogonal matrix. Each column specifies the direction of the local  $x$ -,  $y$ -, and  $z$ -axes, respectively, with respect to the navigation frame. That is, the matrix is the rotation matrix from the global frame to the sensor frame.

Data Types: double

**measurementParameters — Measurement parameters**

structure | array of structure

Measurement parameters, specified as a structure or an array of structures. The fields of the structure are:

Field	Description	Example
Frame	Frame used to report measurements, specified as one of these values: <ul style="list-style-type: none"> <li>'rectangular' — Detections are reported in rectangular coordinates.</li> <li>'spherical' — Detections are reported in spherical coordinates.</li> </ul>	'spherical'

<b>Field</b>	<b>Description</b>	<b>Example</b>
OriginPosition	Position offset of the origin of the frame relative to the parent frame, specified as an [x y z] real-valued vector.	[0 0 0]
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, specified as a [vx vy vz] real-valued vector.	[0 0 0]
Orientation	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	[1 0 0; 0 1 0; 0 0 1]
HasAzimuth	Logical scalar indicating if azimuth is included in the measurement.	1
HasElevation	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if HasElevation is false, the reported measurements assume 0 degrees of elevation.	1
HasRange	Logical scalar indicating if range is included in the measurement.	1
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if Orientation performs a frame rotation from the parent coordinate frame to the child coordinate frame. When IsParentToChild is false, then Orientation performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

If you only want to perform one coordinate transformation, such as a transformation from the body frame to the sensor frame, you only need to specify a measurement parameter structure. If you want to perform multiple coordinate transformations, you need to specify an array of measurement parameter structures. To learn how to perform multiple transformations, see the “Convert Detections to objectDetection Format” (Sensor Fusion and Tracking Toolbox) example.

Data Types: `struct`

## Output Arguments

### **measurementjac** — Measurement Jacobian

real-valued 3-by-5 matrix | real-valued 4-by-5 matrix

Measurement Jacobian, returned as a real-valued 3-by-5 or 4-by-5 matrix. The row dimension and interpretation depend on value of the `frame` argument.

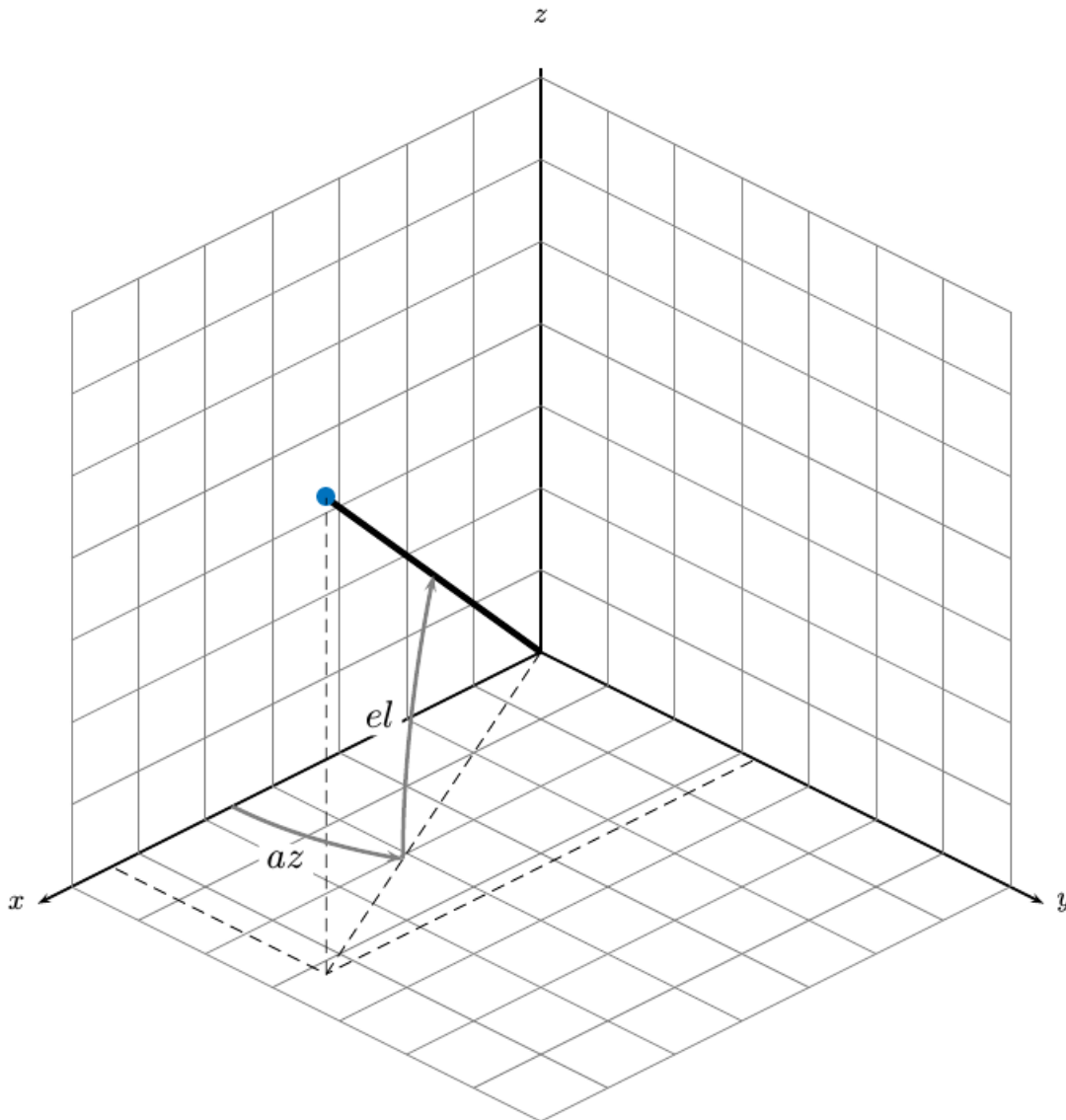
Frame	Measurement Jacobian
'rectangular'	Jacobian of the measurements $[x; y; z]$ with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
'spherical'	Jacobian of the measurement vector $[az; el; r; rr]$ with respect to the state vector. Measurement vector components specify the azimuth angle, elevation angle, range, and range rate of the object with respect to the local sensor coordinate system. Angle units are in degrees. Range units are in meters and range rate units are in meters/second.

## More About

### **Azimuth and Elevation Angle Definitions**

Define the azimuth and elevation angles used in the toolbox.

The azimuth angle of a vector is the angle between the  $x$ -axis and its orthogonal projection onto the  $xy$  plane. The angle is positive in going from the  $x$  axis toward the  $y$  axis. Azimuth angles lie between  $-180$  and  $180$  degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going toward the positive  $z$ -axis from the  $xy$  plane.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

constacc | constaccjac | cameas | cameasjac | constturn | constturnjac | ctmeas | constvel | constveljac | cvmeas | cvmeasjac

### Objects

trackingKF | trackingEKF | trackingUKF



**Introduced in R2021a**

## cvmeas

Measurement function for constant velocity motion

### Syntax

```
measurement = cvmeas(state)
measurement = cvmeas(state, frame)
measurement = cvmeas(state, frame, sensorpos)
measurement = cvmeas(state, frame, sensorpos, sensorvel)
measurement = cvmeas(state, frame, sensorpos, sensorvel, laxes)
measurement = cvmeas(state, measurementParameters)
[measurement, bounds] = cvmeas( ___ )
```

### Description

`measurement = cvmeas(state)` returns the measurement for a constant-velocity Kalman filter motion model in rectangular coordinates. The `state` argument specifies the current state of the tracking filter.

`measurement = cvmeas(state, frame)` also specifies the measurement coordinate system, `frame`.

`measurement = cvmeas(state, frame, sensorpos)` also specifies the sensor position, `sensorpos`.

`measurement = cvmeas(state, frame, sensorpos, sensorvel)` also specifies the sensor velocity, `sensorvel`.

`measurement = cvmeas(state, frame, sensorpos, sensorvel, laxes)` specifies the local sensor axes orientation, `laxes`.

`measurement = cvmeas(state, measurementParameters)` specifies the measurement parameters, `measurementParameters`.

`[measurement, bounds] = cvmeas( ___ )` returns the measurement bounds, used by a tracking filter (`trackingEKF` or `trackingUKF`) in residual calculations. See the `HasMeasurementWrapping` of the filter object for more details.

### Examples

#### Create Measurement from Constant-Velocity Object in Rectangular Frame

Define the state of an object in 2-D constant-velocity motion. The state is the position and velocity in both dimensions. The measurements are in rectangular coordinates.

```
state = [1;10;2;20];
measurement = cvmeas(state)
```

```
measurement = 3×1
```

```
1
2
0
```

The z-component of the measurement is zero.

### Create Measurement from Constant Velocity Object in Spherical Frame

Define the state of an object in 2-D constant-velocity motion. The state is the position and velocity in each spatial dimension. The measurements are in spherical coordinates.

```
state = [1;10;2;20];
measurement = cvmeas(state, 'spherical')
```

```
measurement = 4×1
```

```
63.4349
      0
  2.2361
 22.3607
```

The elevation of the measurement is zero and the range rate is positive. These results indicate that the object is moving away from the sensor.

### Create Measurement from Constant-Velocity Object in Translated Spherical Frame

Define the state of an object in 2-D constant-velocity motion. The state consists of position and velocity in each spatial dimension. The measurements are in spherical coordinates with respect to a frame located at (20;40;0) meters.

```
state = [1;10;2;20];
measurement = cvmeas(state, 'spherical', [20;40;0])
```

```
measurement = 4×1
```

```
-116.5651
      0
  42.4853
 -22.3607
```

The elevation of the measurement is zero and the range rate is negative. These results indicate that the object is moving toward the sensor.

### Create Measurement from Constant-Velocity Object Using Measurement Parameters

Define the state of an object in 2-D constant-velocity motion. The state consists of position and velocity in each spatial dimension. The measurements are in spherical coordinates with respect to a frame located at (20;40;0) meters.

```
state2d = [1;10;2;20];
frame = 'spherical';
sensorpos = [20;40;0];
sensorvel = [0;5;0];
laxes = eye(3);
measurement = cvmeas(state2d,frame,sensorpos,sensorvel,laxes)
```

```
measurement = 4×1
```

```
-116.5651
      0
  42.4853
 -17.8885
```

The elevation of the measurement is zero and the range rate is negative. These results indicate that the object is moving toward the sensor.

Put the measurement parameters in a structure and use the alternative syntax.

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel, ...
    'Orientation',laxes);
measurement = cvmeas(state2d,measparm)
```

```
measurement = 4×1
```

```
-116.5651
      0
  42.4853
 -17.8885
```

### Display Residual Wrapping Bounds for cvmeas

Specify a 2-D state and specify a measurement structure such that the function outputs azimuth, range, and range-rate measurements.

```
state = [10 1 10 1]'; % [x vx y vy]'
mp = struct("Frame","Spherical", ...
    "HasAzimuth",true, ...
    "HasElevation",false, ...
    "HasRange",true, ...
    "HasVelocity",false);
```

Output the measurement and wrapping bounds using the cvmeas function.

```
[measure,bounds] = cvmeas(state,mp)
```

```
measure = 2×1
```

```
45.0000
14.1421
```

```
bounds = 2x2
```

```
-180  180
-Inf  Inf
```

## Input Arguments

### state — Kalman filter state vector

real-valued  $2D$ -by- $N$  matrix

Kalman filter state vector for constant-velocity motion, specified as a real-valued  $2D$ -by- $N$  matrix.  $D$  is the number of spatial degrees of freedom of motion and  $N$  is the number states. The `state` is expected to be Cartesian state. For each spatial degree of motion, the state vector, as a column of the state matrix, takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	[x; vx]
2-D	[x; vx; y; vy]
3-D	[x; vx; y; vy; z; vz]

For example, `x` represents the  $x$ -coordinate and `vx` represents the velocity in the  $x$ -direction. If the motion model is 1-D, values along the  $y$  and  $z$  axes are assumed to be zero. If the motion model is 2-D, values along the  $z$  axis are assumed to be zero. Position coordinates are in meters and velocity coordinates are in meters/sec.

Example: [5; .1; 0; -.2; -3; .05]

Data Types: `single` | `double`

### frame — Measurement output frame

'rectangular' (default) | 'spherical'

Measurement output frame, specified as 'rectangular' or 'spherical'. When the frame is 'rectangular', a measurement consists of  $x$ ,  $y$ , and  $z$  Cartesian coordinates. When specified as 'spherical', a measurement consists of azimuth, elevation, range, and range rate.

Data Types: `char`

### sensorpos — Sensor position

[0; 0; 0] (default) | real-valued 3-by-1 column vector

Sensor position with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in meters.

Data Types: `double`

### sensorvel — Sensor velocity

[0; 0; 0] (default) | real-valued 3-by-1 column vector

Sensor velocity with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in m/s.

Data Types: double

**laxes — Local sensor coordinate axes**

[1,0,0;0,1,0;0,0,1] (default) | 3-by-3 orthogonal matrix

Local sensor coordinate axes, specified as a 3-by-3 orthogonal matrix. Each column specifies the direction of the local x-, y-, and z-axes, respectively, with respect to the navigation frame. That is, the matrix is the rotation matrix from the global frame to the sensor frame.

Data Types: double

**measurementParameters — Measurement parameters**

structure | array of structure

Measurement parameters, specified as a structure or an array of structures. The fields of the structure are:

Field	Description	Example
Frame	Frame used to report measurements, specified as one of these values: <ul style="list-style-type: none"> <li>'rectangular' — Detections are reported in rectangular coordinates.</li> <li>'spherical' — Detections are reported in spherical coordinates.</li> </ul>	'spherical'
OriginPosition	Position offset of the origin of the frame relative to the parent frame, specified as an [x y z] real-valued vector.	[0 0 0]
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, specified as a [vx vy vz] real-valued vector.	[0 0 0]
Orientation	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	[1 0 0; 0 1 0; 0 0 1]
HasAzimuth	Logical scalar indicating if azimuth is included in the measurement.	1

Field	Description	Example
HasElevation	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if HasElevation is false, the reported measurements assume 0 degrees of elevation.	1
HasRange	Logical scalar indicating if range is included in the measurement.	1
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if Orientation performs a frame rotation from the parent coordinate frame to the child coordinate frame. When IsParentToChild is false, then Orientation performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

If you only want to perform one coordinate transformation, such as a transformation from the body frame to the sensor frame, you only need to specify a measurement parameter structure. If you want to perform multiple coordinate transformations, you need to specify an array of measurement parameter structures. To learn how to perform multiple transformations, see the “Convert Detections to objectDetection Format” (Sensor Fusion and Tracking Toolbox) example.

Data Types: struct

## Output Arguments

### measurement — Measurement vector

real-valued  $M$ -by- $N$  matrix

Measurement vector, returned as an  $M$ -by- $N$  matrix.  $M$  is the dimension of the measurement and  $N$ , the number of measurement, is the same as the number of states. The form of each measurement depends upon which syntax you use.

- When the syntax does not use the `measurementParameters` argument, the measurement vector is  $[x, y, z]$  when the `frame` input argument is set to 'rectangular' and  $[az; el; r; rr]$  when the frame is set to 'spherical'.
- When the syntax uses the `measurementParameters` argument, the size of the measurement vector depends on the values of the `frame`, `HasVelocity`, and `HasElevation` fields in the `measurementParameters` structure.

frame	measurement															
'spherical'	<p>Specifies the azimuth angle, <i>az</i>, elevation angle, <i>el</i>, range, <i>r</i>, and range rate, <i>rr</i>, of the object with respect to the local ego vehicle coordinate system. Positive values for range rate indicate that an object is moving away from the sensor.</p> <p><b>Spherical measurements</b></p> <table border="1"> <thead> <tr> <th></th> <th></th> <th colspan="2">HasElevation</th> </tr> <tr> <th></th> <th></th> <th>false</th> <th>true</th> </tr> </thead> <tbody> <tr> <td rowspan="2"><b>HasVelocity</b></td> <td>false</td> <td>[az; r]</td> <td>[az; el; r]</td> </tr> <tr> <td>true</td> <td>[az; r; rr]</td> <td>[az; el; r; rr]</td> </tr> </tbody> </table> <p>Angle units are in degrees, range units are in meters, and range rate units are in m/s.</p>			HasElevation				false	true	<b>HasVelocity</b>	false	[az; r]	[az; el; r]	true	[az; r; rr]	[az; el; r; rr]
		HasElevation														
		false	true													
<b>HasVelocity</b>	false	[az; r]	[az; el; r]													
	true	[az; r; rr]	[az; el; r; rr]													
'rectangular'	<p>Specifies the Cartesian position and velocity coordinates of the tracked object with respect to the ego vehicle coordinate system.</p> <p><b>Rectangular measurements</b></p> <table border="1"> <thead> <tr> <th rowspan="2"><b>HasVelocity</b></th> <th>false</th> <th>[x; y; y]</th> </tr> </thead> <tbody> <tr> <th>true</th> <th>[x; y; z; vx; vy; vz]</th> </tr> </tbody> </table> <p>Position units are in meters and velocity units are in m/s.</p>	<b>HasVelocity</b>	false	[x; y; y]	true	[x; y; z; vx; vy; vz]										
<b>HasVelocity</b>	false		[x; y; y]													
	true	[x; y; z; vx; vy; vz]														

Data Types: double

**bounds — Measurement residual wrapping bounds**

*M*-by-2 real-valued matrix

Measurement residual wrapping bounds, returned as an *M*-by-2 real-valued matrix, where *M* is the dimension of the measurement. Each row of the matrix corresponds to the lower and upper bounds for the specific dimension in the measurement output.

The function returns different bound values based on the `frame` input.

- If the `frame` input is specified as 'Rectangular', each row of the matrix is  $[-Inf \ Inf]$ , indicating the filter does not wrap the measurement residual in the filter.



- If the frame input is specified as 'Spherical', the returned bounds contains the bounds for specific measurement dimension based on the following:
  - When `HasAzimuth = true`, the matrix includes a row of `[-180 180]`, indicating the filter wraps the azimuth residual in the range of `[-180 180]` in degrees.
  - When `HasElevation = true`, the matrix includes a row of `[-90 90]`, indicating the filter wraps the elevation residual in the range of `[-90 90]` in degrees.
  - When `HasRange = true`, the matrix includes a row of `[-Inf Inf]`, indicating the filter does not wrap the range residual.
  - When `HasVelocity = true`, the matrix includes a row of `[-Inf Inf]`, indicating the filter does not wrap the range rate residual.

If you specify any of the options as false, the returned bounds does not contain the corresponding row. For example, if `HasAzimuth = true`, `HasElevation = false`, `HasRange = true`, `HasVelocity = true`, then bounds is returned as

```
-180  180
-Inf  Inf
-Inf  Inf
```

The filter wraps the measuring residuals based on this equation:

$$x_{wrap} = \text{mod}\left(x - \frac{a-b}{2}, b-a\right) + \frac{a-b}{2}$$

where  $x$  is the residual to wrap,  $a$  is the lower bound,  $b$  is the upper bound,  $\text{mod}$  is the modules after division function, and  $x_{wrap}$  is the wrapped residual.

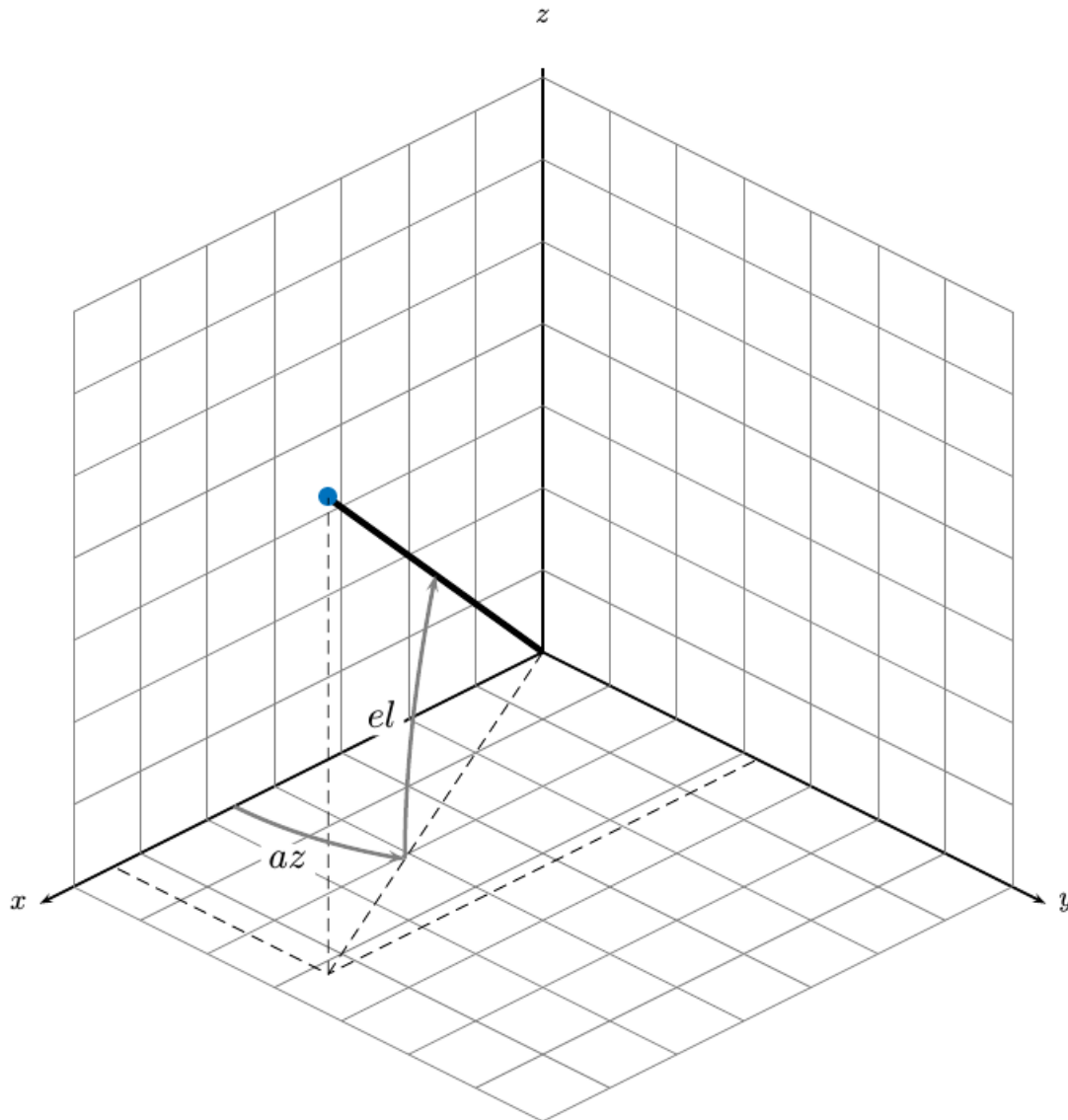
Data Types: `single` | `double`

## More About

### Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in the toolbox.

The azimuth angle of a vector is the angle between the  $x$ -axis and its orthogonal projection onto the  $xy$  plane. The angle is positive in going from the  $x$  axis toward the  $y$  axis. Azimuth angles lie between  $-180$  and  $180$  degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going toward the positive  $z$ -axis from the  $xy$  plane.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

constacc | constaccjac | cameas | cameasjac | constturn | constturnjac | ctmeas | ctmeasjac | constvel | constveljac | cvmeasjac

### Objects

trackingKF | trackingEKF | trackingUKF

**Introduced in R2021a**

## cvmeasjac

Jacobian of measurement function for constant velocity motion

### Syntax

```
measurementjac = cvmeasjac(state)
measurementjac = cvmeasjac(state, frame)
measurementjac = cvmeasjac(state, frame, sensorpos)
measurementjac = cvmeasjac(state, frame, sensorpos, sensorvel)
measurementjac = cvmeasjac(state, frame, sensorpos, sensorvel, laxes)
measurementjac = cvmeasjac(state, measurementParameters)
```

### Description

`measurementjac = cvmeasjac(state)` returns the measurement Jacobian for constant-velocity Kalman filter motion model in rectangular coordinates. `state` specifies the current state of the tracking filter.

`measurementjac = cvmeasjac(state, frame)` also specifies the measurement coordinate system, `frame`.

`measurementjac = cvmeasjac(state, frame, sensorpos)` also specifies the sensor position, `sensorpos`.

`measurementjac = cvmeasjac(state, frame, sensorpos, sensorvel)` also specifies the sensor velocity, `sensorvel`.

`measurementjac = cvmeasjac(state, frame, sensorpos, sensorvel, laxes)` also specifies the local sensor axes orientation, `laxes`.

`measurementjac = cvmeasjac(state, measurementParameters)` specifies the measurement parameters, `measurementParameters`.

### Examples

#### Measurement Jacobian of Constant-Velocity Object in Rectangular Frame

Define the state of an object in 2-D constant-velocity motion. The state is the position and velocity in each spatial dimension. Construct the measurement Jacobian in rectangular coordinates.

```
state = [1;10;2;20];
jacobian = cvmeasjac(state)
```

```
jacobian = 3×4
```

```
    1    0    0    0
    0    0    1    0
    0    0    0    0
```

### Measurement Jacobian of Constant-Velocity Motion in Spherical Frame

Define the state of an object in 2-D constant-velocity motion. The state is the position and velocity in each dimension. Compute the measurement Jacobian with respect to spherical coordinates.

```
state = [1;10;2;20];
measurementjac = cvmeasjac(state, 'spherical')
```

```
measurementjac = 4x4
```

```
-22.9183      0    11.4592      0
      0      0      0      0
  0.4472      0    0.8944      0
  0.0000    0.4472    0.0000    0.8944
```

### Measurement Jacobian of Constant-Velocity Object in Translated Spherical Frame

Define the state of an object in 2-D constant-velocity motion. The state is the position and velocity in each spatial dimension. Compute the measurement Jacobian with respect to spherical coordinates centered at (5;-20;0) meters.

```
state = [1;10;2;20];
sensorpos = [5;-20;0];
measurementjac = cvmeasjac(state, 'spherical', sensorpos)
```

```
measurementjac = 4x4
```

```
-2.5210      0   -0.4584      0
      0      0      0      0
-0.1789      0    0.9839      0
  0.5903   -0.1789    0.1073    0.9839
```

### Create Measurement Jacobian for Constant-Velocity Object Using Measurement Parameters

Define the state of an object in 2-D constant-velocity motion. The state consists of position and velocity in each spatial dimension. The measurements are in spherical coordinates with respect to a frame located at (20;40;0) meters.

```
state2d = [1;10;2;20];
frame = 'spherical';
sensorpos = [20;40;0];
sensorvel = [0;5;0];
laxes = eye(3);
measurementjac = cvmeasjac(state2d, frame, sensorpos, sensorvel, laxes)
```

```
measurementjac = 4x4
```

```
 1.2062      0   -0.6031      0
      0      0      0      0
```

```
-0.4472      0    -0.8944      0
 0.0471    -0.4472    -0.0235    -0.8944
```

Put the measurement parameters in a structure and use the alternative syntax.

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel, ...
    'Orientation',laxes);
measurementjac = cvmeasjac(state2d,measparm)
```

measurementjac = 4x4

```
 1.2062      0    -0.6031      0
      0      0      0      0
 -0.4472      0    -0.8944      0
 0.0471    -0.4472    -0.0235    -0.8944
```

## Input Arguments

### state — Kalman filter state vector

real-valued 2N-element vector

Kalman filter state vector for constant-velocity motion, specified as a real-valued 2N-element column vector where N is the number of spatial degrees of freedom of motion. The state is expected to be Cartesian state. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	[x;vx]
2-D	[x;vx;y;vy]
3-D	[x;vx;y;vy;z;vz]

For example, x represents the x-coordinate and vx represents the velocity in the x-direction. If the motion model is 1-D, values along the y and z axes are assumed to be zero. If the motion model is 2-D, values along the z axis are assumed to be zero. Position coordinates are in meters and velocity coordinates are in meters/sec.

Example: [5;.1;0;-.2;-3;.05]

Data Types: single | double

### frame — Measurement output frame

'rectangular' (default) | 'spherical'

Measurement output frame, specified as 'rectangular' or 'spherical'. When the frame is 'rectangular', a measurement consists of x, y, and z Cartesian coordinates. When specified as 'spherical', a measurement consists of azimuth, elevation, range, and range rate.

Data Types: char

### sensorpos — Sensor position

[0;0;0] (default) | real-valued 3-by-1 column vector

Sensor position with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in meters.

Data Types: double

### **sensorvel — Sensor velocity**

[0;0;0] (default) | real-valued 3-by-1 column vector

Sensor velocity with respect to the navigation frame, specified as a real-valued 3-by-1 column vector. Units are in m/s.

Data Types: double

### **laxes — Local sensor coordinate axes**

[1,0,0;0,1,0;0,0,1] (default) | 3-by-3 orthogonal matrix

Local sensor coordinate axes, specified as a 3-by-3 orthogonal matrix. Each column specifies the direction of the local x-, y-, and z-axes, respectively, with respect to the navigation frame. That is, the matrix is the rotation matrix from the global frame to the sensor frame.

Data Types: double

### **measurementParameters — Measurement parameters**

structure | array of structure

Measurement parameters, specified as a structure or an array of structures. The fields of the structure are:

Field	Description	Example
Frame	Frame used to report measurements, specified as one of these values: <ul style="list-style-type: none"> <li>'rectangular' — Detections are reported in rectangular coordinates.</li> <li>'spherical' — Detections are reported in spherical coordinates.</li> </ul>	'spherical'
OriginPosition	Position offset of the origin of the frame relative to the parent frame, specified as an [x y z] real-valued vector.	[0 0 0]
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, specified as a [vx vy vz] real-valued vector.	[0 0 0]
Orientation	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	[1 0 0; 0 1 0; 0 0 1]
HasAzimuth	Logical scalar indicating if azimuth is included in the measurement.	1

Field	Description	Example
HasElevation	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if HasElevation is false, the reported measurements assume 0 degrees of elevation.	1
HasRange	Logical scalar indicating if range is included in the measurement.	1
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if Orientation performs a frame rotation from the parent coordinate frame to the child coordinate frame. When IsParentToChild is false, then Orientation performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

If you only want to perform one coordinate transformation, such as a transformation from the body frame to the sensor frame, you only need to specify a measurement parameter structure. If you want to perform multiple coordinate transformations, you need to specify an array of measurement parameter structures. To learn how to perform multiple transformations, see the “Convert Detections to objectDetection Format” (Sensor Fusion and Tracking Toolbox) example.

Data Types: struct

## Output Arguments

### measurementjac — Measurement Jacobian

real-valued 3-by-*N* matrix | real-valued 4-by-*N* matrix

Measurement Jacobian, specified as a real-valued 3-by-*N* or 4-by-*N* matrix. *N* is the dimension of the state vector. The first dimension and meaning depend on value of the frame argument.



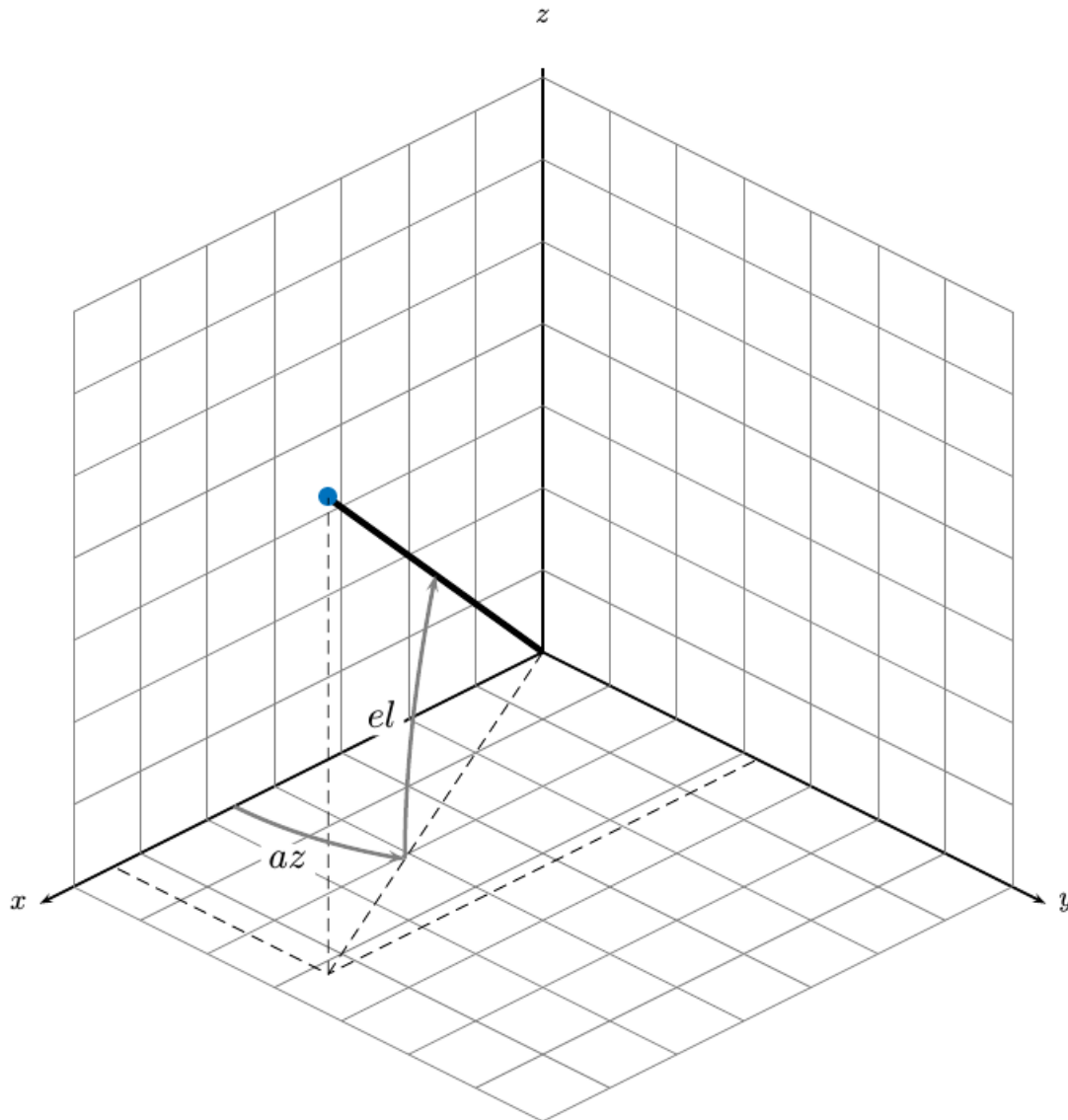
Frame	Measurement Jacobian
'rectangular'	Jacobian of the measurements $[x; y; z]$ with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
'spherical'	Jacobian of the measurement vector $[az; el; r; rr]$ with respect to the state vector. Measurement vector components specify the azimuth angle, elevation angle, range, and range rate of the object with respect to the local sensor coordinate system. Angle units are in degrees. Range units are in meters and range rate units are in meters/second.

## More About

### Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in the toolbox.

The azimuth angle of a vector is the angle between the  $x$ -axis and its orthogonal projection onto the  $xy$  plane. The angle is positive in going from the  $x$  axis toward the  $y$  axis. Azimuth angles lie between  $-180$  and  $180$  degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the  $xy$ -plane. The angle is positive when going toward the positive  $z$ -axis from the  $xy$  plane.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

constacc | constaccjac | cameas | cameasjac | constturn | constturnjac | ctmeas | ctmeasjac | constvel | constveljac | cvmeas

### Objects

trackingKF | trackingEKF | trackingUKF

**Introduced in R2021a**

## initcaabf

Create constant acceleration alpha-beta tracking filter from detection report

### Syntax

```
abf = initcaabf(detection)
```

### Description

`abf = initcaabf(detection)` initializes a constant acceleration alpha-beta tracking filter for object tracking based on information provided in `detection`.

The function initializes a constant acceleration state with the same convention as `constacc` and `cameas`,  $[x; v_x; a_x; y; v_y; a_y; z; v_z; a_z]$ .

### Examples

#### Creating Constant Acceleration trackingABF Object from Detection

Create an `objectDetection` with a position measurement at  $x=1$ ,  $y=3$  and a measurement noise of  $[1 \ 0.2; 0.2 \ 2]$ ;

```
detection = objectDetection(0, [1;3], 'MeasurementNoise', [1 0.2;0.2 2]);
```

Use `initccabf` to create a `trackingABF` filter initialized at the provided position and using the measurement noise defined above.

```
ABF = initcaabf(detection);
```

Check the values of the state and measurement noise. Verify that the filter state, `ABF.State`, has the same position components as the `Detection.Measurement`. Verify that the filter measurement noise, `ABF.MeasurementNoise`, is the same as the `Detection.MeasurementNoise` values.

`ABF.State`

```
ans = 6×1
```

```
1
0
0
3
0
0
```

`ABF.MeasurementNoise`

```
ans = 2×2
```

```
1.0000    0.2000
0.2000    2.0000
```

## Input Arguments

### **detection** — Detection report

`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **abf** — Constant velocity alpha-beta filter

`trackingABF` object

Constant acceleration alpha-beta tracking filter for object tracking, returned as a `trackingABF` object.

## Algorithms

- The function computes the process noise matrix assuming a unit standard deviation for the acceleration change rate.
- You can use this function as the `FilterInitializationFcn` property of trackers.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`trackingABF` | `objectDetection` | `trackingKF` | `trackingEKF` | `trackingUKF`

**Introduced in R2021a**

## initcvabf

Create constant velocity tracking alpha-beta filter from detection report

### Syntax

```
abf = initcvabf(detection)
```

### Description

`abf = initcvabf(detection)` initializes a constant velocity alpha-beta filter for object tracking based on information provided in `detection`.

The function initializes a constant velocity state with the same convention as `constvel` and `cvmeas`,  $[x; v_x; y; v_y; z; v_z]$ .

### Examples

#### Creating trackingABF Object from Detection

Create an `objectDetection` with a position measurement at  $x=1$ ,  $y=3$  and a measurement noise of  $[1 \ 0.2; 0.2 \ 2]$ ;

```
detection = objectDetection(0, [1;3], 'MeasurementNoise', [1 0.2;0.2 2]);
```

Use `initcvabf` to create a `trackingABF` filter initialized at the provided position and using the measurement noise defined above.

```
ABF = initcvabf(detection);
```

Check the values of the state and measurement noise. Verify that the filter state, `ABF.State`, has the same position components as the `Detection.Measurement`. Verify that the filter measurement noise, `ABF.MeasurementNoise`, is the same as the `Detection.MeasurementNoise` values.

`ABF.State`

```
ans = 4×1
```

```
    1  
    0  
    3  
    0
```

`ABF.MeasurementNoise`

```
ans = 2×2
```

```
    1.0000    0.2000  
    0.2000    2.0000
```

## Input Arguments

### **detection** — Detection report

objectDetection object

Detection report, specified as an objectDetection object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **abf** — Constant velocity alpha-beta filter

trackingABF object

Constant velocity alpha-beta tracking filter for object tracking, returned as a trackingABF object.

## Algorithms

- The function computes the process noise matrix assuming a unit acceleration standard deviation.
- You can use this function as the `FilterInitializationFcn` property of trackers.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

trackingABF | objectDetection | trackingKF | trackingEKF | trackingUKF

**Introduced in R2021a**

## initcaekf

Create constant-acceleration extended Kalman filter from detection report

### Syntax

```
filter = initcaekf(detection)
```

### Description

`filter = initcaekf(detection)` creates and initializes a constant-acceleration extended Kalman filter from information contained in a `detection` report. For more information about the extended Kalman filter, see `trackingEKF`.

The function initializes a constant acceleration state with the same convention as `constacc` and `cameas`,  $[x; v_x; a_x; y; v_y; a_y; z; v_z; a_z]$ .

### Examples

#### Initialize 3-D Constant-Acceleration Extended Kalman Filter

Create and initialize a 3-D constant-acceleration extended Kalman filter object from an initial detection report.

Create the detection report from an initial 3-D measurement,  $(-200;30;0)$ , of the object position. Assume uncorrelated measurement noise.

```
detection = objectDetection(0,[-200;-30;0], 'MeasurementNoise', 2.1*eye(3), ...
    'SensorIndex', 1, 'ObjectClassID', 1, 'ObjectAttributes', {'Car', 2});
```

Create the new filter from the detection report and display its properties.

```
filter = initcaekf(detection)

filter =
    trackingEKF with properties:

        State: [9x1 double]
        StateCovariance: [9x9 double]

        StateTransitionFcn: @constacc
        StateTransitionJacobianFcn: @constaccjac
        ProcessNoise: [3x3 double]
        HasAdditiveProcessNoise: 0

        MeasurementFcn: @cameas
        MeasurementJacobianFcn: @cameasjac
        HasMeasurementWrapping: 1
        MeasurementNoise: [3x3 double]
        HasAdditiveMeasurementNoise: 1

        MaxNumOOSMSteps: 0
```



```
EnableSmoothing: 0
```

Show the filter state.

```
filter.State
```

```
ans = 9×1
```

```
-200
  0
  0
 -30
  0
  0
  0
  0
  0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 9×9
```

```
 2.1000    0    0    0    0    0    0    0    0
    0 100.0000    0    0    0    0    0    0    0
    0    0 100.0000    0    0    0    0    0    0
    0    0    0  2.1000    0    0    0    0    0
    0    0    0    0 100.0000    0    0    0    0
    0    0    0    0    0 100.0000    0    0    0
    0    0    0    0    0    0  2.1000    0    0
    0    0    0    0    0    0    0 100.0000    0
    0    0    0    0    0    0    0    0 100.0000
```

### Create 3D Constant Acceleration EKF from Spherical Measurement

Initialize a 3D constant-acceleration extended Kalman filter from an initial detection report made from an initial measurement in spherical coordinates. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to  $45^\circ$ , the elevation to  $22^\circ$ , the range to 1000 meters, and the range rate to -4.0 m/s.

```
frame = 'spherical';
sensorpos = [25,-40,-10].';
sensorvel = [0;5;0];
laxes = eye(3);
```

Create the measurement parameters structure. Set `'HasVelocity'` and `'HasElevation'` to true. Then, the measurement vector consists of azimuth, elevation, range, and range rate.

```
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',true, ...
```

```
    'HasElevation',true);
meas = [45;22;1000;-4];
measnoise = diag([3.0,2.5,2,1.0].^2);
detection = objectDetection(0,meas,'MeasurementNoise', ...
    measnoise,'MeasurementParameters',measparms)
```

```
detection =
  objectDetection with properties:

        Time: 0
    Measurement: [4x1 double]
  MeasurementNoise: [4x4 double]
        SensorIndex: 1
    ObjectClassID: 0
  MeasurementParameters: [1x1 struct]
    ObjectAttributes: {}
```

```
filter = initcaekf(detection);
```

Display the state vector.

```
disp(filter.State)
```

```
680.6180
-2.6225
 0
615.6180
 2.3775
 0
364.6066
-1.4984
 0
```

## Input Arguments

### **detection** – Detection report

objectDetection object

Detection report, specified as an objectDetection object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **filter** – Extended Kalman filter

trackingEKF object

Extended Kalman filter, returned as a trackingEKF object.

## Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration-rate standard deviation of 1 m/s<sup>3</sup>.

- You can use this function as the `FilterInitializationFcn` property of a `radarTracker` object.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`initctekf` | `initctukf` | `initcvkf` | `initcvekf` | `initcvukf` | `initcakf` | `initcaukf`

### Objects

`objectDetection` | `trackingKF` | `trackingEKF` | `trackingUKF` | `radarTracker`

**Introduced in R2021a**

## initcakf

Create constant-acceleration linear Kalman filter from detection report

### Syntax

```
filter = initcakf(detection)
```

### Description

`filter = initcakf(detection)` creates and initializes a constant-acceleration linear Kalman filter from information contained in a `detection` report. For more information about the linear Kalman filter, see `trackingKF`.

The function initializes a constant acceleration state with the same convention as `constacc` and `cameas`,  $[x; v_x; a_x; y; v_y; a_y; z; v_z; a_z]$ .

### Examples

#### Initialize 2-D Constant-Acceleration Linear Kalman Filter

Create and initialize a 2-D constant-acceleration linear Kalman filter object from an initial detection report.

Create the detection report from an initial 2-D measurement, (10,-5), of the object position. Assume uncorrelated measurement noise.

```
detection = objectDetection(0,[10;-5],'MeasurementNoise',eye(2), ...  
    'SensorIndex',1,'ObjectClassID',1,'ObjectAttributes',{'Car',5});
```

Create the new filter from the detection report.

```
filter = initcakf(detection);
```

Show the filter state.

```
filter.State
```

```
ans = 6×1
```

```
    10  
     0  
     0  
    -5  
     0  
     0
```

Show the state transition model.

```
filter.StateTransitionModel
```

ans = 6×6

```

1.0000    1.0000    0.5000         0         0         0
         0    1.0000    1.0000         0         0         0
         0         0    1.0000         0         0         0
         0         0         0    1.0000    1.0000    0.5000
         0         0         0         0    1.0000    1.0000
         0         0         0         0         0    1.0000

```

## Input Arguments

### **detection** — Detection report

objectDetection object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise',[1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **filter** — Linear Kalman filter

trackingKF object

Linear Kalman filter, returned as a `trackingKF` object.

## Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration rate standard deviation of 1 m/s<sup>3</sup>.
- You can use this function as the `FilterInitializationFcn` property of a `radarTracker` object.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

`initcaekf` | `initcaukf` | `initctekf` | `initctukf` | `initcvkf` | `initcvekf` | `initcvukf`

### **Objects**

`objectDetection` | `trackingKF` | `trackingEKF` | `trackingUKF` | `radarTracker`

**Introduced in R2021a**

## initcaukf

Create constant-acceleration unscented Kalman filter from detection report

### Syntax

```
filter = initcaukf(detection)
```

### Description

`filter = initcaukf(detection)` creates and initializes a constant-acceleration unscented Kalman filter from information contained in a `detection` report. For more information about the unscented Kalman filter, see `trackingUKF`.

The function initializes a constant acceleration state with the same convention as `constacc` and `cameas`,  $[x; v_x; a_x; y; v_y; a_y; z; v_z; a_z]$ .

### Examples

#### Initialize 3-D Constant-Acceleration Unscented Kalman Filter

Create and initialize a 3-D constant-acceleration unscented Kalman filter object from an initial detection report.

Create the detection report from an initial 3-D measurement, (-200,-30,5), of the object position. Assume uncorrelated measurement noise.

```
detection = objectDetection(0, [-200; -30; 5], 'MeasurementNoise', 2.0*eye(3), ...
    'SensorIndex', 1, 'ObjectClassID', 1, 'ObjectAttributes', {'Car', 2});
```

Create the new filter from the detection report and display the filter properties.

```
filter = initcaukf(detection)

filter =
    trackingUKF with properties:

        State: [9x1 double]
        StateCovariance: [9x9 double]

        StateTransitionFcn: @constacc
        ProcessNoise: [3x3 double]
        HasAdditiveProcessNoise: 0

        MeasurementFcn: @cameas
        HasMeasurementWrapping: 1
        MeasurementNoise: [3x3 double]
        HasAdditiveMeasurementNoise: 1

        Alpha: 1.0000e-03
        Beta: 2
        Kappa: 0
```

```
EnableSmoothing: 0
```

Show the state.

```
filter.State
```

```
ans = 9×1
```

```
-200
  0
  0
-30
  0
  0
  5
  0
  0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 9×9
```

```
 2.0000    0    0    0    0    0    0    0    0
  0 100.0000    0    0    0    0    0    0    0
  0    0 100.0000    0    0    0    0    0    0
  0    0    0  2.0000    0    0    0    0    0
  0    0    0    0 100.0000    0    0    0    0
  0    0    0    0    0 100.0000    0    0    0
  0    0    0    0    0    0  2.0000    0    0
  0    0    0    0    0    0    0 100.0000    0
  0    0    0    0    0    0    0    0 100.0000
```

### Create 3D Constant Acceleration UKF from Spherical Measurement

Initialize a 3D constant-acceleration unscented Kalman filter from an initial detection report made from a measurement in spherical coordinates. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to  $45^\circ$ , and the range to 1000 meters.

```
frame = 'spherical';
sensorpos = [25,-40,-10].';
sensorvel = [0;5;0];
laxes = eye(3);
```

Create the measurement structure. Set `'HasVelocity'` and `'HasElevation'` to false. Then, the measurement vector consists of azimuth angle and range.

```
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',false, ...
```

```
    'HasElevation',false);
meas = [45;1000];
measnoise = diag([3.0,2.0].^2);
detection = objectDetection(0,meas,'MeasurementNoise', ...
    measnoise,'MeasurementParameters',measparms)
```

```
detection =
  objectDetection with properties:

        Time: 0
    Measurement: [2x1 double]
  MeasurementNoise: [2x2 double]
        SensorIndex: 1
    ObjectClassID: 0
  MeasurementParameters: [1x1 struct]
    ObjectAttributes: {}
```

```
filter = initcaukf(detection);
```

Display the state vector.

```
disp(filter.State)
```

```
732.1068
      0
      0
667.1068
      0
      0
-10.0000
      0
      0
```

## Input Arguments

### **detection** – Detection report

objectDetection object

Detection report, specified as an objectDetection object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **filter** – Unscented Kalman filter

trackingUKF object

Unscented Kalman filter, returned as a trackingUKF object.

## Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration rate standard deviation of 1 m/s<sup>3</sup>.



- You can use this function as the `FilterInitializationFcn` property of a `radarTracker` object.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

`initcakf` | `initcaekf` | `initctekf` | `initctukf` | `initcvkf` | `initcvekf` | `initcvukf`

### Objects

`objectDetection` | `trackingKF` | `trackingEKF` | `trackingUKF` | `radarTracker`

### Introduced in R2021a

## initctekf

Create constant turn-rate extended Kalman filter from detection report

### Syntax

```
filter = initctekf(detection)
```

### Description

`filter = initctekf(detection)` creates and initializes a constant-turn-rate extended Kalman filter from information contained in a `detection` report. For more information about the extended Kalman filter, see `trackingEKF`.

The function initializes a constant turn-rate state with the same convention as `constturn` and `ctmeas`,  $[x; v_x; y; v_y; \omega; z; v_z]$ , where  $\omega$  is the turn-rate.

### Examples

#### Initialize 2-D Constant Turn-Rate Extended Kalman Filter

Create and initialize a 2-D constant turn-rate extended Kalman filter object from an initial detection report.

Create the detection report from an initial 2-D measurement, (-250,-40), of the object position. Assume uncorrelated measurement noise.

Extend the measurement to three dimensions by adding a z-component of zero.

```
detection = objectDetection(0, [-250; -40; 0], 'MeasurementNoise', 2.0*eye(3), ...
    'SensorIndex', 1, 'ObjectClassID', 1, 'ObjectAttributes', {'Car', 2});
```

Create the new filter from the detection report and display the filter properties.

```
filter = initctekf(detection)

filter =
    trackingEKF with properties:

        State: [7x1 double]
        StateCovariance: [7x7 double]

        StateTransitionFcn: @constturn
        StateTransitionJacobianFcn: @constturnjac
        ProcessNoise: [4x4 double]
        HasAdditiveProcessNoise: 0

        MeasurementFcn: @ctmeas
        MeasurementJacobianFcn: @ctmeasjac
        HasMeasurementWrapping: 1
        MeasurementNoise: [3x3 double]
        HasAdditiveMeasurementNoise: 1
```

```
MaxNum00SMSteps: 0
```

```
EnableSmoothing: 0
```

Show the state.

```
filter.State
```

```
ans = 7×1
```

```
-250
  0
 -40
  0
  0
  0
  0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 7×7
```

```
 2.0000    0    0    0    0    0    0
    0 100.0000    0    0    0    0    0
    0    0  2.0000    0    0    0    0
    0    0    0 100.0000    0    0    0
    0    0    0    0 100.0000    0    0
    0    0    0    0    0  2.0000    0
    0    0    0    0    0    0 100.0000
```

## Create 2-D Constant Turnrate EKF from Spherical Measurement

Initialize a 2-D constant-turnrate extended Kalman filter from an initial detection report made from an initial measurement in spherical coordinates. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to 45 degrees, the range to 1000 meters, and the range rate to -4.0 m/s.

```
frame = 'spherical';
sensorpos = [25, -40, -10].';
sensorvel = [0; 5; 0];
laxes = eye(3);
```

Create the measurement parameters structure. Set `'HasElevation'` to `false`. Then, the measurement consists of azimuth, range, and range rate.

```
measparms = struct('Frame', frame, 'OriginPosition', sensorpos, ...
    'OriginVelocity', sensorvel, 'Orientation', laxes, 'HasVelocity', true, ...
    'HasElevation', false);
```

```
meas = [45;1000;-4];  
measnoise = diag([3.0,2,1.0].^2);  
detection = objectDetection(0,meas,'MeasurementNoise', ...  
    measnoise,'MeasurementParameters',measparms)
```

```
detection =  
    objectDetection with properties:  
  
        Time: 0  
    Measurement: [3x1 double]  
    MeasurementNoise: [3x3 double]  
    SensorIndex: 1  
    ObjectClassID: 0  
    MeasurementParameters: [1x1 struct]  
    ObjectAttributes: {}
```

```
filter = initctekf(detection);
```

Filter state vector.

```
disp(filter.State)
```

```
732.1068  
-2.8284  
667.1068  
2.1716  
0  
-10.0000  
0
```

## Input Arguments

### **detection** – Detection report

objectDetection object

Detection report, specified as an objectDetection object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0;  
0 2.0 0; 0 0 1.5])`

## Output Arguments

### **filter** – Extended Kalman filter

trackingEKF object

Extended Kalman filter, returned as a trackingEKF object.

## Algorithms

- The function computes the process noise matrix assuming a one-second time step. The function assumes an acceleration standard deviation of 1 m/s<sup>2</sup>, and a turn-rate acceleration standard deviation of 1°/s<sup>2</sup>.
- You can use this function as the FilterInitializationFcn property of a radarTracker object.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

[initcaukf](#) | [initctukf](#) | [initcvkf](#) | [initcvekf](#) | [initcvukf](#) | [initcakf](#) | [initcaekf](#)

### Objects

[objectDetection](#) | [trackingKF](#) | [trackingEKF](#) | [trackingUKF](#) | [radarTracker](#)

**Introduced in R2021a**

## initctukf

Create constant turn-rate unscented Kalman filter from detection report

### Syntax

```
filter = initctukf(detection)
```

### Description

`filter = initctukf(detection)` creates and initializes a constant-turn-rate unscented Kalman filter from information contained in a `detection` report. For more information about the unscented Kalman filter, see `trackingUKF`.

The function initializes a constant turn-rate state with the same convention as `constturn` and `ctmeas`,  $[x; v_x; y; v_y; \omega; z; v_z]$ , where  $\omega$  is the turn-rate.

### Examples

#### Initialize 2-D Constant Turn-Rate Unscented Kalman Filter

Create and initialize a 2-D constant turn-rate unscented Kalman filter object from an initial detection report.

Create the detection report from an initial 2D measurement, (-250,-40), of the object position. Assume uncorrelated measurement noise.

Extend the measurement to three dimensions by adding a z-component of zero.

```
detection = objectDetection(0, [-250; -40; 0], 'MeasurementNoise', 2.0*eye(3), ...
    'SensorIndex', 1, 'ObjectClassID', 1, 'ObjectAttributes', {'Car', 2});
```

Create the new filter from the detection report and display the filter properties.

```
filter = initctukf(detection)

filter =
    trackingUKF with properties:

        State: [7x1 double]
        StateCovariance: [7x7 double]

        StateTransitionFcn: @constturn
        ProcessNoise: [4x4 double]
        HasAdditiveProcessNoise: 0

        MeasurementFcn: @ctmeas
        HasMeasurementWrapping: 1
        MeasurementNoise: [3x3 double]
        HasAdditiveMeasurementNoise: 1

        Alpha: 1.0000e-03
```

```
Beta: 2
Kappa: 0
```

```
EnableSmoothing: 0
```

Show the filter state.

```
filter.State
```

```
ans = 7×1
```

```
-250
  0
 -40
  0
  0
  0
  0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 7×7
```

```
 2.0000    0    0    0    0    0    0
  0 100.0000    0    0    0    0    0
  0    0  2.0000    0    0    0    0
  0    0    0 100.0000    0    0    0
  0    0    0    0 100.0000    0    0
  0    0    0    0    0  2.0000    0
  0    0    0    0    0    0 100.0000
```

## Create 2-D Constant Turn-rate UKF from Spherical Measurement

Initialize a 2-D constant turn-rate extended Kalman filter from an initial detection report made from an initial measurement in spherical coordinates. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to 45 degrees and the range to 1000 meters.

```
frame = 'spherical';
sensorpos = [25,-40,-10].';
sensorvel = [0;5;0];
laxes = eye(3);
```

Create the measurement parameters structure. Set `'HasVelocity'` and `'HasElevation'` to `false`. Then, the measurement consists of azimuth and range.

```
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',false, ...
    'HasElevation',false);
meas = [45;1000];
measnoise = diag([3.0,2].^2);
```

```
detection = objectDetection(0,meas,'MeasurementNoise', ...  
    measnoise,'MeasurementParameters',measparms)
```

```
detection =  
    objectDetection with properties:  
  
        Time: 0  
        Measurement: [2x1 double]  
        MeasurementNoise: [2x2 double]  
        SensorIndex: 1  
        ObjectClassID: 0  
        MeasurementParameters: [1x1 struct]  
        ObjectAttributes: {}
```

```
filter = initctukf(detection);
```

Filter state vector.

```
disp(filter.State)
```

```
732.1068  
    0  
667.1068  
    0  
    0  
-10.0000  
    0
```

## Input Arguments

### **detection** – Detection report

objectDetection object

Detection report, specified as an objectDetection object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0;  
0 2.0 0; 0 0 1.5])`

## Output Arguments

### **filter** – Unscented Kalman filter

trackingUKF object

Unscented Kalman filter, returned as a trackingUKF object.

## Algorithms

- The function computes the process noise matrix assuming a one-second time step. The function assumes an acceleration standard deviation of 1 m/s<sup>2</sup>, and a turn-rate acceleration standard deviation of 1°/s<sup>2</sup>.
- You can use this function as the FilterInitializationFcn property of a radarTracker object.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

[initcaukf](#) | [initcvkf](#) | [initcvekf](#) | [initcvukf](#) | [initcakf](#) | [initcaekf](#)

### Objects

[objectDetection](#) | [trackingKF](#) | [trackingEKF](#) | [trackingUKF](#) | [radarTracker](#)

**Introduced in R2021a**

## initcvekf

Create constant-velocity extended Kalman filter from detection report

### Syntax

```
filter = initcvekf(detection)
```

### Description

`filter = initcvekf(detection)` creates and initializes a constant-velocity extended Kalman filter from information contained in a `detection` report. For more information about the extended Kalman filter, see `trackingEKF`.

The function initializes a constant velocity state with the same convention as `constvel` and `cvmeas`,  $[x; v_x; y; v_y; z; v_z]$ .

### Examples

#### Initialize 3-D Constant-Velocity Extended Kalman Filter

Create and initialize a 3-D constant-velocity extended Kalman filter object from an initial detection report.

Create the detection report from an initial 3-D measurement, (10,20,-5), of the object position.

```
detection = objectDetection(0,[10;20;-5], 'MeasurementNoise',1.5*eye(3), ...
    'SensorIndex',1, 'ObjectClassID',1, 'ObjectAttributes',{'Sports Car',5});
```

Create the new filter from the detection report.

```
filter = initcvekf(detection)
```

```
filter =
    trackingEKF with properties:
        State: [6x1 double]
        StateCovariance: [6x6 double]
        StateTransitionFcn: @constvel
        StateTransitionJacobianFcn: @constveljac
        ProcessNoise: [3x3 double]
        HasAdditiveProcessNoise: 0
        MeasurementFcn: @cvmeas
        MeasurementJacobianFcn: @cvmeasjac
        HasMeasurementWrapping: 1
        MeasurementNoise: [3x3 double]
        HasAdditiveMeasurementNoise: 1
        MaxNumOOSMSteps: 0
```

```
EnableSmoothing: 0
```

Show the filter state.

```
filter.State
```

```
ans = 6×1
```

```
10
 0
20
 0
-5
 0
```

Show the state covariance.

```
filter.StateCovariance
```

```
ans = 6×6
```

```
1.5000    0    0    0    0    0
 0 100.0000    0    0    0    0
 0    0 1.5000    0    0    0
 0    0    0 100.0000    0    0
 0    0    0    0 1.5000    0
 0    0    0    0    0 100.0000
```

### Create 3-D Constant Velocity EKF from Spherical Measurement

Initialize a 3-D constant-velocity extended Kalman filter from an initial detection report made from a 3-D measurement in spherical coordinates. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to 45 degrees, the elevation to -10 degrees, the range to 1000 meters, and the range rate to -4.0 m/s.

```
frame = 'spherical';
sensorpos = [25,-40,0].';
sensorvel = [0;5;0];
laxes = eye(3);
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',true, ...
    'HasElevation',true);
meas = [45;-10;1000;-4];
measnoise = diag([3.0,2.5,2,1.0].^2);
detection = objectDetection(0,meas,'MeasurementNoise', ...
    measnoise,'MeasurementParameters',measparms)
```

```
detection =
    objectDetection with properties:
        Time: 0
        Measurement: [4x1 double]
```

```
MeasurementNoise: [4x4 double]
  SensorIndex: 1
  ObjectClassID: 0
MeasurementParameters: [1x1 struct]
  ObjectAttributes: {}
```

```
filter = initcvekf(detection);
```

Filter state vector.

```
disp(filter.State)
```

```
721.3642
-2.7855
656.3642
 2.2145
-173.6482
 0.6946
```

## Input Arguments

### **detection** — Detection report

objectDetection object

Detection report, specified as an objectDetection object.

```
Example: detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0;
0 2.0 0; 0 0 1.5])
```

## Output Arguments

### **filter** — Extended Kalman filter

trackingEKF object

Extended Kalman filter, returned as a trackingEKF object.

## Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration standard deviation of 1 m/s<sup>2</sup>.
- You can use this function as the FilterInitializationFcn property of a radarTracker object.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### **Functions**

initcaukf | initctekf | initctukf | initcvkf | initcvukf | initcakf | initcaekf

**Objects**

objectDetection | trackingKF | trackingEKF | trackingUKF | radarTracker

**Introduced in R2021a**

## initcvkf

Create constant-velocity linear Kalman filter from detection report

### Syntax

```
filter = initcvkf(detection)
```

### Description

`filter = initcvkf(detection)` creates and initializes a constant-velocity linear Kalman filter from information contained in a detection report. For more information about the linear Kalman filter, see `trackingKF`.

The function initializes a constant velocity state with the same convention as `constvel` and `cvmeas`,  $[x; v_x; y; v_y; z; v_z]$ .

### Examples

#### Initialize 2-D Constant-Velocity Linear Kalman Filter

Create and initialize a 2-D linear Kalman filter object from an initial detection report.

Create the detection report from an initial 2-D measurement, (10,20), of the object position.

```
detection = objectDetection(0,[10;20], 'MeasurementNoise',[1 0.2; 0.2 2], ...  
    'SensorIndex',1, 'ObjectClassID',1, 'ObjectAttributes',{'Yellow Car',5});
```

Create the new track from the detection report.

```
filter = initcvkf(detection)
```

```
filter =  
    trackingKF with properties:  
  
        State: [4x1 double]  
    StateCovariance: [4x4 double]  
  
        MotionModel: '2D Constant Velocity'  
    ProcessNoise: [2x2 double]  
  
    MeasurementModel: [2x4 double]  
    MeasurementNoise: [2x2 double]  
  
    MaxNumOOSMSteps: 0  
  
    EnableSmoothing: 0
```

Show the state.

```
filter.State
```

```
ans = 4×1
```

```
10
 0
20
 0
```

Show the state transition model.

```
filter.StateTransitionModel
```

```
ans = 4×4
```

```
 1    1    0    0
 0    1    0    0
 0    0    1    1
 0    0    0    1
```

### Initialize 3-D Constant-Velocity Linear Kalman Filter

Create and initialize a 3-D linear Kalman filter object from an initial detection report.

Create the detection report from an initial 3-D measurement, (10,20,-5), of the object position.

```
detection = objectDetection(0,[10;20;-5],'MeasurementNoise',eye(3), ...
    'SensorIndex', 1,'ObjectClassID',1,'ObjectAttributes',{'Green Car', 5});
```

Create the new filter from the detection report and display its properties.

```
filter = initcvkf(detection)
```

```
filter =
  trackingKF with properties:

    State: [6×1 double]
  StateCovariance: [6×6 double]

    MotionModel: '3D Constant Velocity'
  ProcessNoise: [3×3 double]

  MeasurementModel: [3×6 double]
  MeasurementNoise: [3×3 double]

  MaxNum00SMSteps: 0

  EnableSmoothing: 0
```

Show the state.

```
filter.State
```

```
ans = 6×1
```

```
10
0
20
0
-5
0
```

Show the state transition model.

```
filter.StateTransitionModel
```

```
ans = 6×6
```

```
1    1    0    0    0    0
0    1    0    0    0    0
0    0    1    1    0    0
0    0    0    1    0    0
0    0    0    0    1    1
0    0    0    0    0    1
```

## Input Arguments

### **detection** — Detection report

objectDetection object

Detection report, specified as an objectDetection object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise',[1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **filter** — Linear Kalman filter

trackingKF object

Linear Kalman filter, returned as a trackingKF object.

## Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration standard deviation of 1 m/s<sup>2</sup>.
- You can use this function as the FilterInitializationFcn property of a radarTracker object.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.



## See Also

### Functions

initcakf | initcaekf | initcaukf | initctekf | initctukf | initcvekf | initcvukf

### Objects

objectDetection | trackingKF | trackingEKF | trackingUKF | radarTracker

### Introduced in R2021a

## initcvukf

Create constant-velocity unscented Kalman filter from detection report

### Syntax

```
filter = initcvukf(detection)
```

### Description

`filter = initcvukf(detection)` creates and initializes a constant-velocity unscented Kalman filter from information contained in a `detection` report. For more information about the unscented Kalman filter, see `trackingUKF`.

The function initializes a constant velocity state with the same convention as `constvel` and `cvmeas`,  $[x; v_x; y; v_y; z; v_z]$ .

### Examples

#### Initialize 3-D Constant-Velocity Unscented Kalman Filter

Create and initialize a 3-D constant-velocity unscented Kalman filter object from an initial detection report.

Create the detection report from an initial 3-D measurement, (10,200,-5), of the object position.

```
detection = objectDetection(0,[10;200;-5], 'MeasurementNoise',1.5*eye(3), ...
    'SensorIndex',1, 'ObjectClassID',1, 'ObjectAttributes',{'Sports Car',5});
```

Create the new filter from the detection report and display the filter properties.

```
filter = initcvukf(detection)

filter =
    trackingUKF with properties:

        State: [6x1 double]
        StateCovariance: [6x6 double]

        StateTransitionFcn: @constvel
        ProcessNoise: [3x3 double]
        HasAdditiveProcessNoise: 0

        MeasurementFcn: @cvmeas
        HasMeasurementWrapping: 1
        MeasurementNoise: [3x3 double]
        HasAdditiveMeasurementNoise: 1

        Alpha: 1.0000e-03
        Beta: 2
        Kappa: 0
```

```
EnableSmoothing: 0
```

Display the state.

```
filter.State
```

```
ans = 6×1
```

```
10
 0
200
 0
 -5
 0
```

Show the state covariance.

```
filter.StateCovariance
```

```
ans = 6×6
```

```
1.5000    0    0    0    0    0
 0 100.0000    0    0    0    0
 0    0 1.5000    0    0    0
 0    0    0 100.0000    0    0
 0    0    0    0 1.5000    0
 0    0    0    0    0 100.0000
```

### Create Constant Velocity UKF from Spherical Measurement

Initialize a constant-velocity unscented Kalman filter from an initial detection report made from an initial measurement in spherical coordinates. Because the object lies in the x-y plane, no elevation measurement is made. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to 45 degrees, the range to 1000 meters, and the range rate to -4.0 m/s.

```
frame = 'spherical';
sensorpos = [25,-40,0].';
sensorvel = [0;5;0];
laxes = eye(3);
```

Create the measurement parameters structure. Set `'HasElevation'` to `false`. Then, the measurement consists of azimuth, range, and range rate.

```
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',true, ...
    'HasElevation',false);
meas = [45;1000;-4];
measnoise = diag([3.0,2,1.0].^2);
detection = objectDetection(0,meas,'MeasurementNoise', ...
    measnoise,'MeasurementParameters',measparms)
```

```
detection =
    objectDetection with properties:
```

```
        Time: 0
        Measurement: [3x1 double]
MeasurementNoise: [3x3 double]
        SensorIndex: 1
        ObjectClassID: 0
MeasurementParameters: [1x1 struct]
        ObjectAttributes: {}
```

```
filter = initcvukf(detection);
```

Display filter state vector.

```
disp(filter.State)
```

```
732.1068
-2.8284
667.1068
 2.1716
     0
     0
```

## Input Arguments

### **detection** – Detection report

`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **filter** – Unscented Kalman filter

`trackingUKF` object

Unscented Kalman filter, returned as a `trackingUKF` object.

## Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration standard deviation of 1 m/s<sup>2</sup>.
- You can use this function as the `FilterInitializationFcn` property of a `radarTracker` object.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Functions

initcakf | initcaekf | initcaukf | initctekf | initctukf | initcvkf | initcvekf

### Objects

objectDetection | trackingKF | trackingEKF | trackingUKF | radarTracker

### Introduced in R2021a

## cranerainpl

RF signal attenuation due to rainfall using Crane model

### Syntax

```
L = cranerainpl(range, freq, rainrate)
L = cranerainpl(range, freq, rainrate, elev)
L = cranerainpl(range, freq, rainrate, elev, tau)
```

### Description

`L = cranerainpl(range, freq, rainrate)` returns the signal attenuation, `L`, due to rain based on the Crane rain model [1]. Signal attenuation is a function of the signal path length, `range`, the signal frequency, `freq`, and the rain rate, `rainrate`. The rain rate is defined as the long-term statistical rain rate. The attenuation model applies only for frequencies from 1 GHz to 1000 GHz and is valid for ranges up to 22.5 km. The Crane model accounts for the cellular nature of rainstorms.

`L = cranerainpl(range, freq, rainrate, elev)` also specifies the elevation angle, `elev`, of the signal path.

`L = cranerainpl(range, freq, rainrate, elev, tau)` also specifies the polarization tilt angle, `tau`, of the signal.

### Examples

#### Compare Attenuation for Two Rain Rates Using Crane Model

Use the Crane rain model to compute the signal attenuation caused by rain for a 20 GHz signal sent over a distance of 10 km. Use rain rates of 10.0 and 100.0 mm/hr.

First, set the rain rate to 10 mm/hr.

```
rr = 10.0;
L = cranerainpl(10e3, 20.0e9, rr)
```

```
L = 12.5988
```

Repeat the computation using a rain rate of 100.0 mm/hr.

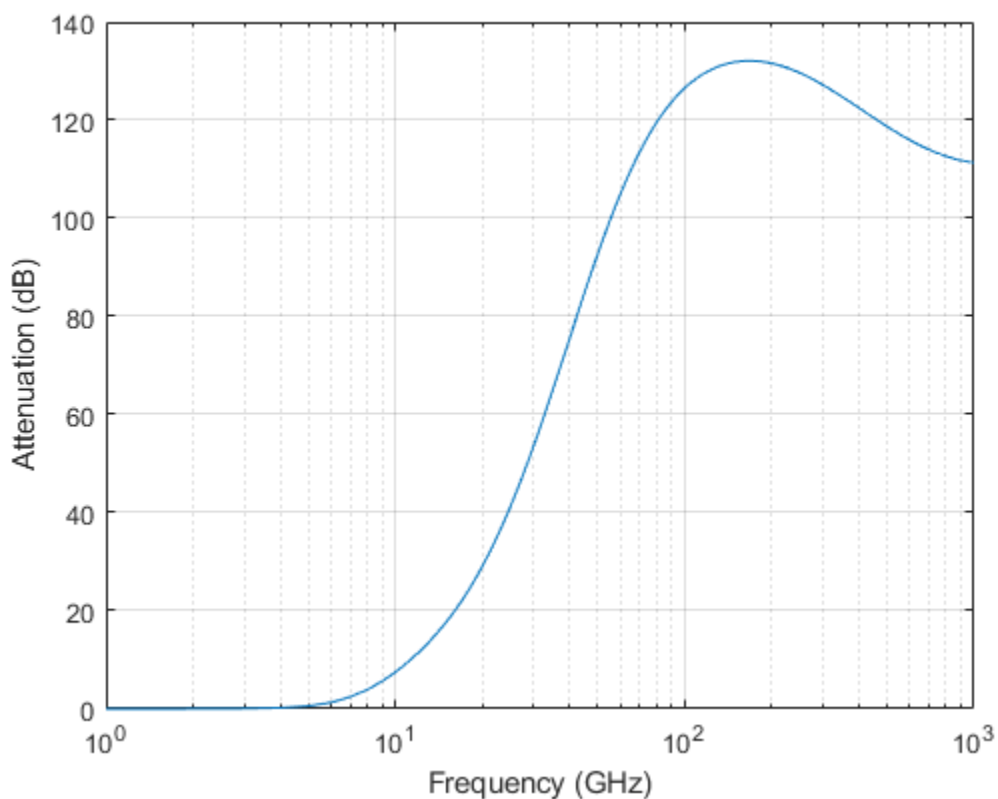
```
rr = 100.0;
L = cranerainpl(10e3, 20.0e9, rr)
```

```
L = 73.1912
```

### Rain Attenuation as a Function of Frequency Using Crane Model

Plot the signal attenuation due to rain for signals in the frequency range from 1 to 1000 GHz. Use the Crane model to compute the attenuation for a rain rate of 30.0 mm/hr and a signal path distance of 10 km.

```
rr = 30.0;
freq = [1:1000]*1e9;
L = cranerainpl(10e3, freq, rr);
semilogx(freq/1e9, L)
grid
xlabel('Frequency (GHz)')
ylabel('Attenuation (dB)')
```



### Rain Attenuation as a Function of Elevation Using Crane Model

Plot the signal attenuation due to rain as a function of elevation angle. Elevation angles vary from 0 to 90 degrees. Assume a path distance of 10 km and a signal frequency of 10 GHz. The rain rate is 100 mm/hr.

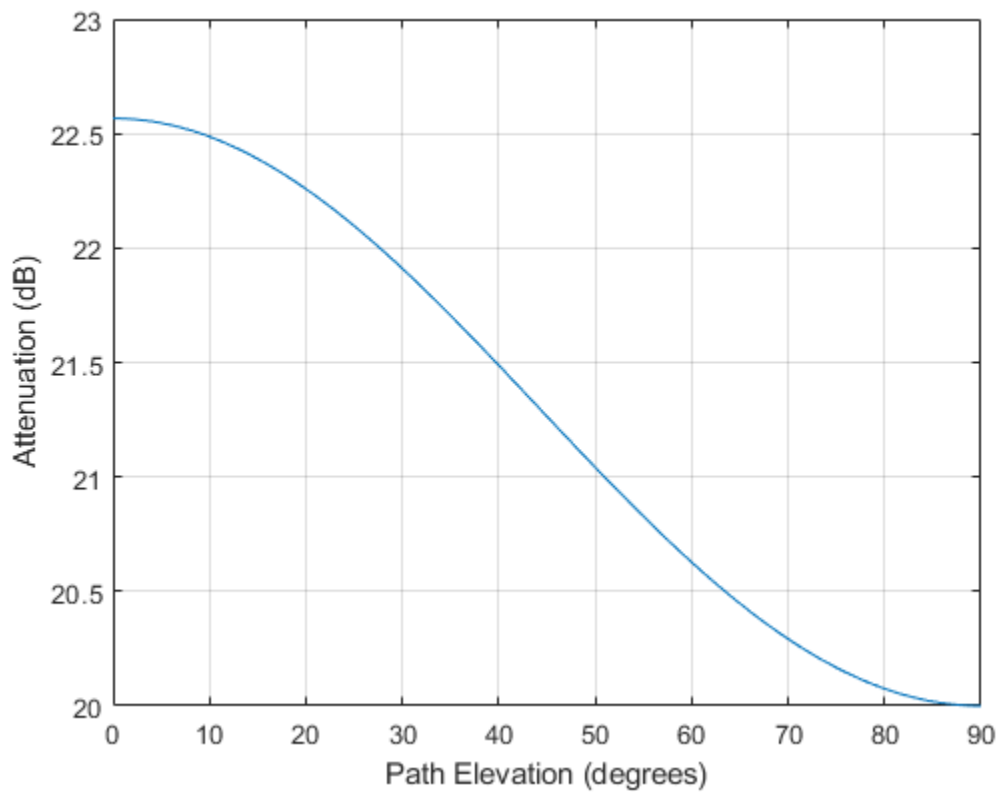
```
rr = 100.0;
```

Set the elevation angles, frequency, and path length.

```
elev = [0:1:90];
freq = 10.0e9;
rng = 10e3*ones(size(elev));
```

Compute and plot the loss.

```
L = cranerainpl(rng,freq,rr,elev);
plot(elev,L)
grid
xlabel('Path Elevation (degrees)')
ylabel('Attenuation (dB)')
```



### Rain Attenuation as a Function of Polarization Using Crane Model

Plot the signal attenuation due to rainfall as a function of the polarization tilt angle. Assume a path distance of 10 km, a signal frequency of 10 GHz, and a path elevation angle of 0 degrees. Set the rainfall rate to 70 mm/hour. Plot the signal attenuation against polarization tilt angle.

Set the polarization tilt angle to vary from -90 to 90 degrees.

```
tau = -90:90;
```

Set the elevation angle, frequency, path distance, and rain rate.

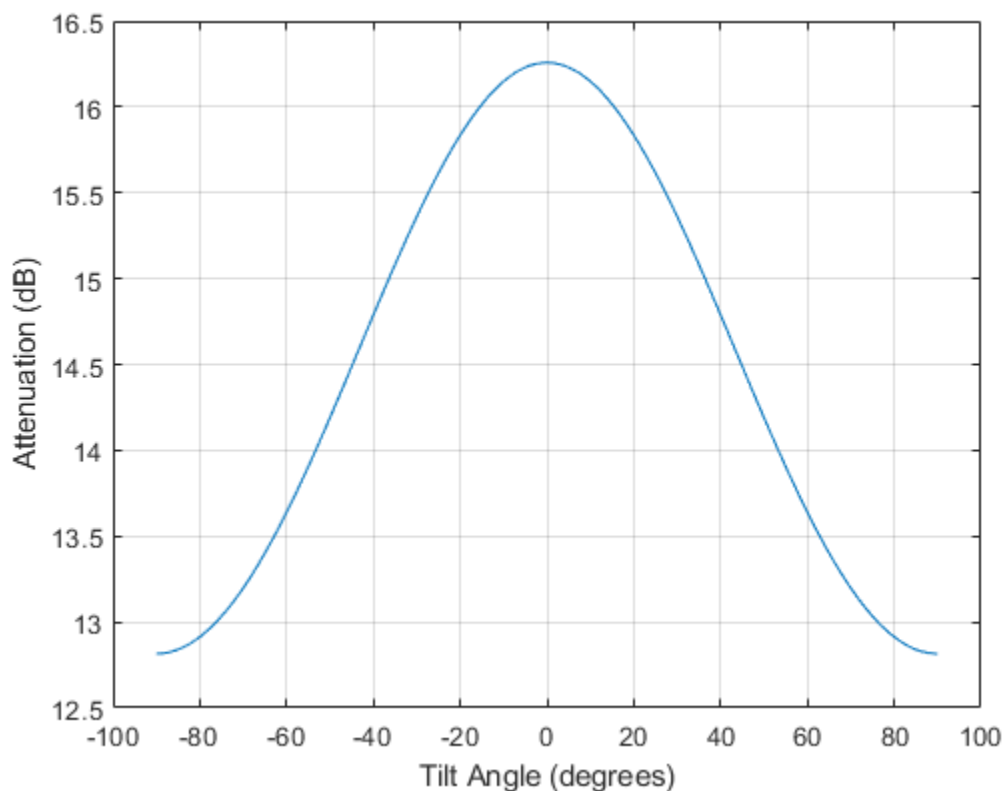
```
elev = 0;
freq = 10.0e9;
```



```
rng = 10e3*ones(size(tau));
rr = 70.0;
```

Compute and plot the attenuation.

```
L = cranerainpl(rng,freq,rr,elev,tau);
plot(tau,L)
grid
xlabel('Tilt Angle (degrees)')
ylabel('Attenuation (dB)')
```



## Input Arguments

### range — Signal path length

positive scalar | real-valued 1-by- $M$  vector of positive values | real-valued  $M$ -by-1 vector of positive values

Signal path length, specified as a positive scalar, a real-valued 1-by- $M$  vector of positive values, or real-valued  $M$ -by-1 vector of positive values. Units are in meters.

Example: [13000.0,14000.0]

### freq — Signal frequency

positive scalar | real-valued 1-by- $N$  vector of positive values | real-valued  $N$ -by-1 vector of positive values

Signal frequency, specified as a positive scalar, a real-valued 1-by- $N$  vector of positive values, or a real-valued  $N$ -by-1 vector of positive values. Units are in Hz. Frequencies must lie in the range 1-1000 GHz.

Example: `[2.0:2:10.0]*1e9]`

**rainrate – Rain rate**

nonnegative scalar

Rain rate, specified as a nonnegative scalar. Rain rate represents the long-term statistical rainfall rate provided by Crane (see [1]). Units are in mm/hr.

Example: `100.5`

**elev – Signal path elevation angle**

0.0 (default) | scalar | real-valued 1-by- $M$  vector | real-valued  $M$ -by-1 vector

Signal path elevation angle, specified as a real-valued scalar, or real-valued  $M$ -by-1 or real-valued 1-by- $M$  vector. Units are in degrees between  $-90^\circ$  and  $90^\circ$ .

- If `elev` is a scalar, all propagation paths have the same elevation angle.
- If `elev` is a vector, its length must match the length of `range` and each element in `elev` corresponds to a propagation range.

Example: `[0,45]`

**tau – Tilt angle of signal polarization ellipse**

0.0 (default) | scalar | real-valued 1-by- $M$  vector | real-valued  $M$ -by-1 vector

Tilt angle of the signal polarization ellipse, specified as a scalar, a real-valued 1-by- $M$  vector, or a real-valued  $M$ -by-1 vector. Tilt angle values are in the range  $-90^\circ$  and  $90^\circ$ , inclusive. Units are in degrees.

- If `tau` is a scalar, all signals have the same tilt angle.
- If `tau` is a vector, its length must match the length of `range`. In that case, each element in `tau` corresponds to a propagation path in `range`.

The tilt angle is defined as the angle between the semimajor axis of the polarization ellipse and the  $x$ -axis. Because the ellipse is symmetrical, a tilt angle of  $10^\circ$  corresponds to the same polarization state as a tilt angle of  $-80^\circ$ . Thus, the tilt angle need only be specified between  $\pm 90^\circ$ .

Example: `[45,30]`

## Output Arguments

**L – Signal attenuation**

real-valued  $M$ -by- $N$  matrix

Signal attenuation, returned as a real-valued  $M$ -by- $N$  matrix. Each matrix row represents a different path where  $M$  is the number of paths. Each column represents a different frequency where  $N$  is the number of frequencies. Units are in dB.

## More About

### Crane Rainfall Attenuation Model

The Crane model calculates the attenuation of signals that propagate through regions of rainfall. The model was developed for use on Earth-space or terrestrial propagation paths and is a commonly-used method for the calculation of rain attenuation. The model is based on observations of rain rate, rain structure, and the vertical variation of temperature in the atmosphere. The Crane model (see *Electromagnetic Wave Propagation through Rain*) is primarily applicable to North America. The Crane model generally predicts losses greater than those of the ITU rain attenuation model used in the function. However, the uncertainty of both models and the short-term variation of fade can be large.

The ITU and Crane models are very similar but have some differences. The ITU and Crane rain attenuation models both require statistical annual rainfall rates and utilize an effective path length reduction factor to account for the cellular nature of storms. The 0.01% rainfall rate tables provided by Crane and the ITU are different. The Crane rainfall zones are similar to the ITU zones but more zones are defined in the US than in the ITU model. The ITU rainfall zones are discussed in *ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. The Crane model is more complex consisting of a piecewise combination of path profiles composed of exponential functions.

The Crane model utilizes two exponential functions to span the distance from 0 to 22.5 km.

- For  $\delta < D < 22.5$ ,

$$L = \gamma \left( \frac{e^{\gamma\delta} - 1}{\gamma} - \frac{b^\alpha e^{z\delta}}{z} + \frac{b^\alpha e^{zD}}{z} \right)$$

- For  $0 < D < \delta$ ,

$$L = \gamma \left( \frac{e^{\gamma D} - 1}{\gamma} \right)$$

where

- $L$  = path attenuation (dB)
- $D$  = propagation distance (km)
- $R$  = statistical 0.01% rain rate (mm/hr)
- $\gamma$  = specific attenuation identical to that calculated in `rainpl`.

$$\gamma_R = kR^\alpha,$$

The parameters  $k$  and  $\alpha$  depend on the frequency, the polarization state, and the elevation angle of the signal path. These coefficients, given by both Crane *Electromagnetic Wave Propagation through Rain* and the *ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*, are identical and are valid from 1 GHz to 1000 GHz. The specific attenuation model is valid for frequencies from 1-1000 GHz. Rainfall specific attenuation is computed according to the ITU rainfall model in *ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*.

The remaining parameters are empirical constants defined as:

- $b = 2.3R^{-0.17}$

- $c = 0.026 - 0.03 \ln R$
- $\delta = 3.8 - 0.6 \ln R$
- $u = \ln (be^{c\delta})/\delta$
- $y = \alpha u$
- $z = \alpha c$

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the propagation distance.

You can also apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands and apply attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

## References

- [1] Crane, Robert K. *Electromagnetic Wave Propagation through Rain*. Wiley, 1996.
- [2] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. P Series, Radiowave Propagation 2005.
- [3] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.530-17: Propagation data and prediction methods required for the design of terrestrial line-of-sight systems*. 2017.
- [4] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.837-7: Characteristics of precipitation for propagation modelling*. 6/2017

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

## See Also

**Introduced in R2020a**

# rainpl

RF signal attenuation due to rainfall

## Syntax

```
L = rainpl(range, freq, rainrate)
L = rainpl(range, freq, rainrate, elev)
L = rainpl(range, freq, rainrate, elev, tau)
L = rainpl(range, freq, rainrate, elev, tau, pct)
```

## Description

`L = rainpl(range, freq, rainrate)` returns the signal attenuation, `L`, due to rainfall. In this syntax, attenuation is a function of signal path length, `range`, signal frequency, `freq`, and rain rate, `rainrate`. The path elevation angle and polarization tilt angles are assumed to zero.

The `rainpl` function applies the International Telecommunication Union (ITU) rainfall attenuation model to calculate path loss of signals propagating in a region of rainfall [1]. The function applies when the signal path is contained entirely in a uniform rainfall environment. Rain rate does not vary along the signal path. The attenuation model applies only for frequencies at 1-1000 GHz.

`L = rainpl(range, freq, rainrate, elev)` also specifies the elevation angle, `elev`, of the propagation path.

`L = rainpl(range, freq, rainrate, elev, tau)` also specifies the polarization tilt angle, `tau`, of the signal.

`L = rainpl(range, freq, rainrate, elev, tau, pct)` also specifies the specified percentage of time, `pct`. `pct` is a scalar in the range of 0.001-1, inclusive. The attenuation, `L`, is computed from a power law using the long-term statistical 0.01% rain rate (in mm/h).

## Examples

### Signal Attenuation Due to Rainfall

Compute the signal attenuation due to rainfall for a 20 GHz signal over a distance of 10 km in light and heavy rain.

Propagate the signal in a light rainfall of 1 mm/hr.

```
rr = 1.0;
L = rainpl(10000, 20.0e9, rr)
```

```
L = 1.3009
```

Propagate the signal in a heavy rainfall of 10 mm/hr.

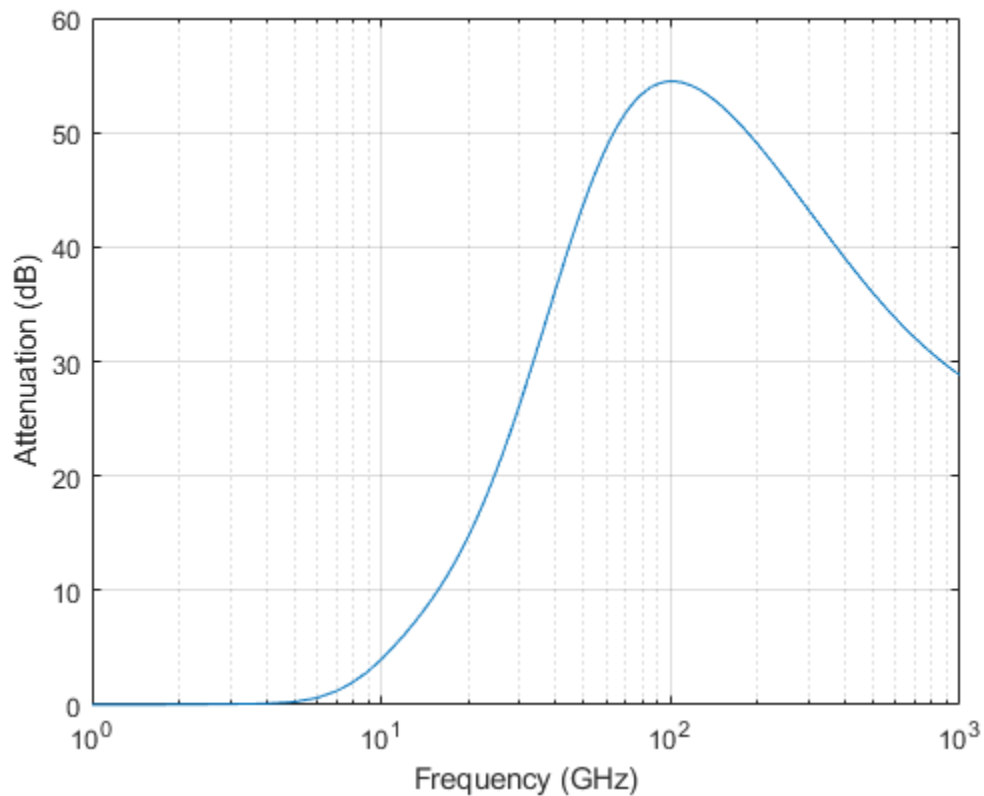
```
rr = 10.0;
L = rainpl(10000, 20.0e9, rr)
```

L = 8.1584

### Signal Attenuation Due to Rainfall as Function of Frequency

Plot the signal attenuation due to a 20 mm/hr statistical rainfall for signals in the frequency range from 1 to 1000 GHz. The path distance is 10 km.

```
rr = 20.0;  
freq = [1:1000]*1e9;  
L = rainpl(10000,freq,rr);  
semilogx(freq/1e9,L)  
grid  
xlabel('Frequency (GHz)')  
ylabel('Attenuation (dB)')
```



### Signal Attenuation Due to Rainfall as Function of Elevation Angle

Compute the signal attenuation due to heavy rain as a function of elevation angle. Elevation angles vary from 0 to 90 degrees. Assume a path distance of 100 km and a signal frequency of 100 GHz.

Set the rain rate to 10 mm/hr.

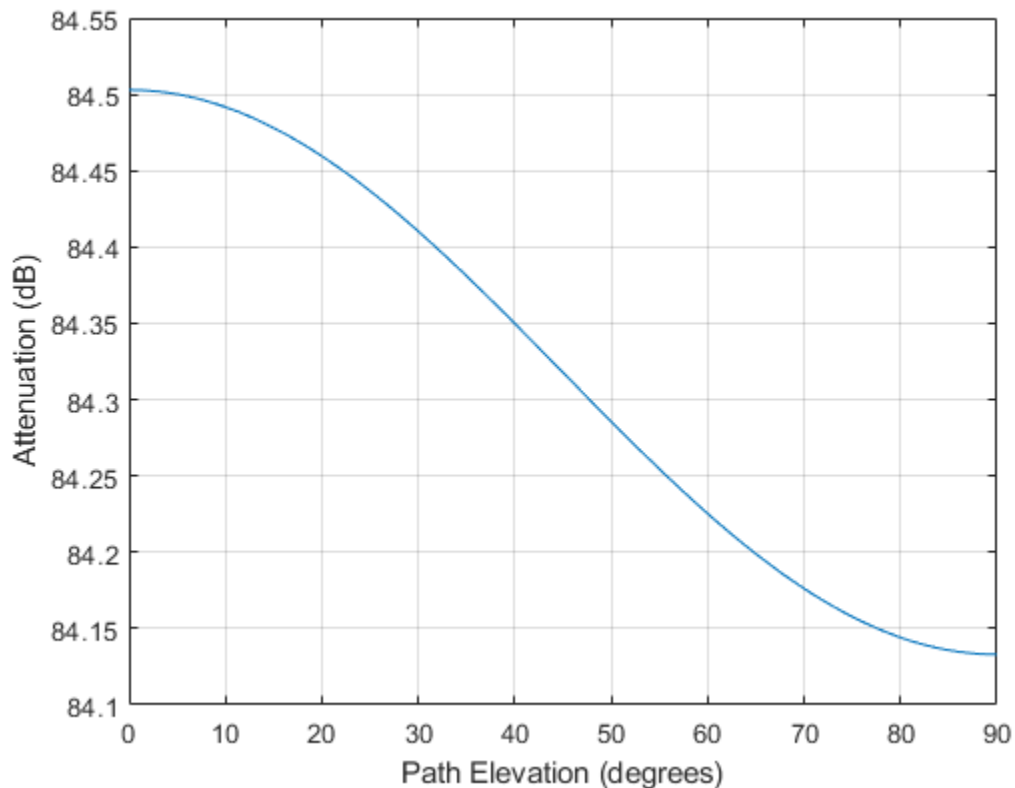
```
rr = 10.0;
```

Set the elevation angles, frequency, range.

```
elev = [0:1:90];
freq = 100.0e9;
rng = 100000.0*ones(size(elev));
```

Compute and plot the loss.

```
L = rainpl(rng,freq,rr,elev);
plot(elev,L)
grid
xlabel('Path Elevation (degrees)')
ylabel('Attenuation (dB)')
```



### Signal Attenuation Due to Rainfall as Function of Polarization

Compute the signal attenuation due to heavy rainfall as a function of the polarization tilt angle. Assume a path distance of 100 km, a signal frequency of 100 GHz, and a path elevation angle of 0 degrees. Set the rainfall rate to 10 mm/hour. Plot the signal attenuation versus polarization tilt angle.

Set the polarization tilt angle to vary from -90 to 90 degrees.

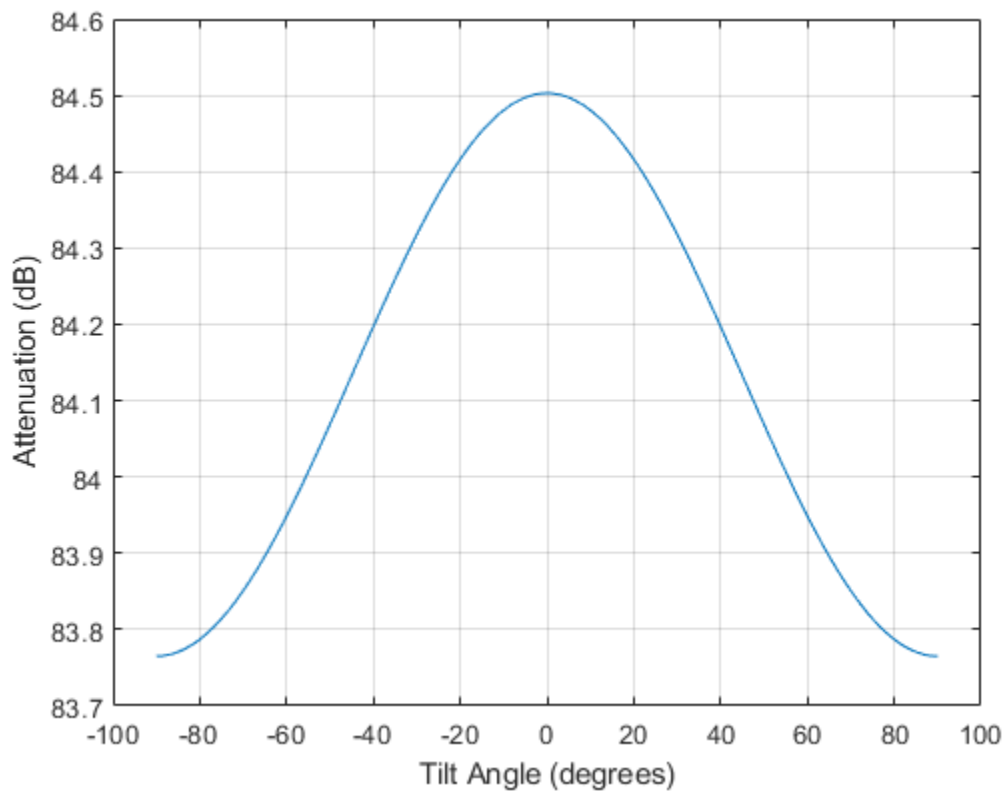
```
tau = -90:90;
```

Set the elevation angle, frequency, path distance, and rain rate.

```
elev = 0;  
freq = 100.0e9;  
rng = 100e3*ones(size(tau));  
rr = 10.0;
```

Compute and plot the attenuation.

```
L = rainpl(rng,freq,rr,elev,tau);  
plot(tau,L)  
grid  
xlabel('Tilt Angle (degrees)')  
ylabel('Attenuation (dB)')
```



## Input Arguments

### range — Signal path length

nonnegative real-valued scalar | nonnegative real-valued  $M$ -by-1 column vector | nonnegative real-valued 1-by- $M$  row vector

Signal path length, specified as a nonnegative real-valued scalar, or as a  $M$ -by-1 or 1-by- $M$  vector. Units are in meters.

Example: [13000.0,14000.0]



**freq — Signal frequency**

positive real-valued scalar | nonnegative real-valued  $N$ -by-1 column vector | nonnegative real-valued 1-by- $N$  row vector

Signal frequency, specified as a positive real-valued scalar, or as a nonnegative  $N$ -by-1 or 1-by- $N$  vector. Frequencies must lie in the range 1–1000 GHz.

Example: [1400.0e6, 2.0e9]

**rainrate — Long-term statistical rain rate**

nonnegative real-valued scalar

Long-term statistical rain rate, specified as a nonnegative real-valued scalar. The long-term statistical rain rate is the rain rate that is exceeded 0.01% of the time. You can adjust the percent of time using the `pct` argument. Units are in mm/hr.

Example: 1.5

**elev — Signal path elevation angle**

0.0 (default) | real-valued scalar | real-valued  $M$ -by-1 column vector | real-valued 1-by- $M$  row vector

Signal path elevation angle, specified as a real-valued scalar, or as an  $M$ -by-1 or 1-by- $M$  vector. Units are in degrees between  $-90^\circ$  and  $90^\circ$ . If `elev` is a scalar, all propagation paths have the same elevation angle. If `elev` is a vector, its length must match the dimension of `range` and each element in `elev` corresponds to a propagation range in `range`.

Example: [0, 45]

**tau — Tilt angle of polarization ellipse**

0.0 (default) | real-valued scalar | real-valued  $M$ -by-1 column vector | real-valued 1-by- $M$  row vector

Tilt angle of the signal polarization ellipse, specified as a real-valued scalar, or as an  $M$ -by-1 or 1-by- $M$  vector. Units are in degrees between  $-90^\circ$  and  $90^\circ$ . If `tau` is a scalar, all signals have the same tilt angle. If `tau` is a vector, its length must match the dimension of `range`. In that case, each element in `tau` corresponds to a propagation path in `range`.

The tilt angle is defined as the angle between the semi-major axis of the polarization ellipse and the  $x$ -axis. Because the ellipse is symmetrical, a tilt angle of  $100^\circ$  corresponds to the same polarization state as a tilt angle of  $-80^\circ$ . Thus, the tilt angle need only be specified between  $\pm 90^\circ$ .

Example: [45, 30]

**pct — Exceedance percentage of rainfall**

0.01 (default) | positive scalar between 0.001 and 1

Exceedance percentage of rainfall, specified as a positive scalar between 0.001 and 1. The long-term statistical rain rate is the rain rate that is exceeded `pct` of the time. Units are dimensionless.

Data Types: double

**Output Arguments****L — Signal attenuation**

real-valued  $M$ -by- $N$  matrix

Signal attenuation, returned as a real-valued  $M$ -by- $N$  matrix. Each matrix row represents a different path where  $M$  is the number of paths. Each column represents a different frequency where  $N$  is the number of frequencies. Units are in dB.

## More About

### Rainfall Attenuation Model

This model calculates the attenuation of signals that propagate through regions of rainfall. Rain attenuation is a dominant fading mechanism and can vary from location-to-location and from year-to-year.

Electromagnetic signals are attenuated when propagating through a region of rainfall. Rainfall attenuation is computed according to the ITU rainfall model *Recommendation ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. The model computes the specific attenuation (attenuation per kilometer) of a signal as a function of rainfall rate, signal frequency, polarization, and path elevation angle. The specific attenuation,  $\gamma_R$ , is modeled as a power law with respect to rain rate

$$\gamma_R = kR^\alpha,$$

where  $R$  is rain rate. Units are in mm/hr. The parameter  $k$  and exponent  $\alpha$  depend on the frequency, the polarization state, and the elevation angle of the signal path. The specific attenuation model is valid for frequencies from 1-1000 GHz.

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the an effective propagation distance,  $d_{\text{eff}}$ . Then, the total attenuation is  $L = d_{\text{eff}}\gamma_R$ .

The effective distance is the geometric distance,  $d$ , multiplied by a scale factor

$$r = \frac{1}{0.477d^{0.633}R_{0.01}^{0.073\alpha}f^{0.123} - 10.579(1 - \exp(-0.024d))}$$

where  $f$  is the frequency. The article *Recommendation ITU-R P.530-17 (12/2017): Propagation data and prediction methods required for the design of terrestrial line-of-sight systems* presents a complete discussion for computing attenuation.

The rain rate,  $R$ , used in these computations is the long-term statistical rain rate,  $R_{0.01}$ . This is the rain rate that is exceeded 0.01% of the time. The calculation of the statistical rain rate is discussed in *Recommendation ITU-R P.837-7 (06/2017): Characteristics of precipitation for propagation modelling*. This article also explains how to compute the attenuation for other percentages from the 0.01% value.

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands and apply attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

## References

- [1] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. 2005.

- [2] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.530-17: Propagation data and prediction methods required for the design of terrestrial line-of-sight systems*. 2017.
- [3] *Recommendation ITU-R P.837-7: Characteristics of precipitation for propagation modelling*
- [4] Seybold, J. *Introduction to RF Propagation*. New York: Wiley & Sons, 2005.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

## **See Also**

## fogpl

RF signal attenuation due to fog and clouds

### Syntax

```
L = fogpl(R, freq, T, den)
```

### Description

`L = fogpl(R, freq, T, den)` returns attenuation, `L`, when signals propagate in fog or clouds. `R` represents the signal path length. `freq` represents the signal carrier frequency, `T` is the ambient temperature, and `den` specifies the liquid water density in the fog or cloud.

The `fogpl` function applies the International Telecommunication Union (ITU) cloud and fog attenuation model to calculate path loss of signals propagating through clouds and fog. See [1]. Fog and clouds are the same atmospheric phenomenon, differing only by height above ground. Both environments are parametrized by their liquid water density. Other model parameters include signal frequency and temperature. This function applies to cases when the signal path is contained entirely in a uniform fog or cloud environment. The liquid water density does not vary along the signal path. The attenuation model applies only for frequencies at 10–1000 GHz.

### Examples

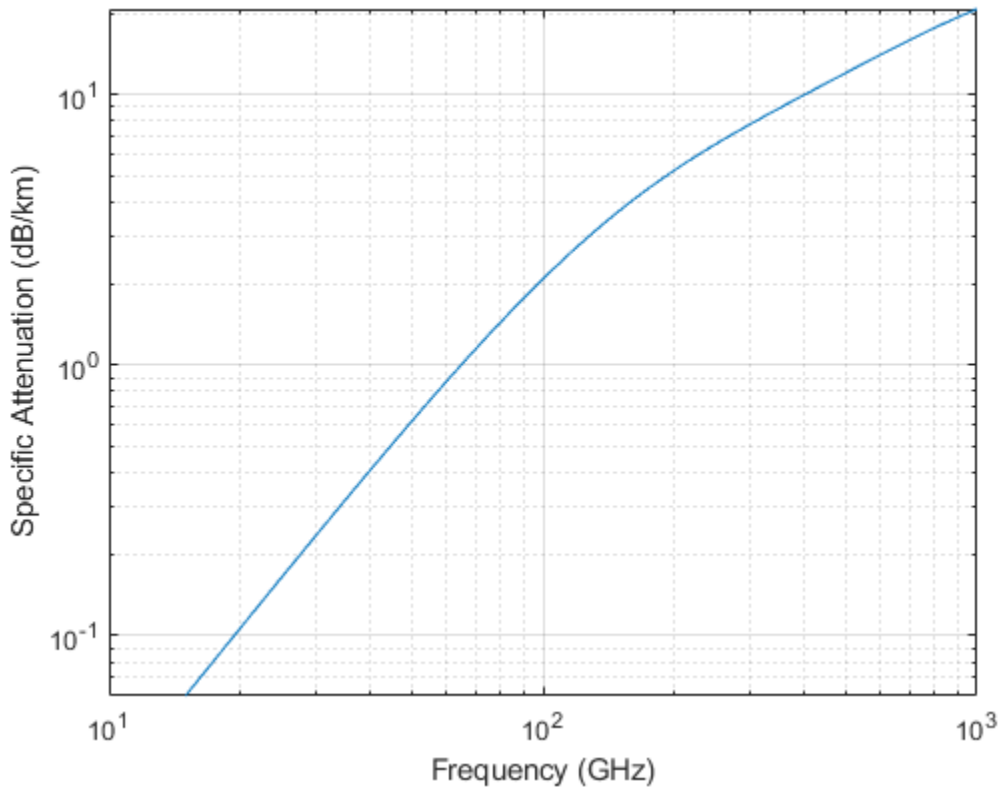
#### Attenuation in Cumulus Clouds

Compute the attenuation of signals propagating through a cloud that is 1 km long at 1000 meters altitude. Compute the attenuation for frequencies from 15 to 1000 GHz. A typical value for the cloud liquid water density is  $0.5 \text{ g/m}^3$ . Assume the atmospheric temperature at 1000 meters is  $20^\circ\text{C}$ .

```
R = 1000.0;  
freq = [15:5:1000]*1e9;  
T = 20.0;  
lwd = 0.5;  
L = fogpl(R, freq, T, lwd);
```

Plot the specific attenuation as a function of frequency. Specific attenuation is the attenuation or loss per kilometer.

```
loglog(freq/1e9, L)  
grid  
xlabel('Frequency (GHz)')  
ylabel('Specific Attenuation (dB/km)')
```



## Input Arguments

### **R** – Signal path length

positive real-valued scalar |  $M$ -by-1 nonnegative real-valued vector | 1-by- $M$  nonnegative real-valued vector

Signal path length, specified as a scalar or as an  $M$ -by-1 or 1-by- $M$  vector of nonnegative real-values. Total attenuation is the specific attenuation multiplied by the path length. Units are meters.

Example: [1300.0, 1400.0]

### **freq** – Signal frequency

positive real-valued scalar |  $N$ -by-1 nonnegative real-valued column vector | 1-by- $N$  nonnegative real-valued row vector

Signal frequency, specified as a positive real-valued scalar or as an  $N$ -by-1 nonnegative real-valued vector or 1-by- $N$  nonnegative real-valued vector. Frequencies must lie in the range 10-1000 GHz. Units are in Hz.

Example: [14.0e9, 15.0e9]

### **T** – Ambient temperature

real-valued scalar

Ambient temperature in fog or cloud, specified as a real-valued scalar. Units are in degrees Celsius.

Example: -10.0

**den — Liquid water density**

nonnegative real-valued scalar

Liquid water density, specified as a nonnegative real-valued scalar. Units are g/m<sup>3</sup>. Typical values for liquid water density in fog range from approximately 0.05 g/m<sup>3</sup> for medium fog to approximately 0.5 g/m<sup>3</sup> for thick fog. For medium fog, visibility is about 300 meters. For heavy fog, visibility is about 50 meters. Cumulus cloud liquid water density is typically 0.5 g/m<sup>3</sup>.

Example: 0.01

**Output Arguments****L — Signal attenuation**

real-valued  $M$ -by- $N$  matrix

Signal attenuation, returned as a real-valued  $M$ -by- $N$  matrix. Each matrix row represents a different path where  $M$  is the number of paths. Each column represents a different frequency where  $N$  is the number of frequencies. Units are in dB.

**More About****Fog and Cloud Attenuation Model**

This model calculates the attenuation of signals that propagate through fog or clouds.

Fog and cloud attenuation are the same atmospheric phenomenon. The ITU model, *Recommendation ITU-R P.840-6: Attenuation due to clouds and fog* is used. The model computes the specific attenuation (attenuation per kilometer), of a signal as a function of liquid water density, signal frequency, and temperature. The model applies to polarized and nonpolarized fields. The formula for specific attenuation at each frequency is

$$\gamma_c = K_l(f)M,$$

where  $M$  is the liquid water density in gm/m<sup>3</sup>. The quantity  $K_l(f)$  is the specific attenuation coefficient and depends on frequency. The cloud and fog attenuation model is valid for frequencies 10-1000 GHz. Units for the specific attenuation coefficient are (dB/km)/(g/m<sup>3</sup>).

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the path length  $R$ . Total attenuation is  $L_c = R\gamma_c$ .

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands, and apply narrowband attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

**References**

- [1] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.840-6: Attenuation due to clouds and fog*. 2013.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

### **See Also**





# Blocks

---

## DBSCAN Clusterer

Cluster detections

**Library:** Radar Toolbox



### Description

Cluster data using the density-based spatial clustering of applications with noise (DBSCAN) algorithm. The DBSCAN Clusterer block can cluster any type of data. The block can also solve for the clustering threshold (epsilon) and can perform data disambiguation in two dimensions.

### Ports

#### Input

##### X — Input data

*N*-by-*P* real-valued matrix

Input data, specified as a real-valued *N*-by-*P* matrix, where *N* is the number of data points to cluster. *P* is the number of feature dimensions. The DBSCAN algorithm can cluster any type of data with appropriate **Minimum number of points in a cluster** and **Cluster threshold epsilon** settings.

Data Types: double

##### Update — Enable automatic update of epsilon

false (default) | true

Enable automatic update of the epsilon estimate, specified as false or true.

- When true, the epsilon threshold is first estimated as the average of the knees of the *k*-NN search curves. The estimate is then added to a buffer of size *L*, set by the **Length of cluster threshold epsilon history** parameter. The final value of epsilon is calculated as the average of the *L*-length epsilon history buffer. If **Length of cluster threshold epsilon history** is set to one, the estimate is memory-less. Memory-less means that each epsilon estimate is immediately used and no moving-average smoothing occurs.
- When false, a previous epsilon estimate is used. Estimating epsilon is computationally intensive and not recommended for large data sets.

#### Dependencies

To enable this port, set the **Source of cluster threshold epsilon** parameter to Auto and set the **Maximum number of points for 'Auto' epsilon** parameter.

Data Types: Boolean

##### AmbLims — Ambiguity limits

1-by-2 real-valued vector (default) | 2-by-2 real-valued matrix

Ambiguity limits, specified as a 1-by-2 real-valued vector or 2-by-2 real-valued matrix. For a single ambiguity dimension, specify the limits as a 1-by-2 vector

[*MinAmbiguityLimitDimension1,MaxAmbiguityLimitDimension1*]. For two ambiguity dimensions, specify the limits as a 2-by-2 matrix [*MinAmbiguityLimitDimension1, MaxAmbiguityLimitDimension1; MinAmbiguityLimitDimension2,MaxAmbiguityLimitDimension2*].

Clustering can occur across boundaries to ensure that ambiguous detections are appropriately clustered for up to two dimensions. The ambiguous columns of the input port data X are defined using the **Indices of ambiguous dimensions** parameter. The **AmbLims** parameter defines the minimum and maximum ambiguity limits in the same units as used in the **Indices of ambiguous dimensions** columns of the input data X.

### Dependencies

To enable this port, select the **Enable disambiguation of dimensions** check box.

Data Types: double

### Output

#### Idx — Cluster indices

*N*-by-1 integer-valued column vector

Cluster indices, returned as an *N*-by-1 integer-valued column vector. Cluster IDs represent the clustering results of the DBSCAN algorithm. A value equal to '-1' implies a DBSCAN noise point. Positive Idx values correspond to clusters that satisfy the DBSCAN clustering criteria.

### Dependencies

To enable this port, set the **Define outputs for Simulink block** parameter to Index or Index and ID.

Data Types: double

#### Clusters — Alternative cluster IDs

1-by-*N* integer-valued row vector

Alternative cluster IDs, returned as a 1-by-*N* row vector of positive integers. Each value is a unique identifier indicating a hypothetical target cluster. This argument contains unique positive cluster IDs for all points including noise. In contrast, the Idx output argument labels noise points with '-1'. Use this output as input to Phased Array System Toolbox™ blocks such as Range Estimator and Doppler Estimator.

### Dependencies

To enable this port, set the **Define outputs for Simulink block** parameter to Cluster ID or Index and ID.

Data Types: double

## Parameters

### Define outputs for Simulink block — Type of cluster data output

Index and ID (default) | Cluster ID | Index

Type of cluster data output, specified as:

- Index and ID -- Enables the Idx and Clusters output ports.

- **Cluster ID** -- Enables the `Clusters` output port only.
- **Index** -- Enables the `Idx` output port only.

### Source of cluster threshold epsilon — Epsilon source

Property (default) | Auto

Epsilon source for cluster threshold:

- **Property** — Epsilon is obtained from the **Cluster threshold epsilon** parameter.
- **Auto** — Epsilon is estimated automatically using a  $k$ -nearest neighbor ( $k$ -NN) search. The search is calculated with  $k$  ranging from one less than the value of **Minimum number of points in a cluster** to one less than the value of **Maximum number of points for 'Auto' epsilon**. The subtraction of one is needed because the neighborhood of a point includes the point itself.

### Cluster threshold epsilon — Cluster neighborhood size

10.0 (default) | positive scalar | positive real-valued 1-by- $P$  row vector

Cluster neighborhood size for a search query, specified as a positive scalar or real-valued 1-by- $P$  row vector.  $P$  is the number of clustering dimensions in the input data  $X$ .

Epsilon defines the radius around a point inside which to count the number of detections. When epsilon is a scalar, the same value applies to all clustering feature dimensions. You can specify different epsilon values for different clustering dimensions by specifying a real-valued 1-by- $P$  row vector. Using a row vector creates a multi-dimensional ellipse search area, which is useful when the data columns have different physical meanings such as range and Doppler.

### Minimum number of points in a cluster — Minimum number of points required for cluster

3 (default) | positive integer

Minimum number of points required for a cluster, specified as a positive integer. This parameter defines the minimum number of points in a cluster when determining whether a point is a core point.

### Maximum number of points for 'Auto' epsilon — Maximum number of points required for cluster

10 (default) | positive integer

Maximum number of points in a cluster, specified as a positive integer. This property is used to estimate epsilon when the object performs a  $k$ -NN search.

### Dependencies

To enable this parameter, set the **Source of cluster threshold epsilon** parameter to Auto.

### Length of cluster threshold epsilon history — Length of cluster threshold epsilon history

10 (default) | positive integer

Length of the stored cluster threshold epsilon history, specified as a positive integer. When set to one, the history is memory-less. Then, each epsilon estimate is immediately used and no moving-average smoothing occurs. When greater than one, the epsilon value is averaged over the history length specified.

Example: 5

Data Types: double

**Enable disambiguation of dimensions — Turn on disambiguation**

off (default) | on

Check box to enable disambiguation of dimensions, specified as `false` or `true`. When checked, clustering occurs across boundaries defined by the values in the input port `AmbLims` at execution. Ambiguous detections are appropriately clustered. Use the **Indices of ambiguous dimensions** parameter to specify those column indices of `X` in which ambiguities can occur. Up to two ambiguous dimensions are permitted. Turning on disambiguation is not recommended for large data sets.

Data Types: Boolean

**Indices of ambiguous dimensions — Indices of ambiguous dimensions**

1 (default) | positive integer | 1-by-2 vector of positive integers

Indices of ambiguous dimensions, specified as a positive integer or 1-by-2 vector of positive integers. This property specifies the column indices of the input port data `X` in which disambiguation can occur. A positive integer corresponds to a single ambiguous dimension in the input data matrix `X`. A 1-by-2 length row vector of indices corresponds to two ambiguous dimensions. The size and order of **Indices of ambiguous dimensions** must be consistent with the `AmbLims` input port value.

Example: [3 4]

**Dependencies**

To enable this parameter, select the **Enable disambiguation of dimensions** check box.

Data Types: double

**Simulate using — Block simulation method**

Interpreted Execution (default) | Code Generation

Block simulation, specified as `Interpreted Execution` or `Code Generation`. If you want your block to use the MATLAB interpreter, choose `Interpreted Execution`. If you want your block to run as compiled code, choose `Code Generation`. Compiled code requires time to compile but usually runs faster.

Interpreted execution is useful when you are developing and tuning a model. The block runs the underlying System object™ in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using `Code Generation`. Long simulations run faster with generated code than in interpreted execution. You can run repeated executions without recompiling, but if you change any block parameters, then the block automatically recompiles before execution.

This table shows how the **Simulate using** parameter affects the overall simulation behavior.

When the Simulink® model is in Accelerator mode, the block mode specified using **Simulate using** overrides the simulation mode.

### Acceleration Modes

Block Simulation	Simulation Behavior		
	Normal	Accelerator	Rapid Accelerator
Interpreted Execution	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
Code Generation	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink).

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### See Also

`clusterDBSCAN.discoverClusters` | `clusterDBSCAN.estimateEpsilon` | `clusterDBSCAN`

**Introduced in R2021a**

# Backscatter Bicyclist

Backscatter signals from bicyclist

**Library:** Radar Toolbox



## Description

The Backscatter Bicyclist block simulates backscattered radar signals reflected from a moving bicyclist. The bicyclist consists of the bicycle and its rider. The object models the motion of the bicyclist and computes the sum of all reflected signals from multiple discrete scatterers on the bicyclist. The model ignores internal occlusions within the bicyclist. The reflected signals are computed using a multi-scatterer model developed from a 77-GHz radar system.

Scatterers are located on five major bicyclist components:

- bicycle frame and rider
- bicycle pedals
- upper and lower legs of the rider
- front wheel
- back wheel

Excluding the wheels, there are 114 scatterers on the bicyclist. The wheels contain scatterers on the rim and spokes. The number of scatterers on the wheels depends on the number of spokes per wheel, which can be specified using the `NumWheelSpokes` property.

## Ports

### Input

#### **X — Incident radar signals**

complex-valued  $M$ -by- $N$  matrix

Incident radar signals on each bicyclist scatterer, specified as a complex-valued  $M$ -by- $N$  matrix.  $M$  is the number of samples in the signal.  $N$  is the number of point scatterers on the bicyclist and is determined partly from the number of spokes in each wheel,  $N_{ws}$ . See “Bicyclist Scatterer Indices” on page 2-12 for the column representing the incident signal at each scatterer.

The size of the first dimension of the input matrix can vary to simulate a changing signal length. A size change can occur, for example, in the case of a pulse waveform with variable pulse repetition frequency.

Data Types: `double`

Complex Number Support: Yes

#### **AngH — Bicyclist heading**

0.0 | scalar

Heading of the bicyclist, specified as a scalar. Heading is measured in the  $xy$ -plane from the  $x$ -axis towards the  $y$ -axis. Units are in degrees.

Example: -34

Data Types: `double`

### **Ang — Directions of incident signals**

real-valued 2-by- $N$  vector

Directions of incident signals on the scatterers, specified as a real-valued 2-by- $N$  matrix. Each column of **Ang** specifies the incident direction of the signal to the corresponding scatterer. Each column takes the form of an  $[AzimuthAngle;ElevationAngle]$  pair. Units are in degrees. See “Bicyclist Scatterer Indices” on page 2-12 for the column representing the incident arrival angle at each scatterer.

Data Types: `double`

### **Speed — Bicyclist speed**

nonnegative scalar

Speed of bicyclist, specified as a nonnegative scalar. The motion model limits the speed to 60 m/s. Units are in meters per second.

Example: 8

Data Types: `double`

### **Coast — Bicyclist coasting state**

`false` (default) | `true`

Bicyclist coasting state, specified as `false` or `true`. This property controls the coasting of the bicyclist. If set to `true`, the bicyclist does not pedal but the wheels are still rotating (freewheeling). If set to `false`, the bicyclist is pedaling and the `Gear transmission ratio` parameter determines the ratio of wheel rotations to pedal rotations.

**Tunable:** Yes

Data Types: `Boolean`

## **Output**

### **Y — Combined reflected radar signals**

complex-valued  $M$ -by-1 column vector

Combined reflected radar signals, returned as a complex-valued  $M$ -by-1 column vector.  $M$  equals the number of samples in the input signal,  $X$ .

Data Types: `double`

Complex Number Support: Yes

### **Pos — Positions of scatterers**

real-valued 3-by- $N$  matrix

Positions of scatterers, returned as a real-valued 3-by- $N$  matrix.  $N$  is the number of scatterers on the bicyclist. Each column represents the Cartesian position,  $[x;y;z]$ , of one of the scatterers. Units are in meters. See “Bicyclist Scatterer Indices” on page 2-12 for the column representing the position of each scatterer.



Data Types: double

### **Vel — Velocity scatterers**

real-valued 3-by- $N$  matrix

Velocity of scatterers, returned as a real-valued 3-by- $N$  matrix.  $N$  is the number of scatterers on the bicyclist. Each column represents the Cartesian velocity,  $[v_x; v_y; v_z]$ , of one of the scatterers. Units are in meters per second. See “Bicyclist Scatterer Indices” on page 2-12 for the column representing the velocity of each scatterer.

Data Types: double

### **Ax — Orientation of scatterers**

real-valued 3-by-3 matrix

Orientation axes of scatterers, returned as a real-valued 3-by-3 matrix.

Data Types: double

## **Parameters**

### **Number of wheel spokes — Number of spokes per wheel**

20 (default) | positive integer

Number of spokes per wheel of the bicycle, specified as a positive integer from 3 through 50, inclusive. Units are dimensionless.

Data Types: double

### **Gear transmission ratio — Ratio of wheel rotations to pedal rotations**

1.5 (default) | positive scalar

Ratio of wheel rotations to pedal rotations, specified as a positive scalar. The gear ratio must be in the range 0.5 through 6. Units are dimensionless.

Data Types: double

### **Signal carrier frequency (Hz) — Carrier frequency**

77e9 (default) | positive scalar

Carrier frequency of narrowband incident signals, specified as a positive scalar. Units are in Hz.

Example: 1e9

Data Types: double

### **Initial position (m) — Initial position of bicyclist**

[0;0;0] (default) | 3-by-1 real-valued vector

Initial position of the bicyclist, specified as a 3-by-1 real-valued vector in the form of  $[x;y;z]$ . Units are in meters.

Data Types: double

### **Initial heading direction (deg) — Initial heading of bicyclist**

0 (default) | scalar

Initial heading of the bicyclist, specified as a scalar. Heading is measured in the  $xy$ -plane from the  $x$ -axis towards  $y$ -axis. Units are in degrees.

Data Types: double

### **Initial bicyclist speed (m/s) — Initial speed of bicyclist**

4 (default) | nonnegative scalar

Initial speed of bicyclist, specified as a nonnegative scalar. The motion model limits the speed to a maximum of 60 m/s (216 kph). Units are in meters per second.

**Tunable:** Yes

Data Types: double

### **Propagation speed (m/s) — Signal propagation speed**

physconst('LightSpeed') (default) | positive scalar

Signal propagation speed, specified as a real-valued positive scalar. The default value of the speed of light is the value returned by `physconst('LightSpeed')`.

Data Types: double

### **RCS pattern — Source of RCS pattern**

Auto (default) | Property

Source of the RCS pattern, specified as either Auto or Property. When you specify Auto, the pattern is a 1-by-361 matrix containing values derived from radar measurements taken at 77 GHz.

### **Azimuth angles (deg) — Azimuth angles**

[-180:180] (default) | 1-by- $P$  real-valued row vector |  $P$ -by-1 real-valued column vector

Azimuth angles used to define the angular coordinates of each column of the matrix specified by the **Radar cross section pattern (square meters)** parameter. Specify the azimuth angles as a length  $P$  vector.  $P$  must be greater than two. Angle units are in degrees.

Example: [-45:0:1:45]

#### **Dependencies**

To enable this parameter, set the **RCS pattern** parameter to Property.

Data Types: double

### **Elevation angles (deg) — Elevation angles**

[-90:90] (default) | 1-by- $Q$  real-valued row vector |  $Q$ -by-1 real-valued column vector

Elevation angles used to define the angular coordinates of each row of the matrix specified by the **Radar cross section pattern (square meters)** parameter. Specify the elevation angles as a length  $Q$  vector.  $Q$  must be greater than two. Angle units are in degrees.

#### **Dependencies**

To enable this parameter, set the **RCS pattern** parameter to Property.

Data Types: double

### **Radar cross section pattern (square meters) — Radar cross-section pattern**

1-by-361 real-valued matrix (default) |  $Q$ -by- $P$  real-valued matrix | 1-by- $P$  real-valued vector

Radar cross-section (RCS) pattern as a function of elevation and azimuth angle, specified as a  $Q$ -by- $P$  real-valued matrix or a 1-by- $P$  real-valued vector.  $Q$  is the length of the vector defined by the `ElevationAngles` property.  $P$  is the length of the vector defined by the `AzimuthAngles` property. Units are in square meters.

You can also specify the pattern as a 1-by- $P$  real-valued vector of azimuth angles for one elevation.

The default value of this property is a 1-by-361 matrix containing values derived from radar measurements taken at 77 GHz found in `backscatterBicyclist.defaultRCSPattern`.

### Dependencies

To enable this parameter, set the **RCS pattern** parameter to Property.

Data Types: double

### Simulate using – Block simulation method

Interpreted Execution (default) | Code Generation

Block simulation, specified as `Interpreted Execution` or `Code Generation`. If you want your block to use the MATLAB interpreter, choose `Interpreted Execution`. If you want your block to run as compiled code, choose `Code Generation`. Compiled code requires time to compile but usually runs faster.

Interpreted execution is useful when you are developing and tuning a model. The block runs the underlying System object in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using `Code Generation`. Long simulations usually run faster as compiled code than interpreted execution. You can run repeated executions without recompiling, but if you change any block parameters, then the block automatically recompiles before execution.

This table shows how the **Simulate using** parameter affects the overall simulation behavior.

When the Simulink model is in Accelerator mode, the block mode specified using **Simulate using** overrides the simulation mode.

### Acceleration Modes

Block Simulation	Simulation Behavior		
	Normal	Accelerator	Rapid Accelerator
Interpreted Execution	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
Code Generation	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink).

## More About

### Bicyclist Scatterer Indices

Bicyclist scatterer indices define which columns in the scatterer position or velocity matrices contain the position and velocity data for a specific scatterer. For example, column 92 of `bpos` specifies the 3-D position of one of the scatterers on a pedal.

The wheel scatterers are equally divided between the wheels. You can determine the total number of wheel scatterers,  $N$ , by subtracting 113 from the output of the `getNumScatterers` function. The number of scatterers per wheel is  $N_{sw} = N/2$ .

### Bicyclist Scatterer Indices

Bicyclist Component	Bicyclist Scatterer Index
Frame and rider	1 ... 90
Pedals	91 ... 99
Rider legs	100 ... 113
Front wheel	114 ... 114 + $N_{sw}$ - 1
Rear wheel	114 + $N_{sw}$ ... 114 + $N$ - 1

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Objects

`backscatterBicyclist` | `phased.BackscatterRadarTarget` | `phased.RadarTarget`

### Blocks

Backscatter Radar Target | Radar Target | Backscatter Pedestrian

### Introduced in R2021a

# Backscatter Pedestrian

Backscatter signals from pedestrian

**Library:** Radar Toolbox



## Description

The Backscatter Pedestrian block models the monostatic reflection of non-polarized electromagnetic signals from a walking pedestrian. The pedestrian walking model coordinates the motion of 16 body segments to simulate natural motion. The model also simulates the radar reflectivity of each body segment. From this model, you can obtain the position and velocity of each segment and the total backscattered radiation as the body moves.

## Ports

### Input

#### **X — Incident radar signals**

complex-valued  $M$ -by-16 matrix

Incident radar signals on each body segment, specified as a complex-valued  $M$ -by-16 matrix.  $M$  is the number of samples in the signal. See “Body Segment Indices” on page 2-16 for the column representing the incident signal at each body segment.

The size of the first dimension of the input matrix can vary to simulate a changing signal length. A size change can occur, for example, in the case of a pulse waveform with variable pulse repetition frequency.

Data Types: `double`

Complex Number Support: Yes

#### **Ang — Incident signal directions**

real-valued 2-by-16 matrix

Incident signal directions on the body segments, specified as a real-valued 2-by-16 matrix. Each column of ANG specifies the incident direction of the signal to the corresponding body part. Each column takes the form of an [AzimuthAngle;ElevationAngle] pair. Units are in degrees. See “Body Segment Indices” on page 2-16 for the column representing the incident direction at each body segment.

Data Types: `double`

#### **AngH — Pedestrian heading**

scalar

Heading of the pedestrian, specified as a scalar. Heading is measured in the  $xy$ -plane from the  $x$ -axis towards the  $y$ -axis. Units are in degrees.

Example: -34

Data Types: double

### Output

#### **Y — Combined reflected radar signals**

complex-valued  $M$ -by-1 column vector

Combined reflected radar signals, returned as a complex-valued  $M$ -by-1 column vector.  $M$  equals the same number of samples as in the input signal,  $X$ .

Data Types: double

Complex Number Support: Yes

#### **Pos — Positions of body segments**

real-valued 3-by-16 matrix

Positions of body segments, returned as a real-valued 3-by-16 matrix. Each column represents the Cartesian position,  $[x; y; z]$ , of one of 16 body segments. Units are in meters. See “Body Segment Indices” on page 2-16 for the column representing the position of each body segment.

Data Types: double

#### **Vel — Velocity of body segments**

real-valued 3-by-16 matrix

Velocity of body segments, returned as a real-valued 3-by-16 matrix. Each column represents the Cartesian velocity,  $[vx; vy; vz]$ , of one of 16 body segments. Units are in meters per second. See “Body Segment Indices” on page 2-16 for the column representing the velocity of each body segment.

Data Types: double

#### **Ax — Orientation of body segments**

real-valued 3-by-3-by-16 array

Orientation axes of body segments, returned as a real-valued 3-by-3-by-16 array. Each page represents the 3-by-3 orientation axes of one of 16 body segments. Units are dimensionless. See “Body Segment Indices” on page 2-16 for the page representing the orientation of each body segment.

Data Types: double

## Parameters

#### **Height (m) — Height of pedestrian**

1.65 (default) | positive scalar

Height of pedestrian, specified as a positive scalar. Units are in meters.

Data Types: double

#### **Walking Speed (m/s) — Walking speed of pedestrian**

1.4 times pedestrian height (default) | nonnegative scalar

Walking speed of the pedestrian, specified as a nonnegative scalar. The motion model limits the walking speed to 1.4 times the pedestrian height set in the **Height (m)** parameter. Units are in meters per second.

Data Types: double

### Propagation speed (m/s) — Signal propagation speed

`physconst('LightSpeed')` (default) | positive scalar

Signal propagation speed, specified as a real-valued positive scalar. The default value of the speed of light is the value returned by `physconst('LightSpeed')`.

Data Types: double

### Operating Frequency (Hz) — Carrier frequency

`300e6` (default) | positive scalar

Carrier frequency of narrowband incident signals, specified as a positive scalar. Units are in Hz.

Example: `1e9`

Data Types: double

### Initial Position (m) — Initial position of pedestrian

`[0;0;0]` (default) | 3-by-1 real-valued vector

Initial position of the pedestrian, specified as a 3-by-1 real-valued vector in the form of `[x;y;z]`. Units are in meters.

Data Types: double

### Initial Heading (deg) — Initial heading of pedestrian

`0` (default) | scalar

Initial heading of the pedestrian, specified as a scalar. Heading is measured in the *xy*-plane from the *x*-axis towards *y*-axis. Units are in degrees.

Data Types: double

### Simulate using — Block simulation method

`Interpreted Execution` (default) | `Code Generation`

Block simulation, specified as `Interpreted Execution` or `Code Generation`. If you want your block to use the MATLAB interpreter, choose `Interpreted Execution`. If you want your block to run as compiled code, choose `Code Generation`. Compiled code requires time to compile but usually runs faster.

`Interpreted execution` is useful when you are developing and tuning a model. The block runs the underlying System object in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using `Code Generation`. Long simulations run faster than in interpreted execution. You can run repeated executions without recompiling, but if you change any block parameters, then the block automatically recompiles before execution.

This table shows how the **Simulate using** parameter affects the overall simulation behavior.

When the Simulink model is in Accelerator mode, the block mode specified using **Simulate using** overrides the simulation mode.

## Acceleration Modes

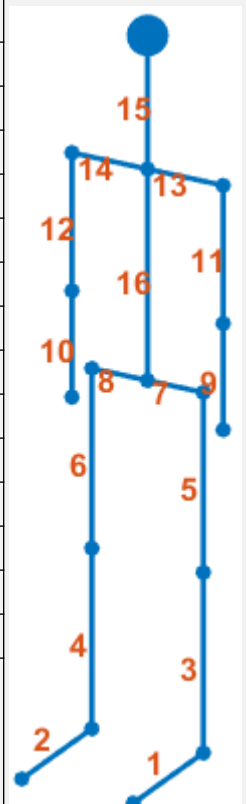
Block Simulation	Simulation Behavior		
	Normal	Accelerator	Rapid Accelerator
Interpreted Execution	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
Code Generation	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink).

## More About

### Body Segment Indices

Body segment indices define which columns in the **X**, **Ang**, **BPPOS**, and **BPVEL** ports contain the data for a specific body segment. Body segment indices define which page in the **Ax** port contains the data for a specific body segments. For example, column 3 of **X** contains sample data for the left lower leg. Column 3 of **Ang** contains the arrival angle of the signal at the left lower leg.

Body Segment	Index	
Left foot	1	
Right foot	2	
Left lower leg	3	
Right lower leg	4	
Left upper leg	5	
Right upper leg	6	
Left hip	7	
Right hip	8	
Left lower arm	9	
Right lower arm	10	
Left upper arm	11	
Right upper arm	12	
Left shoulder	13	
Right shoulder	14	
Head	15	
Torso	16	



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Objects

[backscatterPedestrian](#) | [backscatterBicyclist](#) | [phased.BackscatterRadarTarget](#) | [phased.RadarTarget](#)

### Blocks

[Backscatter Radar Target](#) | [Radar Target](#) | [Backscatter Bicyclist](#)

## Introduced in R2021a

## Barrage Jammer

Barrage jammer interference source



## Library

Radar Toolbox

## Description

The Barrage Jammer block generates a wideband noise-like jamming signal.

## Parameters

### Effective radiated power (W)

Specify the effective radiated power (ERP) in watts of the jamming signal as a positive scalar.

### Source of number of samples per frame

Specify the source for number of samples per frame as `Property` or `Derive from reference input port`. When you choose `Property`, the block obtains the number of samples from the **Number of samples per frame** parameter. When you choose `Derive from reference input port` the block uses the number of samples from a reference signal passed into the `Ref` input port.

### Number of samples per frame

Specify the number of samples in the jamming signal output as a positive integer. The number of samples must match the number of samples produced by a signal source. This parameter appears only when **Source of number of samples per frame** is set to `Property`. As an example, if you use the Rectangular Waveform block as a signal source and set its **Output signal format** to `Samples`, the value of **Number of samples per frame** should match the Rectangular Waveform block's **Number of samples in output** parameter. If you set the **Output signal format** to `Pulses`, the **Number of samples per frame** should match the product of **Sample rate** and **Number of pulses in output** divided by the **Pulse repetition frequency**.

### Simulate using

Block simulation method, specified as `Interpreted Execution` or `Code Generation`. If you want your block to use the MATLAB interpreter, choose `Interpreted Execution`. If you want your block to run as compiled code, choose `Code Generation`. Compiled code requires time to compile but usually runs faster.

Interpreted execution is useful when you are developing and tuning a model. The block runs the underlying System object in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using `Code Generation`. Long simulations run faster than they would in interpreted execution. You can run repeated executions without recompiling. However, if you change any block parameters, then the block automatically recompiles before execution.

When setting this parameter, you must take into account the overall model simulation mode. The table shows how the **Simulate using** parameter interacts with the overall simulation mode.

When the Simulink model is in Accelerator mode, the block mode specified using **Simulate using** overrides the simulation mode.

### Acceleration Modes

Block Simulation	Simulation Behavior		
	Normal	Accelerator	Rapid Accelerator
Interpreted Execution	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
Code Generation	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink).

## Ports

**Note** The block input and output ports correspond to the input and output parameters described in the `step` method of the underlying System object. See link at the bottom of this page.

Port	Description	Supported Data Types
Ref	Reference signal input	Double-precision floating point
Out	Jammer output	Double-precision floating point

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

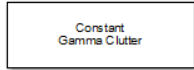
barrageJammer

**Introduced in R2021a**

## Constant Gamma Clutter

Constant gamma clutter simulation

**Library:** Radar Toolbox



### Description

The Constant Gamma Clutter block generates constant gamma clutter reflected from homogeneous terrain for a monostatic radar transmitting a narrowband signal into free space. The radar is assumed to be at constant altitude moving at constant speed.

### Ports

#### Input

##### PRFIdx — PRF Index

positive integer

Index to select the pulse repetition frequency (PRF), specified as a positive integer. The index selects the PRF from the predefined vector of values specified by the **Pulse repetition frequency (Hz)** parameter.

Example: 4

##### Dependencies

To enable this port, select **Enable PRF selection input**.

Data Types: double

##### W — Element weights

length- $N$  complex-valued vector

Weights applied to each element in array, specified as a length- $N$  complex-valued vector.  $N$  is the number of elements in the array selected in the **Sensor array** panel.

##### Dependencies

To enable this port, select the **Enable weights input** check box.

Data Types: double

##### WS — Subarray element weights

$N_E$ -by- $N_S$  complex-valued matrix

Weights applied to each element in a subarray, specified as an  $N_E$ -by- $N_S$  complex-valued matrix.

- When you set **Specify sensor array** to **Replicated Subarray**, all subarrays have the same dimensions. Then, you can specify the subarray element weights as a complex-valued  $N_E$ -by- $N_S$  matrix.  $N_E$  is the number of elements in each subarray and  $N_S$  is the number of subarrays. Each column of  $WS$  specifies the weights for the corresponding subarray.

- When you set **Specify sensor array** to `Partitioned array`, subarrays are not required to have identical dimensions and sizes. You can specify subarray element weights as a complex-valued  $N_E$ -by- $N_S$  matrix, where  $N_E$  now is the number of elements in the largest subarray. The first  $K$  entries in each column are the element weights for the corresponding subarray where  $K$  is the number of elements in the subarray.

### Dependencies

To enable this port, set **Specify sensor array** to `Partitioned array` or `Replicated Subarray`. Then, set **Subarray steering method** to `Custom`.

Data Types: `double`

### Steer — Steering angle input

scalar | 2-by-1 real-valued vector

Steering angle, specified as a scalar or a 2-by-1 real-valued vector. As a vector, the steering angle takes the form of `[AzimuthAngle; ElevationAngle]`. As a scalar, the steering angle represents the azimuth angle only. Then the elevation angle is assumed to be zero degrees. Units are in degrees.

### Dependencies

To enable this port, set **Specify sensor array** to `Partitioned array` or `Replicated Subarray`. Then, set **Subarray steering method** to `Phase` or `Time`.

Data Types: `double`

### Output

#### Out — Simulated clutter

$N$ -by- $M$  complex-valued matrix

Simulated clutter, returned as an  $N$ -by- $M$  complex-valued matrix.

$N$  is the number of samples output from the block. When you set the **Output signal format** parameter to `Samples`, specify  $N$  using the **Number of samples in output** parameter. When you set the **Output signal format** parameter to `Pulses`,  $N$  is the total number of samples in the next  $P$  pulses where  $P$  is specified in the **Number of pulse in output** parameter.

$M$  is either

- the number of subarrays in the sensor array if sensor array contains subarrays.
- the number of radiating or collecting elements if the sensor array does not contain subarrays.

Data Types: `double`

## Parameters

### Main Tab

#### Terrain gamma value (dB) — Clutter model parameter

0 (default) | scalar

Clutter model parameter, specified as a scalar. This parameter contains the  $\gamma$  value used in the constant  $\gamma$  clutter model. The  $\gamma$  value depends on both terrain type and the operating frequency. Units are in dB.

Example: -5.0

Data Types: double

**Earth model – Earth shape**

Flat (default) | Curved

Specify the earth model used in clutter simulation as Flat or Curved. When you set this parameter to Flat, the earth is assumed to be a plane. When you set this parameter to Curved, the earth is assumed to be spherical.

**Minimum range of clutter region (m) – Minimum range of clutter region**

0 | nonnegative scalar

Specify the minimum range for the clutter simulation as a positive scalar. The minimum range must be nonnegative. Units are in meters.

**Maximum range of clutter region (m) – Maximum range of clutter region**

5000 | nonnegative scalar

Specify the maximum range for the clutter simulation as a positive scalar. The maximum range must be greater than the value specified in the Radar height parameter. Units are in meters.

**Azimuth center of clutter region (deg) – Azimuth center of clutter region**

0 | scalar

The azimuth angle in the ground plane about which clutter patches are generated. Patches are generated symmetrically about this angle. Units are in degrees.

**Azimuth span of clutter region (deg) – Azimuth span of clutter region**

60 (default) | positive scalar

Specify the azimuth span of each clutter patch as a positive scalar. Units are in degrees. Units are in degrees.

**Azimuth span of clutter patches (deg) – Azimuth span of clutter patches**

1 (default) | positive scalar

Azimuth span of each clutter patch, specified as a positive scalar. Units are in degrees.

Data Types: double

**Clutter coherence time (s) – Coherence time of clutter simulation**

Inf (default) | positive scalar

Coherence time for the clutter simulation, specified as a positive scalar. After the coherence time elapses, the block updates the random numbers it uses for the clutter simulation at the next pulse. When you use the default value of Inf, the random numbers are never updated. Units are in seconds.

Example: 4

Data Types: double

**Signal propagation speed (m/s) – Signal propagation speed**

physconst('LightSpeed') (default) | real-valued positive scalar

Signal propagation speed, specified as a real-valued positive scalar. The default value of the speed of light is the value returned by `physconst('LightSpeed')`. Units are in meters per second.

Example: `3e8`

Data Types: `double`

### **Sample rate (Hz) – Clutter sample rate**

`1e6` (default) | positive scalar

Clutter sample rate, specified as a positive scalar. Units are in Hertz.

Example: `10e6`

Data Types: `double`

### **Pulse repetition frequency (Hz) – Pulse repetition frequency**

`1e4` (default) | positive scalar | row vector of positive values

Pulse repetition frequency, PRF, specified as a positive scalar or a row vector of positive values. Units are in Hertz.

Example: `[1e4, 2e4]`

Data Types: `double`

### **Enable PRF selection input – Select predefined PRF**

`off` (default) | `on`

Select this parameter to enable the `PRFIdx` port.

- When enabled, pass in an index into a vector of predefined PRFs. Set predefined PRFs using the **Pulse repetition frequency (Hz)** parameter.
- When not enabled, the block cycles through the vector of PRFs specified by the **Pulse repetition frequency (Hz)** parameter. If **Pulse repetition frequency (Hz)** is a scalar, the PRF is constant.

### **Source of simulation sample time – Source of simulation sample time**

`Derive from waveform parameters` (default) | `Inherit from Simulink engine`

Source of simulation sample time, specified as `Derive from waveform parameters` or `Inherit from Simulink engine`. When set to `Derive from waveform parameters`, the block runs at a variable rate determined by the PRF of the selected waveform. The elapsed time is variable. When set to `Inherit from Simulink engine`, the block runs at a fixed rate so the elapsed time is a constant.

### **Dependencies**

To enable this parameter, select the **Enable PRF selection input** parameter.

### **Output signal format – Format of the output signal**

`Pulses` (default) | `Samples`

The format of the output signal, specified as `Pulses` or `Samples`.

If you set this parameter to `Samples`, the output of the block consists of multiple samples. The number of samples is the value of the **Number of samples in output** parameter.

If you set this parameter to **Pulses**, the output of the block consists of multiple pulses. The number of pulses is the value of the **Number of pulses in output** parameter.

**Number of samples in output – Number of samples in output**

100 (default) | positive integer

Number of samples in the block output, specified as a positive integer.

Example: 1000

**Dependencies**

To enable this parameter, set the **Output signal format** parameter to **Samples**.

Data Types: double

**Number of pulses in output – Number of pulses in output**

1 (default) | positive integer

Number of pulses in the block output, specified as a positive integer.

Example: 2

**Dependencies**

To enable this parameter, set the **Output signal format** parameter to **Pulses**.

Data Types: double

**Simulate using – Block simulation method**

Interpreted Execution (default) | Code Generation

Block simulation, specified as **Interpreted Execution** or **Code Generation**. If you want your block to use the MATLAB interpreter, choose **Interpreted Execution**. If you want your block to run as compiled code, choose **Code Generation**. Compiled code requires time to compile but usually runs faster.

Interpreted execution is useful when you are developing and tuning a model. The block runs the underlying System object in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using **Code Generation**. Long simulations run faster with generated code than in interpreted execution. You can run repeated executions without recompiling, but if you change any block parameters, then the block automatically recompiles before execution.

This table shows how the **Simulate using** parameter affects the overall simulation behavior.

When the Simulink model is in **Accelerator mode**, the block mode specified using **Simulate using** overrides the simulation mode.



**Acceleration Modes**

Block Simulation	Simulation Behavior		
	Normal	Accelerator	Rapid Accelerator
Interpreted Execution	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
Code Generation	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink).

**Radar Tab****Operating frequency (Hz) — System operating frequency**

3.0e8 (default) | positive real scalar

System operating frequency, specified as a positive scalar. Units are in Hz.

**Effective transmitted power (W) — radar system effective transmitted power**

5000 (default) | positive scalar

Effective radiated power (ERP) of the radar system, specified as a positive scalar. Units are in watts.

Example: 3500

Data Types: double

**Radar height (m) — Height of radar above surface**

0 (default) | nonnegative scalar

Height of radar above surface, specified as a nonnegative scalar. Units are in meters.

Example: 50

Data Types: double

**Radar speed (m/s) — Radar platform speed**

0 (default) | nonnegative scalar

Radar platform speed, specified as a nonnegative scalar. Units are in meters per second.

Example: 5

Data Types: double

**Radar motion direction (deg) — Direction of motion of radar platform**

[90;0] (default) | 2-by-1 real vector

Specify the direction of radar platform motion as a 2-by-1 real vector in the form [AzimuthAngle;ElevationAngle]. Units are in degrees. Both azimuth and elevation angle are measured in the local coordinate system of the radar antenna or antenna array. Azimuth angle must be between  $-180^\circ$  and  $180^\circ$ . Elevation angle must be between  $-90^\circ$  and  $90^\circ$ .

The default value of this parameter indicates that the radar platform is moving perpendicular to the radar antenna array broadside direction.

Example: [25;30]

Data Types: double

### Sensor mounting angles sensor (deg) — Sensor mounting angles

[0 0 0] (default) | length-3 vector of positive values

Specify a 3-element vector that gives the intrinsic yaw, pitch, and roll of the sensor frame from the inertial frame. The 3 elements define the rotations around the z, y, and x axes respectively, in that order. The first rotation, rotates the body axes around the z-axis. Because these angles define intrinsic rotations, the second rotation is performed around the y-axis in its new position resulting from the previous rotation. The final rotation around the x-axis is performed around the x-axis as rotated by the first two rotations in the intrinsic system.

Example: [0, -10, 4]

Data Types: double

### Enable weights input — Enable antenna element weights input port

unchecked (default) | checked

Check box to enable antenna element weights input port, W.

## Sensor Array Tab

### Specify sensor array as — Method to specify array

Array (no subarrays) (default) | Partitioned array | Replicated subarray | MATLAB expression

Method to specify array, specified as Array (no subarrays) or MATLAB expression.

- Array (no subarrays) — use the block parameters to specify the array.
- Partitioned array — use the block parameters to specify the array.
- Replicated subarray — use the block parameters to specify the array.
- MATLAB expression — create the array using a MATLAB expression.

### Expression — MATLAB expression used to create an array

Phased Array System Toolbox array System object

MATLAB expression used to create an array, specified as a valid Phased Array System Toolbox array System object.

Example: phased.URA('Size',[5,3])

### Dependencies

To enable this parameter, set **Specify sensor array as** to MATLAB expression.

### Element Parameters

#### Element type — Array element types

Isotropic Antenna (default) | Cosine Antenna | Custom Antenna | Omni Microphone | Custom Microphone

Antenna or microphone type, specified as one of the following:

- Isotropic Antenna
- Cosine Antenna
- Custom Antenna
- Omni Microphone
- Custom Microphone

### **Operating frequency range (Hz) — Operating frequency range of the antenna or microphone element**

[0, 1.0e20] (default) | real-valued 1-by-2 row vector

Specify the operating frequency range of the antenna or microphone element as a 1-by-2 row vector in the form [LowerBound, UpperBound]. The element has no response outside this frequency range. Frequency units are in Hz.

#### **Dependencies**

To enable this parameter, set **Element type** to Isotropic Antenna, Cosine Antenna, or Omni Microphone.

### **Operating frequency vector (Hz) — Operating frequency range of custom antenna or microphone elements**

[0, 1.0e20] (default) | real-valued row vector

Specify the frequencies at which to set antenna and microphone frequency responses as a 1-by-*L* row vector of increasing real values. The antenna or microphone element has no response outside the frequency range specified by the minimum and maximum elements of this vector. Frequency units are in Hz.

#### **Dependencies**

To enable this parameter, set **Element type** to Custom Antenna or Custom Microphone. Use **Frequency responses (dB)** to set the responses at these frequencies.

### **Baffle the back of the element — Set back response of an Isotropic Antenna element or an Omni Microphone element to zero**

off (default) | on

Select this check box to baffle the back response of the element. When back baffled, the responses at all azimuth angles beyond  $\pm 90^\circ$  from broadside are set to zero. The broadside direction is defined as  $0^\circ$  azimuth angle and  $0^\circ$  elevation angle.

#### **Dependencies**

To enable this check box, set **Element type** to Isotropic Antenna or Omni Microphone.

### **Exponent of cosine pattern — Exponents of azimuth and elevation cosine patterns**

[1.5 1.5] (default) | nonnegative scalar | real-valued 1-by-2 matrix of nonnegative values

Specify the exponents of the cosine pattern as a nonnegative scalar or a real-valued 1-by-2 matrix of nonnegative values. When **Exponent of cosine pattern** is a 1-by-2 vector, the first element is the exponent in the azimuth direction and the second element is the exponent in the elevation direction. When you set this parameter to a scalar, both the azimuth direction and elevation direction cosine patterns are raised to the same power.

**Dependencies**

To enable this parameter, set **Element type** to Cosine Antenna.

**Frequency responses (dB) — Antenna and microphone frequency response**

[0, 0] (default) | real-valued row vector

Frequency response of a custom antenna or custom microphone for the frequencies defined by the **Operating frequency vector (Hz)** parameter. The dimensions of **Frequency responses (dB)** must match the dimensions of the vector specified by the **Operating frequency vector (Hz)** parameter.

**Dependencies**

To enable this parameter, set **Element type** to Custom Antenna or Custom Microphone.

**Input Pattern Coordinate System — Coordinate system of custom antenna pattern**

az-el (default) | phi-theta

Coordinate system of custom antenna pattern, specified az-el or phi-theta. When you specify az-el, use the **Azimuth angles (deg)** and **Elevations angles (deg)** parameters to specify the coordinates of the pattern points. When you specify phi-theta, use the **Phi angles (deg)** and **Theta angles (deg)** parameters to specify the coordinates of the pattern points.

**Dependencies**

To enable this parameter, set **Element type** to Custom Antenna.

**Azimuth angles (deg) — Azimuth angles of antenna radiation pattern**

[-180:180] (default) | real-valued row vector

Specify the azimuth angles at which to calculate the antenna radiation pattern as a 1-by- $P$  row vector.  $P$  must be greater than 2. Azimuth angles must lie between  $-180^\circ$  and  $180^\circ$ , inclusive, and be in strictly increasing order.

**Dependencies**

To enable this parameter, set the **Element type** parameter to Custom Antenna and the **Input Pattern Coordinate System** parameter to az-el.

**Elevation angles (deg) — Elevation angles of antenna radiation pattern**

[-90:90] (default) | real-valued row vector

Specify the elevation angles at which to compute the radiation pattern as a 1-by- $Q$  vector.  $Q$  must be greater than 2. Angle units are in degrees. Elevation angles must lie between  $-90^\circ$  and  $90^\circ$ , inclusive, and be in strictly increasing order.

**Dependencies**

To enable this parameter, set the **Element type** parameter to Custom Antenna and the **Input Pattern Coordinate System** parameter to az-el.

**Phi Angles (deg) — Phi angle coordinates of custom antenna radiation pattern**

0:360 | real-valued 1-by- $P$  row vector

Phi angles of points at which to specify the antenna radiation pattern, specify as a real-valued 1-by- $P$  row vector.  $P$  must be greater than 2. Angle units are in degrees. Phi angles must lie between  $0^\circ$  and  $360^\circ$  and be in strictly increasing order.

**Dependencies**

To enable this parameter, set the **Element type** parameter to Custom Antenna and the **Input Pattern Coordinate System** parameter to phi - theta.

**Theta Angles (deg) — Theta angle coordinates of custom antenna radiation pattern**

0:180 | real-valued 1-by- $Q$  row vector

Theta angles of points at which to specify the antenna radiation pattern, specify as a real-valued 1-by- $Q$  row vector.  $Q$  must be greater than 2. Angle units are in degrees. Theta angles must lie between 0° and 360° and be in strictly increasing order.

**Dependencies**

To enable this parameter, set the **Element type** parameter to Custom Antenna and the **Input Pattern Coordinate System** parameter to phi - theta.

**Magnitude pattern (dB) — Magnitude of combined antenna radiation pattern**

zeros(181,361) (default) | real-valued  $Q$ -by- $P$  matrix | real-valued  $Q$ -by- $P$ -by- $L$  array

Magnitude of the combined antenna radiation pattern, specified as a  $Q$ -by- $P$  matrix or a  $Q$ -by- $P$ -by- $L$  array.

- When the **Input Pattern Coordinate System** parameter is set to az - el,  $Q$  equals the length of the vector specified by the **Elevation angles (deg)** parameter and  $P$  equals the length of the vector specified by the **Azimuth angles (deg)** parameter.
- When the **Input Pattern Coordinate System** parameter is set to phi - theta,  $Q$  equals the length of the vector specified by the **Theta Angles (deg)** parameter and  $P$  equals the length of the vector specified by the **Phi Angles (deg)** parameter.

The quantity  $L$  equals the length of the **Operating frequency vector (Hz)**.

- If this parameter is a  $Q$ -by- $P$  matrix, the same pattern is applied to *all* frequencies specified in the **Operating frequency vector (Hz)** parameter.
- If the value is a  $Q$ -by- $P$ -by- $L$  array, each  $Q$ -by- $P$  page of the array specifies a pattern for the *corresponding* frequency specified in the **Operating frequency vector (Hz)** parameter.

**Dependencies**

To enable this parameter, set **Element type** to Custom Antenna.

**Phase pattern (deg) — Custom antenna radiation phase pattern**

zeros(181,361) (default) | real-valued  $Q$ -by- $P$  matrix | real-valued  $Q$ -by- $P$ -by- $L$  array

Phase of the combined antenna radiation pattern, specified as a  $Q$ -by- $P$  matrix or a  $Q$ -by- $P$ -by- $L$  array.

- When the **Input Pattern Coordinate System** parameter is set to az - el,  $Q$  equals the length of the vector specified by the **Elevation angles (deg)** parameter and  $P$  equals the length of the vector specified by the **Azimuth angles (deg)** parameter.
- When the **Input Pattern Coordinate System** parameter is set to phi - theta,  $Q$  equals the length of the vector specified by the **Theta Angles (deg)** parameter and  $P$  equals the length of the vector specified by the **Phi Angles (deg)** parameter.

The quantity  $L$  equals the length of the **Operating frequency vector (Hz)**.

- If this parameter is a  $Q$ -by- $P$  matrix, the same pattern is applied to *all* frequencies specified in the **Operating frequency vector (Hz)** parameter.
- If the value is a  $Q$ -by- $P$ -by- $L$  array, each  $Q$ -by- $P$  page of the array specifies a pattern for the *corresponding* frequency specified in the **Operating frequency vector (**

#### Dependencies

To enable this parameter, set **Element type** to Custom Antenna.

#### MatchArrayNormal — Rotate antenna element to array normal

on (default) | off

Select this check box to rotate the antenna element pattern to align with the array normal. When not selected, the element pattern is not rotated.

When the antenna is used in an antenna array and the **Input Pattern Coordinate System** parameter is `az-el`, selecting this check box rotates the pattern so that the x-axis of the element coordinate system points along the array normal. Not selecting uses the element pattern without the rotation.

When the antenna is used in an antenna array and **Input Pattern Coordinate System** is set to `phi-theta`, selecting this check box rotates the pattern so that the z-axis of the element coordinate system points along the array normal.

Use the parameter in conjunction with the **Array normal** parameter of the URA and UCA arrays.

#### Dependencies

To enable this parameter, set **Element type** to Custom Antenna.

#### Polar pattern frequencies (Hz) — Polar pattern microphone response frequencies

1e3 (default) | real scalar | real-valued 1-by- $L$  row vector

Polar pattern microphone response frequencies, specified as a real scalar, or a real-valued, 1-by- $L$  vector. The response frequencies lie within the frequency range specified by the **Operating frequency vector (Hz)** vector.

#### Dependencies

To enable this parameter, set **Element type** set to Custom Microphone.

#### Polar pattern angles (deg) — Polar pattern response angles

[-180:180] (default) | real-valued -by- $P$  row vector

Specify the polar pattern response angles, as a 1-by- $P$  vector. The angles are measured from the central pickup axis of the microphone and must be between  $-180^\circ$  and  $180^\circ$ , inclusive.

#### Dependencies

To enable this parameter, set **Element type** to Custom Microphone.

#### Polar pattern (dB) — Custom microphone polar response

zeros(1,361) (default) | real-valued  $L$ -by- $P$  matrix

Specify the magnitude of the custom microphone element polar patterns as an  $L$ -by- $P$  matrix.  $L$  is the number of frequencies specified in **Polar pattern frequencies (Hz)**.  $P$  is the number of angles

specified in **Polar pattern angles (deg)**. Each row of the matrix represents the magnitude of the polar pattern measured at the corresponding frequency specified in **Polar pattern frequencies (Hz)** and all angles specified in **Polar pattern angles (deg)**. The pattern is measured in the azimuth plane. In the azimuth plane, the elevation angle is 0° and the central pickup axis is 0° degrees azimuth and 0° degrees elevation. The polar pattern is symmetric around the central axis. You can construct the microphone response pattern in 3-D space from the polar pattern.

#### Dependencies

To enable this parameter, set **Element type** to Custom Microphone.

#### Array Parameters

##### Geometry — Array geometry

ULA (default) | URA | UCA | Conformal Array

Array geometry, specified as one of

- ULA — Uniform linear array
- URA — Uniform rectangular array
- UCA — Uniform circular array
- Conformal Array — arbitrary element positions

##### Number of elements — Number of array elements

2 for ULA arrays and 5 for UCA arrays (default) | integer greater than or equal to 2

The number of array elements for ULA or UCA arrays, specified as an integer greater than or equal to 2.

When you set **Specify sensor array as** to Replicated subarray, this parameter applies to each subarray.

#### Dependencies

To enable this parameter, set **Geometry** to ULA or UCA.

##### Element spacing (m) — Spacing between array elements

0.5 for ULA arrays and [0.5, 0.5] for URA arrays (default) | positive scalar for ULA or URA arrays | 2-element vector of positive values for URA arrays

Spacing between adjacent array elements:

- ULA — specify the spacing between two adjacent elements in the array as a positive scalar.
- URA — specify the spacing as a positive scalar or a 1-by-2 vector of positive values. If **Element spacing (m)** is a scalar, the row and column spacings are equal. If **Element spacing (m)** is a vector, the vector has the form `[SpacingBetweenArrayRows, SpacingBetweenArrayColumns]`.
- When you set **Specify sensor array as** to Replicated subarray, this parameter applies to each subarray.

#### Dependencies

To enable this parameter, set **Geometry** to ULA or URA.

**Array axis — Linear axis direction of ULA**

y (default) | x | z

Linear axis direction of ULA, specified as y, x, or z. All ULA array elements are uniformly spaced along this axis in the local array coordinate system.

**Dependencies**

- To enable this parameter, set **Geometry** to ULA.
- This parameter is also enabled when the block only supports ULA arrays.

**Array size — Dimensions of URA array**

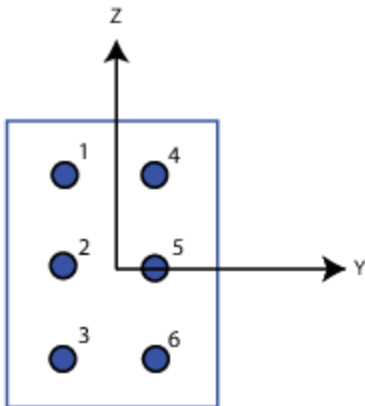
[2, 2] (default) | positive integer | 1-by-2 vector of positive integers

Dimensions of a URA array, specified as a positive integer or 1-by-2 vector of positive integers.

- If **Array size** is a 1-by-2 vector, the vector has the form [NumberOfArrayRows, NumberOfArrayColumns].
- If **Array size** is an integer, the array has the same number of rows and columns.
- When you set **Specify sensor array as** to Replicated subarray, this parameter applies to each subarray.

For a URA, array elements are indexed from top to bottom along the leftmost column, and then continue to the next columns from left to right. In this figure, the **Array size** value of [3, 2] creates an array having three rows and two columns.

**Size and Element Indexing Order  
for Uniform Rectangular Arrays**  
Example: Size = [3,2]

**Dependencies**

To enable this parameter, set **Geometry** to URA.

**Element lattice — Lattice of URA element positions**

Rectangular (default) | Triangular

Lattice of URA element positions, specified as Rectangular or Triangular.



- **Rectangular** — Aligns all the elements in row and column directions.
- **Triangular** — Shifts the even-row elements of a rectangular lattice toward the positive row-axis direction. The displacement is one-half the element spacing along the row dimension.

#### Dependencies

To enable this parameter, set **Geometry** to URA.

#### Array normal — Array normal direction

x for URA arrays or z for UCA arrays (default) | y

Array normal direction, specified as x, y, or z.

Elements of planar arrays lie in a plane orthogonal to the selected array normal direction. Element boresight directions point along the array normal direction.

Array Normal Parameter Value	Element Positions and Boresight Directions
x	Array elements lie in the yz-plane. All element boresight vectors point along the x-axis.
y	Array elements lie in the zx-plane. All element boresight vectors point along the y-axis.
z	Array elements lie in the xy-plane. All element boresight vectors point along the z-axis.

#### Dependencies

To enable this parameter, set **Geometry** to URA or UCA.

#### Radius of UCA (m) — UCA array radius

0.5 (default) | positive scalar

Radius of UCA array, specified as a positive scalar.

#### Dependencies

To enable this parameter, set **Geometry** to UCA.

#### Element positions (m) — Positions of conformal array elements

[0;0;0] (default) | 3-by- $N$  matrix of real values

Positions of the elements in a conformal array, specified as a 3-by- $N$  matrix of real values, where  $N$  is the number of elements in the conformal array. Each column of this matrix represents the position  $[x; y; z]$  of an array element in the array local coordinate system. The origin of the local coordinate system is  $(0,0,0)$ . Units are in meters.

When you set **Specify sensor array as** to `Replicated` subarray, this parameter applies to each subarray.

#### Dependencies

To enable this parameter set **Geometry** to `Conformal Array`.

#### Element normals (deg) — Direction of conformal array element normal vectors

[0;0] | 2-by-1 column vector | 2-by- $N$  matrix

Direction of element normal vectors in a conformal array, specified as a 2-by-1 column vector or a 2-by- $N$  matrix.  $N$  indicates the number of elements in the array. For a matrix, each column specifies the normal direction of the corresponding element in the form [azimuth;elevation] with respect to the local coordinate system. The local coordinate system aligns the positive  $x$ -axis with the direction normal to the conformal array. If the parameter value is a 2-by-1 column vector, the same pointing direction is used for all array elements.

When you set **Specify sensor array as** to `Replicated subarray`, this parameter applies to each subarray.

You can use the **Element positions (m)** and **Element normals (deg)** parameters to represent any arrangement in which pairs of elements differ by certain transformations. The transformations can combine translation, azimuth rotation, and elevation rotation. However, you cannot use transformations that require rotation about the normal direction.

### Dependencies

To enable this parameter, set **Geometry** to `Conformal Array`.

### Taper — Array element tapers

1 (default) | complex-valued scalar | complex-valued row vector

Element tapering, specified as a complex-valued scalar or a complex-valued 1-by- $N$  row vector. In this vector,  $N$  represents the number of elements in the array.

Also known as element weights, tapers multiply the array element responses. Tapers modify both amplitude and phase of the response to reduce side lobes or steer the main response axis.

If **Taper** is a scalar, the same weight is applied to each element. If **Taper** is a vector, a weight from the vector is applied to the corresponding sensor element. The number of weights must match the number of elements of the array.

When you set **Specify sensor array as** to `Replicated subarray`, this parameter applies to each subarray.

### Subarray definition matrix — Define elements belonging to subarrays

logical matrix

Specify the subarray selection as an  $M$ -by- $N$  matrix.  $M$  is the number of subarrays and  $N$  is the total number of elements in the array. Each row of the matrix represents a subarray and each entry in the row indicates when an element belongs to the subarray. When the entry is zero, the element does not belong the subarray. A nonzero entry represents a complex-valued weight applied to the corresponding element. Each row must contain at least one nonzero entry.

The phase center of each subarray lies at the subarray geometric center. The subarray geometric center depends on the **Subarray definition matrix** and **Geometry** parameters.

### Dependencies

To enable this parameter, set **Specify sensor array as** to `Partitioned array`.

### Subarray steering method — Specify subarray steering method

None (default) | Phase | Time

Subarray steering method, specified as one of

- None
- Phase
- Time
- Custom

Selecting **Phase** or **Time** opens the **Steer** input port on the Narrowband Receive Array, Narrowband Transmit Array, Wideband Receive Array, Wideband Transmit Array blocks, Constant Gamma Clutter, and GPU Constant Gamma Clutter blocks.

Selecting **Custom** opens the **WS** input port on the Narrowband Receive Array, Narrowband Transmit Array, Wideband Receive Array, Wideband Transmit Array blocks, Constant Gamma Clutter, and GPU Constant Gamma Clutter blocks.

#### Dependencies

To enable this parameter, set **Specify sensor array as** to **Partitioned array** or **Replicated subarray**.

#### Phase shifter frequency (Hz) — Subarray phase shifting frequency

3.0e8 (default) | positive real-valued scalar

Operating frequency of subarray steering phase shifters, specified as a positive real-valued scalar. Units are Hz.

#### Dependencies

To enable this parameter, set **Sensor array** to **Partitioned array** or **Replicated subarray** and set **Subarray steering method** to **Phase**.

#### Number of bits in phase shifters — Subarray steering phase shift quantization bits

0 (default) | non-negative integer

Subarray steering phase shift quantization bits, specified as a non-negative integer. A value of zero indicates that no quantization is performed.

#### Dependencies

To enable this parameter, set **Sensor array** to **Partitioned array** or **Replicated subarray** and set **Subarray steering method** to **Phase**.

#### Subarrays layout — Subarray position specification

Rectangular (default) | Custom

Specify the layout of replicated subarrays as **Rectangular** or **Custom**.

- When you set this parameter to **Rectangular**, use the **Grid size** and **Grid spacing** parameters to place the subarrays.
- When you set this parameter to **Custom**, use the **Subarray positions (m)** and **Subarray normals** parameters to place the subarrays.

#### Dependencies

To enable this parameter, set **Sensor array** to **Replicated subarray**

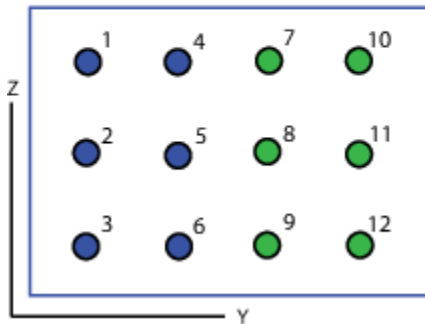
#### Grid size — Dimensions of rectangular subarray grid

[1, 2] (default)

Rectangular subarray grid size, specified as a single positive integer, or a 1-by-2 row vector of positive integers.

If **Grid size** is an integer scalar, the array has an equal number of subarrays in each row and column. If **Grid size** is a 1-by-2 vector of the form `[NumberOfRows, NumberOfColumns]`, the first entry is the number of subarrays along each column. The second entry is the number of subarrays in each row. A row is along the local y-axis, and a column is along the local z-axis. The figure here shows how you can replicate a 3-by-2 URA subarray using a **Grid size** of `[1, 2]`.

3 x 2 Element URA  
Replicated on a 1 x 2 Grid



#### Dependencies

To enable this parameter, set **Sensor array** to `Replicated` subarray and **Subarrays layout** to `Rectangular`.

#### Grid spacing (m) — Spacing between subarrays on rectangular grid

Auto (default) | positive real-valued scalar | 1-by-2 vector of positive real-values

The rectangular grid spacing of subarrays, specified as a positive, real-valued scalar, a 1-by-2 row vector of positive, real-values, or `Auto`. Units are in meters.

- If **Grid spacing** is a scalar, the spacing along the row and the spacing along the column is the same.
- If **Grid spacing** is a 1-by-2 row vector, the vector has the form `[SpacingBetweenRows, SpacingBetweenColumn]`. The first entry specifies the spacing between rows along a column. The second entry specifies the spacing between columns along a row.
- If **Grid spacing** is set to `Auto`, replication preserves the element spacing of the subarray for both rows and columns while building the full array. This option is available only when you specify **Geometry** as `ULA` or `URA`.

#### Dependencies

To enable this parameter, set **Sensor array** to `Replicated` subarray and **Subarrays layout** to `Rectangular`.

#### Subarray positions (m) — Positions of subarrays

`[0, 0; 0.5, 0.5; 0, 0]` (default) | 3-by-*N* real-valued matrix

Positions of the subarrays in the custom grid, specified as a real 3-by-*N* matrix, where *N* is the number of subarrays in the array. Each column of the matrix represents the position of a single

subarray in the array local coordinate system. The coordinates are expressed in the form  $[x; y; z]$ . Units are in meters.

#### Dependencies

To enable this parameter, set **Sensor array** to Replicated subarray and **Subarrays layout** to Custom.

#### Subarray normals — Direction of subarray normal vectors

$[0, 0; 0, 0]$  (default) | 2-by- $N$  real matrix

Specify the normal directions of the subarrays in the array. This parameter value is a 2-by- $N$  matrix, where  $N$  is the number of subarrays in the array. Each column of the matrix specifies the normal direction of the corresponding subarray, in the form  $[\text{azimuth}; \text{elevation}]$ . Angle units are in degrees. Angles are defined with respect to the local coordinate system.

You can use the **Subarray positions** and **Subarray normals** parameters to represent any arrangement in which pairs of subarrays differ by certain transformations. The transformations can combine translation, azimuth rotation, and elevation rotation. However, you cannot use transformations that require rotation about the normal.

#### Dependencies

To enable this parameter, set the **Sensor array** parameter to Replicated subarray and the **Subarrays layout** to Custom.

## Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### See Also

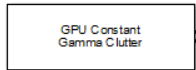
[constantGammaClutter](#) | [gpuConstantGammaClutter](#) | GPU Constant Gamma Clutter

**Introduced in R2021a**

## GPU Constant Gamma Clutter

Constant gamma clutter simulation using gpu

**Library:** Radar Toolbox



### Description

The GPU Constant Gamma Clutter block generates, using a graphical processing unit (GPU), constant gamma clutter reflected from a homogeneous terrain for a monostatic radar transmitting a narrowband signal into free space. The radar is assumed to be at a constant altitude moving at a constant speed.

### Ports

#### Input

##### PRFIdx — PRF Index

positive integer

Index to select the pulse repetition frequency (PRF), specified as a positive integer. The index selects the PRF from the predefined vector of values specified by the **Pulse repetition frequency (Hz)** parameter.

Example: 4

#### Dependencies

To enable this port, select **Enable PRF selection input**.

Data Types: double

##### W — Element weights

length- $N$  complex-valued vector

Weights applied to each element in array, specified as a length- $N$  complex-valued vector.  $N$  is the number of elements in the array selected in the **Sensor array** panel.

#### Dependencies

To enable this port, select the **Enable weights input** check box.

Data Types: double

##### WS — Subarray element weights

$N_E$ -by- $N_S$  complex-valued matrix

Weights applied to each element in a subarray, specified as an  $N_E$ -by- $N_S$  complex-valued matrix.

- When you set **Specify sensor array** to Replicated Subarray, all subarrays have the same dimensions. Then, you can specify the subarray element weights as a complex-valued  $N_E$ -by- $N_S$

matrix.  $N_E$  is the number of elements in each subarray and  $N_S$  is the number of subarrays. Each column of **WS** specifies the weights for the corresponding subarray.

- When you set **Specify sensor array** to **Partitioned array**, subarrays are not required to have identical dimensions and sizes. You can specify subarray element weights as a complex-valued  $N_E$ -by- $N_S$  matrix, where  $N_E$  now is the number of elements in the largest subarray. The first  $K$  entries in each column are the element weights for the corresponding subarray where  $K$  is the number of elements in the subarray.

### Dependencies

To enable this port, set **Specify sensor array** to **Partitioned array** or **Replicated Subarray**. Then, set **Subarray steering method** to **Custom**.

Data Types: `double`

### Steer — Steering angle input

scalar | 2-by-1 real-valued vector

Steering angle, specified as a scalar or a 2-by-1 real-valued vector. As a vector, the steering angle takes the form of `[AzimuthAngle; ElevationAngle]`. As a scalar, the steering angle represents the azimuth angle only. Then the elevation angle is assumed to be zero degrees. Units are in degrees

### Dependencies

To enable this port, set **Specify sensor array** to **Partitioned array** or **Replicated Subarray**. Then, set **Subarray steering method** to **Phase** or **Time**.

Data Types: `double`

### Output

#### Out — Simulated clutter

$N$ -by- $M$  complex-valued matrix

Simulated clutter, returned as an  $N$ -by- $M$  complex-valued matrix.

$N$  is the number of samples output from the block. When you set the **Output signal format** parameter to **Samples**, specify  $N$  using the **Number of samples in output** parameter. When you set the **Output signal format** parameter to **Pulses**,  $N$  is the total number of samples in the next  $P$  pulses where  $P$  is specified in the **Number of pulse in output** parameter.

$M$  is either

- the number of subarrays in the sensor array if sensor array contains subarrays.
- the number of radiating or collecting elements if the sensor array does not contain subarrays.

Data Types: `double`

## Parameters

### Main Tab

#### Terrain gamma value (dB) — Clutter model parameter

0 (default) | scalar

Clutter model parameter, specified as a scalar. This parameter contains the  $\gamma$  value used in the constant  $\gamma$  clutter model. The  $\gamma$  value depends on both terrain type and the operating frequency. Units are in dB.

Example: -5.0

Data Types: double

### **Earth model – Earth shape**

Flat (default) | Curved

Specify the earth model used in clutter simulation as Flat or Curved. When you set this parameter to Flat, the earth is assumed to be a plane. When you set this parameter to Curved, the earth is assumed to be spherical.

### **Minimum range of clutter region (m) – Minimum range of clutter region**

0 | nonnegative scalar

Specify the minimum range for the clutter simulation as a positive scalar. The minimum range must be nonnegative. Units are in meters.

### **Maximum range of clutter region (m) – Maximum range of clutter region**

5000 | nonnegative scalar

Specify the maximum range for the clutter simulation as a positive scalar. The maximum range must be greater than the value specified in the Radar height parameter. Units are in meters.

### **Azimuth center of clutter region (deg) – Azimuth center of clutter region**

0 | scalar

The azimuth angle in the ground plane about which clutter patches are generated. Patches are generated symmetrically about this angle. Units are in degrees.

### **Azimuth span of clutter region (deg) – Azimuth span of clutter region**

60 (default) | positive scalar

Specify the azimuth span of each clutter patch as a positive scalar. Units are in degrees. Units are in degrees.

### **Azimuth span of clutter patches (deg) – Azimuth span of clutter patches**

1 (default) | positive scalar

Azimuth span of each clutter patch, specified as a positive scalar. Units are in degrees.

Data Types: double

### **Clutter coherence time (s) – Coherence time of clutter simulation**

Inf (default) | positive scalar

Coherence time for the clutter simulation, specified as a positive scalar. After the coherence time elapses, the block updates the random numbers it uses for the clutter simulation at the next pulse. When you use the default value of Inf, the random numbers are never updated. Units are in seconds.

Example: 4

Data Types: double



**Signal propagation speed (m/s) – Signal propagation speed**

physconst('LightSpeed') (default) | real-valued positive scalar

Signal propagation speed, specified as a real-valued positive scalar. The default value of the speed of light is the value returned by `physconst('LightSpeed')`. Units are in meters per second.

Example: `3e8`

Data Types: `double`

**Sample rate (Hz) – Clutter sample rate**

`1e6` (default) | positive scalar

Clutter sample rate, specified as a positive scalar. Units are in Hertz.

Example: `10e6`

Data Types: `double`

**Pulse repetition frequency (Hz) – Pulse repetition frequency**

`1e4` (default) | positive scalar | row vector of positive values

Pulse repetition frequency, PRF, specified as a positive scalar or a row vector of positive values. Units are in Hertz.

Example: `[1e4, 2e4]`

Data Types: `double`

**Enable PRF selection input – Select predefined PRF**

`off` (default) | `on`

Select this parameter to enable the PRFIdx port.

- When enabled, pass in an index into a vector of predefined PRFs. Set predefined PRFs using the **Pulse repetition frequency (Hz)** parameter.
- When not enabled, the block cycles through the vector of PRFs specified by the **Pulse repetition frequency (Hz)** parameter. If **Pulse repetition frequency (Hz)** is a scalar, the PRF is constant.

**Output signal format – Format of the output signal**

`Pulses` (default) | `Samples`

The format of the output signal, specified as `Pulses` or `Samples`.

If you set this parameter to `Samples`, the output of the block consists of multiple samples. The number of samples is the value of the **Number of samples in output** parameter.

If you set this parameter to `Pulses`, the output of the block consists of multiple pulses. The number of pulses is the value of the **Number of pulses in output** parameter.

**Number of samples in output – Number of samples in output**

`100` (default) | positive integer

Number of samples in the block output, specified as a positive integer.

Example: `1000`

**Dependencies**

To enable this parameter, set the **Output signal format** parameter to `Samples`.

Data Types: `double`

**Number of pulses in output — Number of pulses in output**

1 (default) | positive integer

Number of pulses in the block output, specified as a positive integer.

Example: 2

**Dependencies**

To enable this parameter, set the **Output signal format** parameter to `Pulses`.

Data Types: `double`

**Simulate using — Block simulation method**

Interpreted Execution (default) | Code Generation

Block simulation, specified as `Interpreted Execution` or `Code Generation`. If you want your block to use the MATLAB interpreter, choose `Interpreted Execution`. If you want your block to run as compiled code, choose `Code Generation`. Compiled code requires time to compile but usually runs faster.

Interpreted execution is useful when you are developing and tuning a model. The block runs the underlying System object in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using `Code Generation`. Long simulations run faster with generated code than in interpreted execution. You can run repeated executions without recompiling, but if you change any block parameters, then the block automatically recompiles before execution.

This table shows how the **Simulate using** parameter affects the overall simulation behavior.

When the Simulink model is in Accelerator mode, the block mode specified using **Simulate using** overrides the simulation mode.

**Acceleration Modes**

Block Simulation	Simulation Behavior		
	Normal	Accelerator	Rapid Accelerator
Interpreted Execution	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
Code Generation	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink).

**Radar Tab****Operating frequency (Hz) — System operating frequency**

3.0e8 (default) | positive real scalar

System operating frequency, specified as a positive scalar. Units are in Hz.

**Effective transmitted power (W) — radar system effective transmitted power**

5000 (default) | positive scalar

Effective radiated power (ERP) of the radar system, specified as a positive scalar. Units are in watts.

Example: 3500

Data Types: double

**Radar height (m) — Height of radar above surface**

0 (default) | nonnegative scalar

Height of radar above surface, specified as a nonnegative scalar. Units are in meters.

Example: 50

Data Types: double

**Radar speed (m/s) — Radar platform speed**

0 (default) | nonnegative scalar

Radar platform speed, specified as a nonnegative scalar. Units are in meters per second.

Example: 5

Data Types: double

**Radar motion direction (deg) — Direction of motion of radar platform**

[90;0] (default) | 2-by-1 real vector

Specify the direction of radar platform motion as a 2-by-1 real vector in the form [AzimuthAngle;ElevationAngle]. Units are in degrees. Both azimuth and elevation angle are measured in the local coordinate system of the radar antenna or antenna array. Azimuth angle must be between  $-180^\circ$  and  $180^\circ$ . Elevation angle must be between  $-90^\circ$  and  $90^\circ$ .

The default value of this parameter indicates that the radar platform is moving perpendicular to the radar antenna array broadside direction.

Example: [25;30]

Data Types: double

**Sensor mounting angles sensor (deg) — Sensor mounting angles**

[0 0 0] (default) | length-3 vector of positive values

Specify a 3-element vector that gives the intrinsic yaw, pitch, and roll of the sensor frame from the inertial frame. The 3 elements define the rotations around the z, y, and x axes respectively, in that order. The first rotation, rotates the body axes around the z-axis. Because these angles define intrinsic rotations, the second rotation is performed around the y-axis in its new position resulting from the previous rotation. The final rotation around the x-axis is performed around the x-axis as rotated by the first two rotations in the intrinsic system.

Example: [0, -10, 4]

Data Types: double

**Enable weights input — Enable antenna element weights input port**

unchecked (default) | checked

Check box to enable antenna element weights input port,  $W$ .

### Sensor Array Tab

#### Specify sensor array as — Method to specify array

Array (no subarrays) (default) | Partitioned array | Replicated subarray | MATLAB expression

Method to specify array, specified as Array (no subarrays) or MATLAB expression.

- Array (no subarrays) — use the block parameters to specify the array.
- Partitioned array — use the block parameters to specify the array.
- Replicated subarray — use the block parameters to specify the array.
- MATLAB expression — create the array using a MATLAB expression.

#### Expression — MATLAB expression used to create an array

Phased Array System Toolbox array System object

MATLAB expression used to create an array, specified as a valid Phased Array System Toolbox array System object.

Example: `phased.URA('Size',[5,3])`

#### Dependencies

To enable this parameter, set **Specify sensor array as** to MATLAB expression.

#### Element Parameters

#### Element type — Array element types

Isotropic Antenna (default) | Cosine Antenna | Custom Antenna | Omni Microphone | Custom Microphone

Antenna or microphone type, specified as one of the following:

- Isotropic Antenna
- Cosine Antenna
- Custom Antenna
- Omni Microphone
- Custom Microphone

#### Operating frequency range (Hz) — Operating frequency range of the antenna or microphone element

[0,1.0e20] (default) | real-valued 1-by-2 row vector

Specify the operating frequency range of the antenna or microphone element as a 1-by-2 row vector in the form [LowerBound,UpperBound]. The element has no response outside this frequency range. Frequency units are in Hz.

#### Dependencies

To enable this parameter, set **Element type** to Isotropic Antenna, Cosine Antenna, or Omni Microphone.

**Operating frequency vector (Hz) — Operating frequency range of custom antenna or microphone elements**

[0, 1.0e20] (default) | real-valued row vector

Specify the frequencies at which to set antenna and microphone frequency responses as a 1-by- $L$  row vector of increasing real values. The antenna or microphone element has no response outside the frequency range specified by the minimum and maximum elements of this vector. Frequency units are in Hz.

**Dependencies**

To enable this parameter, set **Element type** to Custom Antenna or Custom Microphone. Use **Frequency responses (dB)** to set the responses at these frequencies.

**Baffle the back of the element — Set back response of an Isotropic Antenna element or an Omni Microphone element to zero**

off (default) | on

Select this check box to baffle the back response of the element. When back baffled, the responses at all azimuth angles beyond  $\pm 90^\circ$  from broadside are set to zero. The broadside direction is defined as  $0^\circ$  azimuth angle and  $0^\circ$  elevation angle.

**Dependencies**

To enable this check box, set **Element type** to Isotropic Antenna or Omni Microphone.

**Exponent of cosine pattern — Exponents of azimuth and elevation cosine patterns**

[1.5 1.5] (default) | nonnegative scalar | real-valued 1-by-2 matrix of nonnegative values

Specify the exponents of the cosine pattern as a nonnegative scalar or a real-valued 1-by-2 matrix of nonnegative values. When **Exponent of cosine pattern** is a 1-by-2 vector, the first element is the exponent in the azimuth direction and the second element is the exponent in the elevation direction. When you set this parameter to a scalar, both the azimuth direction and elevation direction cosine patterns are raised to the same power.

**Dependencies**

To enable this parameter, set **Element type** to Cosine Antenna.

**Frequency responses (dB) — Antenna and microphone frequency response**

[0, 0] (default) | real-valued row vector

Frequency response of a custom antenna or custom microphone for the frequencies defined by the **Operating frequency vector (Hz)** parameter. The dimensions of **Frequency responses (dB)** must match the dimensions of the vector specified by the **Operating frequency vector (Hz)** parameter.

**Dependencies**

To enable this parameter, set **Element type** to Custom Antenna or Custom Microphone.

**Input Pattern Coordinate System — Coordinate system of custom antenna pattern**

az-el (default) | phi-theta

Coordinate system of custom antenna pattern, specified az-el or phi-theta. When you specify az-el, use the **Azimuth angles (deg)** and **Elevations angles (deg)** parameters to specify the coordinates of the pattern points. When you specify phi-theta, use the **Phi angles (deg)** and **Theta angles (deg)** parameters to specify the coordinates of the pattern points.

**Dependencies**

To enable this parameter, set **Element type** to Custom Antenna.

**Azimuth angles (deg) — Azimuth angles of antenna radiation pattern**

`[-180:180]` (default) | real-valued row vector

Specify the azimuth angles at which to calculate the antenna radiation pattern as a 1-by- $P$  row vector.  $P$  must be greater than 2. Azimuth angles must lie between  $-180^\circ$  and  $180^\circ$ , inclusive, and be in strictly increasing order.

**Dependencies**

To enable this parameter, set the **Element type** parameter to Custom Antenna and the **Input Pattern Coordinate System** parameter to az-el.

**Elevation angles (deg) — Elevation angles of antenna radiation pattern**

`[-90:90]` (default) | real-valued row vector

Specify the elevation angles at which to compute the radiation pattern as a 1-by- $Q$  vector.  $Q$  must be greater than 2. Angle units are in degrees. Elevation angles must lie between  $-90^\circ$  and  $90^\circ$ , inclusive, and be in strictly increasing order.

**Dependencies**

To enable this parameter, set the **Element type** parameter to Custom Antenna and the **Input Pattern Coordinate System** parameter to az-el.

**Phi Angles (deg) — Phi angle coordinates of custom antenna radiation pattern**

`0:360` | real-valued 1-by- $P$  row vector

Phi angles of points at which to specify the antenna radiation pattern, specify as a real-valued 1-by- $P$  row vector.  $P$  must be greater than 2. Angle units are in degrees. Phi angles must lie between  $0^\circ$  and  $360^\circ$  and be in strictly increasing order.

**Dependencies**

To enable this parameter, set the **Element type** parameter to Custom Antenna and the **Input Pattern Coordinate System** parameter to phi-theta.

**Theta Angles (deg) — Theta angle coordinates of custom antenna radiation pattern**

`0:180` | real-valued 1-by- $Q$  row vector

Theta angles of points at which to specify the antenna radiation pattern, specify as a real-valued 1-by- $Q$  row vector.  $Q$  must be greater than 2. Angle units are in degrees. Theta angles must lie between  $0^\circ$  and  $360^\circ$  and be in strictly increasing order.

**Dependencies**

To enable this parameter, set the **Element type** parameter to Custom Antenna and the **Input Pattern Coordinate System** parameter to phi-theta.

**Magnitude pattern (dB) — Magnitude of combined antenna radiation pattern**

`zeros(181,361)` (default) | real-valued  $Q$ -by- $P$  matrix | real-valued  $Q$ -by- $P$ -by- $L$  array

Magnitude of the combined antenna radiation pattern, specified as a  $Q$ -by- $P$  matrix or a  $Q$ -by- $P$ -by- $L$  array.

- When the **Input Pattern Coordinate System** parameter is set to `az-e $\ell$` ,  $Q$  equals the length of the vector specified by the **Elevation angles (deg)** parameter and  $P$  equals the length of the vector specified by the **Azimuth angles (deg)** parameter.
- When the **Input Pattern Coordinate System** parameter is set to `phi-theta`,  $Q$  equals the length of the vector specified by the **Theta Angles (deg)** parameter and  $P$  equals the length of the vector specified by the **Phi Angles (deg)** parameter.

The quantity  $L$  equals the length of the **Operating frequency vector (Hz)**.

- If this parameter is a  $Q$ -by- $P$  matrix, the same pattern is applied to *all* frequencies specified in the **Operating frequency vector (Hz)** parameter.
- If the value is a  $Q$ -by- $P$ -by- $L$  array, each  $Q$ -by- $P$  page of the array specifies a pattern for the *corresponding* frequency specified in the **Operating frequency vector (Hz)** parameter.

#### Dependencies

To enable this parameter, set **Element type** to Custom Antenna.

#### Phase pattern (deg) — Custom antenna radiation phase pattern

`zeros(181, 361)` (default) | real-valued  $Q$ -by- $P$  matrix | real-valued  $Q$ -by- $P$ -by- $L$  array

Phase of the combined antenna radiation pattern, specified as a  $Q$ -by- $P$  matrix or a  $Q$ -by- $P$ -by- $L$  array.

- When the **Input Pattern Coordinate System** parameter is set to `az-e $\ell$` ,  $Q$  equals the length of the vector specified by the **Elevation angles (deg)** parameter and  $P$  equals the length of the vector specified by the **Azimuth angles (deg)** parameter.
- When the **Input Pattern Coordinate System** parameter is set to `phi-theta`,  $Q$  equals the length of the vector specified by the **Theta Angles (deg)** parameter and  $P$  equals the length of the vector specified by the **Phi Angles (deg)** parameter.

The quantity  $L$  equals the length of the **Operating frequency vector (Hz)**.

- If this parameter is a  $Q$ -by- $P$  matrix, the same pattern is applied to *all* frequencies specified in the **Operating frequency vector (Hz)** parameter.
- If the value is a  $Q$ -by- $P$ -by- $L$  array, each  $Q$ -by- $P$  page of the array specifies a pattern for the *corresponding* frequency specified in the **Operating frequency vector (**

#### Dependencies

To enable this parameter, set **Element type** to Custom Antenna.

#### MatchArrayNormal — Rotate antenna element to array normal

`on` (default) | `off`

Select this check box to rotate the antenna element pattern to align with the array normal. When not selected, the element pattern is not rotated.

When the antenna is used in an antenna array and the **Input Pattern Coordinate System** parameter is `az-e $\ell$` , selecting this check box rotates the pattern so that the  $x$ -axis of the element coordinate system points along the array normal. Not selecting uses the element pattern without the rotation.

When the antenna is used in an antenna array and **Input Pattern Coordinate System** is set to phi-theta, selecting this check box rotates the pattern so that the z-axis of the element coordinate system points along the array normal.

Use the parameter in conjunction with the **Array normal** parameter of the URA and UCA arrays.

#### Dependencies

To enable this parameter, set **Element type** to Custom Antenna.

**Polar pattern frequencies (Hz) — Polar pattern microphone response frequencies**  
1e3 (default) | real scalar | real-valued 1-by- $L$  row vector

Polar pattern microphone response frequencies, specified as a real scalar, or a real-valued, 1-by- $L$  vector. The response frequencies lie within the frequency range specified by the **Operating frequency vector (Hz)** vector.

#### Dependencies

To enable this parameter, set **Element type** set to Custom Microphone.

**Polar pattern angles (deg) — Polar pattern response angles**  
[-180:180] (default) | real-valued -by- $P$  row vector

Specify the polar pattern response angles, as a 1-by- $P$  vector. The angles are measured from the central pickup axis of the microphone and must be between  $-180^\circ$  and  $180^\circ$ , inclusive.

#### Dependencies

To enable this parameter, set **Element type** to Custom Microphone.

**Polar pattern (dB) — Custom microphone polar response**  
zeros(1,361) (default) | real-valued  $L$ -by- $P$  matrix

Specify the magnitude of the custom microphone element polar patterns as an  $L$ -by- $P$  matrix.  $L$  is the number of frequencies specified in **Polar pattern frequencies (Hz)**.  $P$  is the number of angles specified in **Polar pattern angles (deg)**. Each row of the matrix represents the magnitude of the polar pattern measured at the corresponding frequency specified in **Polar pattern frequencies (Hz)** and all angles specified in **Polar pattern angles (deg)**. The pattern is measured in the azimuth plane. In the azimuth plane, the elevation angle is  $0^\circ$  and the central pickup axis is  $0^\circ$  degrees azimuth and  $0^\circ$  degrees elevation. The polar pattern is symmetric around the central axis. You can construct the microphone response pattern in 3-D space from the polar pattern.

#### Dependencies

To enable this parameter, set **Element type** to Custom Microphone.

#### Array Parameters

**Geometry — Array geometry**  
ULA (default) | URA | UCA | Conformal Array

Array geometry, specified as one of

- ULA — Uniform linear array
- URA — Uniform rectangular array



- UCA — Uniform circular array
- Conformal Array — arbitrary element positions

### Number of elements — Number of array elements

2 for ULA arrays and 5 for UCA arrays (default) | integer greater than or equal to 2

The number of array elements for ULA or UCA arrays, specified as an integer greater than or equal to 2.

When you set **Specify sensor array as** to `Replicated subarray`, this parameter applies to each subarray.

#### Dependencies

To enable this parameter, set **Geometry** to ULA or UCA.

### Element spacing (m) — Spacing between array elements

0.5 for ULA arrays and [0.5, 0.5] for URA arrays (default) | positive scalar for ULA or URA arrays | 2-element vector of positive values for URA arrays

Spacing between adjacent array elements:

- ULA — specify the spacing between two adjacent elements in the array as a positive scalar.
- URA — specify the spacing as a positive scalar or a 1-by-2 vector of positive values. If **Element spacing (m)** is a scalar, the row and column spacings are equal. If **Element spacing (m)** is a vector, the vector has the form `[SpacingBetweenArrayRows, SpacingBetweenArrayColumns]`.
- When you set **Specify sensor array as** to `Replicated subarray`, this parameter applies to each subarray.

#### Dependencies

To enable this parameter, set **Geometry** to ULA or URA.

### Array axis — Linear axis direction of ULA

y (default) | x | z

Linear axis direction of ULA, specified as y, x, or z. All ULA array elements are uniformly spaced along this axis in the local array coordinate system.

#### Dependencies

- To enable this parameter, set **Geometry** to ULA.
- This parameter is also enabled when the block only supports ULA arrays.

### Array size — Dimensions of URA array

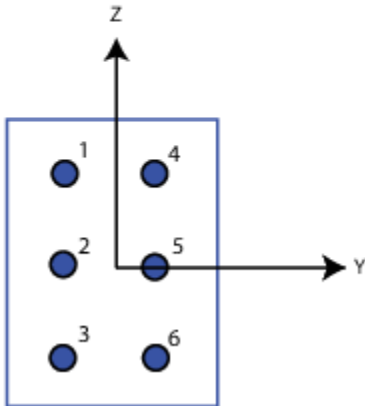
[2, 2] (default) | positive integer | 1-by-2 vector of positive integers

Dimensions of a URA array, specified as a positive integer or 1-by-2 vector of positive integers.

- If **Array size** is a 1-by-2 vector, the vector has the form `[NumberOfArrayRows, NumberOfArrayColumns]`.
- If **Array size** is an integer, the array has the same number of rows and columns.
- When you set **Specify sensor array as** to `Replicated subarray`, this parameter applies to each subarray.

For a URA, array elements are indexed from top to bottom along the leftmost column, and then continue to the next columns from left to right. In this figure, the **Array size** value of [3, 2] creates an array having three rows and two columns.

**Size and Element Indexing Order  
for Uniform Rectangular Arrays**  
Example: Size = [3,2]



**Dependencies**

To enable this parameter, set **Geometry** to URA.

**Element lattice – Lattice of URA element positions**

Rectangular (default) | Triangular

Lattice of URA element positions, specified as Rectangular or Triangular.

- Rectangular — Aligns all the elements in row and column directions.
- Triangular — Shifts the even-row elements of a rectangular lattice toward the positive row-axis direction. The displacement is one-half the element spacing along the row dimension.

**Dependencies**

To enable this parameter, set **Geometry** to URA.

**Array normal – Array normal direction**

x for URA arrays or z for UCA arrays (default) | y

Array normal direction, specified as x, y, or z.

Elements of planar arrays lie in a plane orthogonal to the selected array normal direction. Element boresight directions point along the array normal direction.

Array Normal Parameter Value	Element Positions and Boresight Directions
x	Array elements lie in the yz-plane. All element boresight vectors point along the x-axis.

Array Normal Parameter Value	Element Positions and Boresight Directions
y	Array elements lie in the $zx$ -plane. All element boresight vectors point along the $y$ -axis.
z	Array elements lie in the $xy$ -plane. All element boresight vectors point along the $z$ -axis.

### Dependencies

To enable this parameter, set **Geometry** to URA or UCA.

### Radius of UCA (m) – UCA array radius

0.5 (default) | positive scalar

Radius of UCA array, specified as a positive scalar.

### Dependencies

To enable this parameter, set **Geometry** to UCA.

### Element positions (m) – Positions of conformal array elements

[0;0;0] (default) | 3-by- $N$  matrix of real values

Positions of the elements in a conformal array, specified as a 3-by- $N$  matrix of real values, where  $N$  is the number of elements in the conformal array. Each column of this matrix represents the position  $[x; y; z]$  of an array element in the array local coordinate system. The origin of the local coordinate system is  $(0,0,0)$ . Units are in meters.

When you set **Specify sensor array as** to `Replicated` subarray, this parameter applies to each subarray.

### Dependencies

To enable this parameter set **Geometry** to `Conformal Array`.

### Element normals (deg) – Direction of conformal array element normal vectors

[0;0] | 2-by-1 column vector | 2-by- $N$  matrix

Direction of element normal vectors in a conformal array, specified as a 2-by-1 column vector or a 2-by- $N$  matrix.  $N$  indicates the number of elements in the array. For a matrix, each column specifies the normal direction of the corresponding element in the form  $[\text{azimuth}; \text{elevation}]$  with respect to the local coordinate system. The local coordinate system aligns the positive  $x$ -axis with the direction normal to the conformal array. If the parameter value is a 2-by-1 column vector, the same pointing direction is used for all array elements.

When you set **Specify sensor array as** to `Replicated` subarray, this parameter applies to each subarray.

You can use the **Element positions (m)** and **Element normals (deg)** parameters to represent any arrangement in which pairs of elements differ by certain transformations. The transformations can combine translation, azimuth rotation, and elevation rotation. However, you cannot use transformations that require rotation about the normal direction.

### Dependencies

To enable this parameter, set **Geometry** to `Conformal Array`.

**Taper – Array element tapers**

1 (default) | complex-valued scalar | complex-valued row vector

Element tapering, specified as a complex-valued scalar or a complex-valued 1-by- $N$  row vector. In this vector,  $N$  represents the number of elements in the array.

Also known as element weights, tapers multiply the array element responses. Tapers modify both amplitude and phase of the response to reduce side lobes or steer the main response axis.

If **Taper** is a scalar, the same weight is applied to each element. If **Taper** is a vector, a weight from the vector is applied to the corresponding sensor element. The number of weights must match the number of elements of the array.

When you set **Specify sensor array as** to `Replicated subarray`, this parameter applies to each subarray.

**Subarray definition matrix – Define elements belonging to subarrays**

logical matrix

Specify the subarray selection as an  $M$ -by- $N$  matrix.  $M$  is the number of subarrays and  $N$  is the total number of elements in the array. Each row of the matrix represents a subarray and each entry in the row indicates when an element belongs to the subarray. When the entry is zero, the element does not belong the subarray. A nonzero entry represents a complex-valued weight applied to the corresponding element. Each row must contain at least one nonzero entry.

The phase center of each subarray lies at the subarray geometric center. The subarray geometric center depends on the **Subarray definition matrix** and **Geometry** parameters.

**Dependencies**

To enable this parameter, set **Specify sensor array as** to `Partitioned array`.

**Subarray steering method – Specify subarray steering method**

None (default) | Phase | Time

Subarray steering method, specified as one of

- None
- Phase
- Time
- Custom

Selecting `Phase` or `Time` opens the `Steer` input port on the Narrowband Receive Array, Narrowband Transmit Array, Wideband Receive Array, Wideband Transmit Array blocks, Constant Gamma Clutter, and GPU Constant Gamma Clutter blocks.

Selecting `Custom` opens the `WS` input port on the Narrowband Receive Array, Narrowband Transmit Array, Wideband Receive Array, Wideband Transmit Array blocks, Constant Gamma Clutter, and GPU Constant Gamma Clutter blocks.

**Dependencies**

To enable this parameter, set **Specify sensor array as** to `Partitioned array` or `Replicated subarray`.

**Phase shifter frequency (Hz) — Subarray phase shifting frequency**

3.0e8 (default) | positive real-valued scalar

Operating frequency of subarray steering phase shifters, specified as a positive real-valued scalar. Units are Hz.

**Dependencies**

To enable this parameter, set **Sensor array** to `Partitioned array` or `Replicated subarray` and set **Subarray steering method** to `Phase`.

**Number of bits in phase shifters — Subarray steering phase shift quantization bits**

0 (default) | non-negative integer

Subarray steering phase shift quantization bits, specified as a non-negative integer. A value of zero indicates that no quantization is performed.

**Dependencies**

To enable this parameter, set **Sensor array** to `Partitioned array` or `Replicated subarray` and set **Subarray steering method** to `Phase`.

**Subarrays layout — Subarray position specification**

Rectangular (default) | Custom

Specify the layout of replicated subarrays as `Rectangular` or `Custom`.

- When you set this parameter to `Rectangular`, use the **Grid size** and **Grid spacing** parameters to place the subarrays.
- When you set this parameter to `Custom`, use the **Subarray positions (m)** and **Subarray normals** parameters to place the subarrays.

**Dependencies**

To enable this parameter, set **Sensor array** to `Replicated subarray`

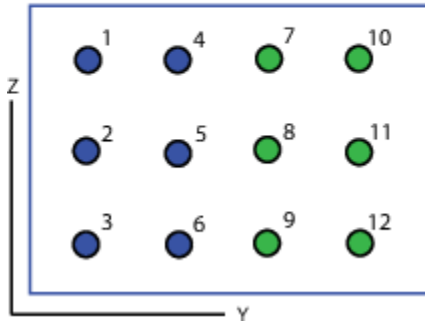
**Grid size — Dimensions of rectangular subarray grid**

[1, 2] (default)

Rectangular subarray grid size, specified as a single positive integer, or a 1-by-2 row vector of positive integers.

If **Grid size** is an integer scalar, the array has an equal number of subarrays in each row and column. If **Grid size** is a 1-by-2 vector of the form `[NumberOfRows, NumberOfColumns]`, the first entry is the number of subarrays along each column. The second entry is the number of subarrays in each row. A row is along the local *y*-axis, and a column is along the local *z*-axis. The figure here shows how you can replicate a 3-by-2 URA subarray using a **Grid size** of `[1, 2]`.

### 3 x 2 Element URA Replicated on a 1 x 2 Grid



#### Dependencies

To enable this parameter, set **Sensor array** to `Replicated` subarray and **Subarrays layout** to `Rectangular`.

#### Grid spacing (m) — Spacing between subarrays on rectangular grid

`Auto` (default) | positive real-valued scalar | 1-by-2 vector of positive real-values

The rectangular grid spacing of subarrays, specified as a positive, real-valued scalar, a 1-by-2 row vector of positive, real-values, or `Auto`. Units are in meters.

- If **Grid spacing** is a scalar, the spacing along the row and the spacing along the column is the same.
- If **Grid spacing** is a 1-by-2 row vector, the vector has the form `[SpacingBetweenRows, SpacingBetweenColumn]`. The first entry specifies the spacing between rows along a column. The second entry specifies the spacing between columns along a row.
- If **Grid spacing** is set to `Auto`, replication preserves the element spacing of the subarray for both rows and columns while building the full array. This option is available only when you specify **Geometry** as `ULA` or `URA`.

#### Dependencies

To enable this parameter, set **Sensor array** to `Replicated` subarray and **Subarrays layout** to `Rectangular`.

#### Subarray positions (m) — Positions of subarrays

`[0,0;0.5,0.5;0,0]` (default) | 3-by- $N$  real-valued matrix

Positions of the subarrays in the custom grid, specified as a real 3-by- $N$  matrix, where  $N$  is the number of subarrays in the array. Each column of the matrix represents the position of a single subarray in the array local coordinate system. The coordinates are expressed in the form `[x; y; z]`. Units are in meters.

#### Dependencies

To enable this parameter, set **Sensor array** to `Replicated` subarray and **Subarrays layout** to `Custom`.

**Subarray normals — Direction of subarray normal vectors**

[0, 0; 0, 0] (default) | 2-by- $N$  real matrix

Specify the normal directions of the subarrays in the array. This parameter value is a 2-by- $N$  matrix, where  $N$  is the number of subarrays in the array. Each column of the matrix specifies the normal direction of the corresponding subarray, in the form [azimuth; elevation]. Angle units are in degrees. Angles are defined with respect to the local coordinate system.

You can use the **Subarray positions** and **Subarray normals** parameters to represent any arrangement in which pairs of subarrays differ by certain transformations. The transformations can combine translation, azimuth rotation, and elevation rotation. However, you cannot use transformations that require rotation about the normal.

**Dependencies**

To enable this parameter, set the **Sensor array** parameter to Replicated subarray and the **Subarrays layout** to Custom.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**See Also**

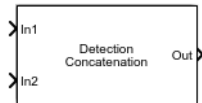
gpuConstantGammaClutter | constantGammaClutter | Constant Gamma Clutter

**Introduced in R2021a**

## Detection Concatenation

Combine detection reports from different sensors

**Library:** Automated Driving Toolbox  
Sensor Fusion and Tracking Toolbox / Utilities



### Description

The Detection Concatenation block combines detection reports from multiple sensors onto a single output bus. Concatenation is useful when detections from multiple sensor blocks are passed into a tracker block such as the block. You can accommodate additional sensors by changing the **Number of input sensors to combine** parameter to increase the number of input ports.

### Ports

#### Input

#### **In1, In2, ..., InN — Sensor detections to combine**

Simulink buses containing MATLAB structures

Sensor detections to combine, where each detection is a Simulink bus containing a MATLAB structure. See “Create Nonvirtual Buses” (Simulink) for more details.

The structure has the form:

Field	Description	Type
NumDetections	Number of detections	integer
Detections	Object detections	Array of object detection structures. The first NumDetections of these detections are actual detections.

The fields of Detections are:

Field	Description	Type
Time	Measurement time	single or double
Measurement	Object measurements	single or double
MeasurementNoise	Measurement noise covariance matrix	single or double
SensorIndex	Unique ID of the sensor	single or double
ObjectClassID	Object classification ID	single or double



Field	Description	Type
MeasurementParameters	Parameters used by initialization functions of tracking filters	Simulink Bus
ObjectAttributes	Additional information passed to tracker	Simulink Bus

By default, the block includes two ports for input detections. To add more ports, use the **Number of input sensors to combine** parameter.

## Output

### Out — Combined sensor detections

Simulink bus containing MATLAB structure

Combined sensor detections from all input buses, returned as a Simulink bus containing a MATLAB structure. See “Create Nonvirtual Buses” (Simulink).

The structure has the form:

Field	Description	Type
NumDetections	Number of detections	integer
Detections	Object detections	Array of object detection structures. The first NumDetections of these detections are actual detections.

The fields of Detections are:

Field	Description	Type
Time	Measurement time	single or double
Measurement	Object measurements	single or double
MeasurementNoise	Measurement noise covariance matrix	single or double
SensorIndex	Unique ID of the sensor	single or double
ObjectClassID	Object classification ID	single or double
MeasurementParameters	Parameters used by initialization functions of tracking filters	Simulink Bus
ObjectAttributes	Additional information passed to tracker	Simulink Bus

The **Maximum number of reported detections** output is the sum of the **Maximum number of reported detections** of all input ports. The number of actual detections is the sum of the number of actual detections in each input port. The **ObjectAttributes** fields in the detection structure are the union of the **ObjectAttributes** fields in each input port.

## Parameters

### Number of input sensors to combine — Number of input sensor ports

2 (default) | positive integer

Number of input sensor ports, specified as a positive integer. Each input port is labeled **In1**, **In2**, ..., **InN**, where *N* is the value set by this parameter.

Data Types: double

### Source of output bus name — Source of output bus name

Auto (default) | Property

Source of output bus name, specified as Auto or Property.

- If you select Auto, the block automatically generates a bus name.
- If you select Property, specify the bus name using the **Specify an output bus name** parameter.

### Specify an output bus name — Name of output bus

no default

## Dependencies

To enable this parameter, set the **Source of output bus name** parameter to Property.

### Simulate using — Type of simulation to run

Interpreted execution (default) | Code generation

- **Interpreted execution** — Simulate the model using the MATLAB interpreter. This option shortens startup time. In Interpreted execution mode, you can debug the source code of the block.
- **Code generation** — Simulate the model using generated C/C++ code. The first time you run a simulation, Simulink generates C/C++ code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

### Topics

“Create Nonvirtual Buses” (Simulink)

# Pulse Compression Library

Library of pulse compression specifications

**Library:** Radar Toolbox



## Description

The Pulse Compression Library block performs range processing using pulse compression. Pulse compression techniques include matched filtering and stretch processing. The block lets you create a library of different pulse compression specifications. The output is the filter response consisting of a matrix or a three-dimensional array with rows representing range gates.

## Ports

### Input

#### X — Input signal

complex-valued  $K$ -by- $L$  matrix | complex-valued  $K$ -by- $N$  matrix | complex-valued  $K$ -by- $N$ -by- $L$  array

Input signal, specified as a complex-valued  $K$ -by- $L$  matrix, complex-valued  $K$ -by- $N$  matrix, or a complex-valued  $K$ -by- $N$ -by- $L$  array.  $K$  denotes the number of fast time samples,  $L$  the number of pulses, and  $N$  is the number of channels. Channels can be array elements or beams.

Data Types: `double`

#### Idx — Index of processing specification

positive integer

Index of the processing specification in the pulse compression library, specified as a positive integer.

Data Types: `double`

### Output

#### Y — Output signal

complex-valued  $M$ -by- $L$  matrix | complex-valued  $M$ -by- $N$  matrix | complex-valued  $M$ -by- $N$ -by- $L$  array

Output signal, returned as a complex-valued  $M$ -by- $L$  matrix, complex-valued  $M$ -by- $N$  matrix, or a complex-valued  $M$ -by- $N$ -by- $L$  array.  $M$  denotes the number of fast time samples,  $L$  the number of pulses, and  $N$  is the number of channels. Channels can be array elements or beams. The number of dimensions of  $Y$  matches the number of dimensions of  $X$ .

When matched filtering is performed,  $M$  is equal to the number of rows in  $X$ . When stretch processing is performed and you specify a value for the `RangeFFTLlength` name-value pair,  $M$  is set to the value of `RangeFFTLlength`. When you do not specify `RangeFFTLlength`,  $M$  is equal to the number of rows in  $X$ .

Data Types: `double`

**Range — Sample range**real-valued length- $M$  vector

Sample ranges, returned as a real-valued length- $M$  vector where  $M$  is the number of rows of  $Y$ . Elements of this vector denote the ranges corresponding to the rows of  $Y$ .

Data Types: double

**Parameters****Signal propagation speed (m/s) — Signal propagation speed**

physconst('LightSpeed') (default) | real-valued positive scalar

Signal propagation speed, specified as a real-valued positive scalar. The default value of the speed of light is the value returned by physconst('LightSpeed'). Units are in meters per second.

Example: 3e8

Data Types: double

**Specification of each waveform in the library — Specification of pulse waveforms in the library**

```
{{'Rectangular','PRF',1e4,'PulseWidth',50e-6},
{'LinearFM','PRF',1e4,'PulseWidth',50e-6,'SweepBandwidth',1e5,'SweepDirection',
'Up','SweepInterval','Positive'}} (default) | cell array
```

Pulse waveforms, specified as a cell array. Each cell of the array contains the specification of one waveform. Each waveform specification is also a cell array containing the parameters of the waveform.

```
{{Waveform 1 Specification},{Waveform 2 Specification},{Waveform 3 Specification}, ...}
```

This block supports four built-in waveforms and also lets you specify custom waveforms. Each built-in waveform specifier consists of a waveform identifier followed by several name-value pairs that set the properties of the waveform.

**Built-in Waveforms**

Waveform type	Waveform identifier	Waveform name-value pair arguments
Linear FM	'LinearFM'	See “Linear FM Waveform Arguments” on page 4-379
Phase coded	'PhaseCoded'	See “Phase-Coded Waveform Arguments” on page 4-381
Rectangular	'Rectangular'	See “Rectangular Waveform Arguments” on page 4-382
Stepped FM	'SteppedFM'	See “Stepped FM Waveform Arguments” on page 4-398

You can create a custom waveform with a user-defined function. The first input argument of the function must be the sample rate. Use a function handle instead of the waveform identifier in the first cell of a waveform specification. The remaining cells contain all function input arguments except the sample rate. Specify all input arguments in the order they are passed into the function. The function must have at least one output argument to return the samples of each pulse in a column vector. You can only create custom waveforms when you set **Simulate using** to Interpreted Execution.

### Pulse compression specifications – Specify type of pulse compression

```
{{'MatchedFilter','SpectrumWindow','None'},
{'StretchProcessor','RangeSpan',200,'ReferenceRange',5e3,'RangeWindow','None'
}} (default) | cell array
```

Waveform processing type and parameters, specified as a cell array of processing specifications. Each processing specification is itself a cell array containing the processing type and processing arguments.

```
{{Processing 1 Specification},{Processing 2 Specification},{Processing 3 Specification}, ...}
```

Each processing specification indicates which type of processing to apply to a waveform and the arguments needed for processing.

```
{processtype,Name,Value,...}
```

The value of `processtype` is either `'MatchedFilter'` or `'StretchProcessor'`.

- `'MatchedFilter'` - The name-value pair arguments are
  - `'Coefficients',coeff` - specifies the matched filter coefficients, `coeff`, as a column vector. When not specified, the coefficients are calculated from the `WaveformSpecification` property. For the Stepped FM waveform containing multiple pulses, `coeff` corresponds to each pulse until the pulse index, `idx` changes.
  - `'SpectrumWindow',sw` - specifies the spectrum weighting window, `sw`, applied to the waveform. Window values are one of `'None'`, `'Hamming'`, `'Chebyshev'`, `'Hann'`, `'Kaiser'`, and `'Taylor'`. The default value is `'None'`.
  - `'SidelobeAttenuation',slb` - specifies the sidelobe attenuation window, `slb`, of the Chebyshev or Taylor window as a positive scalar. The default value is 30. This parameter applies when you set `'SpectrumWindow'` to `'Chebyshev'` or `'Taylor'`.
  - `'Beta',beta` - specifies the parameter, `beta`, that determines the Kaiser window sidelobe attenuation as a nonnegative scalar. The default value is 0.5. This parameter applies when you set `'SpectrumWindow'` to `'Kaiser'`.
  - `'Nbar',nbar` - specifies the number of nearly constant level sidelobes, `nbar`, adjacent to the main lobe in a Taylor window as a positive integer. The default value is 4. This parameter applies when you set `'SpectrumWindow'` to `'Taylor'`.
  - `'SpectrumRange',sr` - specifies the spectrum region, `sr`, on which the spectrum window is applied as a 1-by-2 vector having the form `[StartFrequency EndFrequency]`. The default value is `[0 1.0e5]`. This parameter applies when you set the `'SpectrumWindow'` to any value other than `'None'`. Units are in Hz.

Both `StartFrequency` and `EndFrequency` are measured in the baseband region  $[-Fs/2 Fs/2]$ . `Fs` is the sample rate specified by the `SampleRate` property. `StartFrequency` cannot be larger than `EndFrequency`.

- `'StretchProcessor'` - The name-value pair arguments are
  - `'ReferenceRange',refrng` - specifies the center of ranges of interest, `refrng`, as a positive scalar. The `refrng` must be within the unambiguous range of one pulse. The default value is 5000. Units are in meters.
  - `'RangeSpan',rngspan` - specifies the span of the ranges of interest. `rngspan`, as a positive scalar. The range span is centered at the range value specified in the `'ReferenceRange'` parameter. The default value is 500. Units are in meters.
  - `'RangeFFTLength',len` - specifies the FFT length in the range domain, `len`, as a positive integer. If not specified, the default value is same as the input data length.

- 'RangeWindow', *rw* specifies the window used for range processing, *rw*, as one of 'None', 'Hamming', 'Chebyshev', 'Hann', 'Kaiser', and 'Taylor'. The default value is 'None'.

Data Types: cell

### Inherit sample rate — Inherit sample rate from upstream blocks

on (default) | off

Select this parameter to inherit the sample rate from upstream blocks. Otherwise, specify the sample rate using the **Sample rate (Hz)** parameter.

Data Types: Boolean

### Sample rate (Hz) — Sampling rate of signal

1e6 (default) | positive real-valued scalar

Specify the signal sampling rate as a positive scalar. Units are in Hz.

#### Dependencies

To enable this parameter, clear the **Inherit sample rate** check box.

Data Types: double

### Simulate using — Block simulation method

Interpreted Execution (default) | Code Generation

Block simulation, specified as Interpreted Execution or Code Generation. If you want your block to use the MATLAB interpreter, choose Interpreted Execution. If you want your block to run as compiled code, choose Code Generation. Compiled code requires time to compile but usually runs faster.

Interpreted execution is useful when you are developing and tuning a model. The block runs the underlying System object in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using Code Generation. Long simulations run faster with generated code than in interpreted execution. You can run repeated executions without recompiling, but if you change any block parameters, then the block automatically recompiles before execution.

This table shows how the **Simulate using** parameter affects the overall simulation behavior.

When the Simulink model is in Accelerator mode, the block mode specified using **Simulate using** overrides the simulation mode.

#### Acceleration Modes

Block Simulation	Simulation Behavior		
	Normal	Accelerator	Rapid Accelerator
Interpreted Execution	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
Code Generation	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink).

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **See Also**

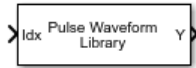
`pulseCompressionLibrary` | Pulse Compression Library

### **Introduced in R2021a**

## Pulse Waveform Library

Library of pulse waveforms

**Library:** Radar Toolbox



### Description

The Pulse Waveform Library generates different types of pulse waveforms from a library of waveforms.

### Ports

#### Input

##### **Idx — Waveform index**

positive integer

Index to select the waveform, specified as a positive integer. The index selects the waveform from the set of waveforms defined by the **Specification of each waveform in the library** parameter.

Data Types: `double`

#### Output

##### **Y — Pulse waveform samples**

complex-valued column vector | complex-valued matrix

Pulse waveform samples, returned as a complex-valued vector or complex-valued matrix.

Data Types: `double`

### Parameters

##### **Sample rate (Hz) — Sample rate of the output waveform**

1e6 (default) | positive scalar

Sample rate of the output waveform, specified as a positive scalar. The ratio of **Sample rate (Hz)** to each element in the **Pulse repetition frequency (Hz)** vector must be an integer. This restriction is equivalent to requiring that the pulse repetition interval is an integral multiple of the sample interval.

##### **Specification of each waveform in the library — Pulse waveforms in the library**

```
{{'Rectangular', 'PRF', 1e4, 'PulseWidth', 50e-6},
{'LinearFM', 'PRF', 1e4, 'PulseWidth', 50e-6, 'SweepBandwidth', 1e5, 'SweepDirection', 'Up', 'SweepInterval', 'Positive'}} (default) | cell array
```

Pulse waveforms, specified as a cell array. Each cell of the array contains the specification of one waveform. Each waveform is also a cell array containing the parameters of the waveform.

```
{{Waveform 1 Specification},{Waveform 2 Specification},{Waveform 3 Specification}, ...}
```



This block supports four built-in waveforms and also lets you specify custom waveforms. Each built-in waveform specifier consists of a waveform identifier followed by several name-value pairs that set the properties of the waveform.

### Built-in Waveforms

Waveform type	Waveform identifier	Waveform name-value pair arguments
Linear FM	'LinearFM'	See “Linear FM Waveform Arguments” on page 4-379
Phase coded	'PhaseCoded'	See “Phase-Coded Waveform Arguments” on page 4-381
Rectangular	'Rectangular'	See “Rectangular Waveform Arguments” on page 4-382
Stepped FM	'SteppedFM'	See “Stepped FM Waveform Arguments” on page 4-398

You can create a custom waveform with a user-defined function. The first input argument of the function must be the sample rate. Use a function handle instead of the waveform identifier in the first cell of a waveform specification. The remaining cells contain all function input arguments except the sample rate. Specify all input arguments in the order they are passed into the function. The function must have at least one output argument to return the samples of each pulse in a column vector. You can only create custom waveforms when you set **Simulate using** to `Interpreted Execution`.

### Source of simulation sample time — Source of simulation sample time

`Derive from waveform parameters (default)` | `Inherit from Simulink engine`

Source of simulation sample time, specified as `Derive from waveform parameters` or `Inherit from Simulink engine`. When set to `Derive from waveform parameters`, the block runs at a variable rate determined by the PRF of the selected waveform. The elapsed time is variable. When set to `Inherit from Simulink engine`, the block runs at a fixed rate so the elapsed time is a constant.

### Dependencies

To enable this parameter, select the **Enable PRF selection input** parameter.

### Simulate using — Block simulation method

`Interpreted Execution (default)` | `Code Generation`

Block simulation, specified as `Interpreted Execution` or `Code Generation`. If you want your block to use the MATLAB interpreter, choose `Interpreted Execution`. If you want your block to run as compiled code, choose `Code Generation`. Compiled code requires time to compile but usually runs faster.

`Interpreted execution` is useful when you are developing and tuning a model. The block runs the underlying System object in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using `Code Generation`. Long simulations run faster with generated code than in interpreted execution. You can run repeated executions without recompiling, but if you change any block parameters, then the block automatically recompiles before execution.

This table shows how the **Simulate using** parameter affects the overall simulation behavior.

When the Simulink model is in Accelerator mode, the block mode specified using **Simulate using** overrides the simulation mode.

### Acceleration Modes

Block Simulation	Simulation Behavior		
	Normal	Accelerator	Rapid Accelerator
Interpreted Execution	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
Code Generation	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink).

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### See Also

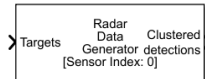
[pulseWaveformLibrary](#) | [pulseCompressionLibrary](#)

**Introduced in R2021a**

# Radar Data Generator

Generate radar sensor detections and tracks

**Library:** Radar Toolbox



## Description

The Radar Data Generator block reads target poses and time from a scenario reader and generates detection and track reports of targets from a radar sensor model. Use this block to generate sensor data from a scenario containing targets, sensors, and trajectories, which you can read from a Scenario Reader block or Tracking Scenario Reader.

The Radar Data Generator block can generate clustered or unclustered detections with added random noise and can also generate false alarm detections. You can also generate tracks from the Radar Data Generator block. Use the **Target reporting format** parameter to specify whether targets are output as clustered detections, unclustered detections, or tracks.

## Ports

### Input

#### Targets — Target poses

Simulink bus containing MATLAB structure

Target poses in platform coordinates, specified as a Simulink bus containing a MATLAB structure. The Targets input port can accept output from the Actors output port of the Scenario Reader block in the Automated Driving Toolbox™ or from the Platforms output port of the Tracking Scenario Reader in the Sensor Fusion and Tracking Toolbox™.

The Scenario Reader block and the Tracking Scenario Reader block output pose data in different formats. The Radar Data Generator reads data from either block. In each case, the data consists of two data fields followed by an array of structures. These structures define the number of Platforms or the number of Actors. Platforms and Actors are collectively called Targets.

Field		Description	Type
<b>Input block</b>	<b>Field name</b>	Number of valid target poses	Nonnegative integer
Scenario Reader	NumActors		
Tracking Scenario Reader	NumPlatforms		

Field		Description	Type
<b>Time</b>		Current simulation time (optional). If missing, the current Simulink simulation time is used.	Real-valued scalar
<b>Input block</b>	<b>Field name</b>	Valid target poses	Array of target pose structures
Scenario Reader	Actors		
Tracking Scenario Reader	Platforms		

The **Actors** structure is described in the output port of the Scenario Reader block and the **Platforms** structure is described in the output port of the Tracking Scenario Reader block.

### INS — Radar pose from INS

Simulink bus containing MATLAB structure

Radar pose information from an inertial navigation system (INS), specified as a Simulink bus containing a single MATLAB structure. The structure includes pose information for the radar platform that is provided by the INS. The INS information can then be used to estimate the target positions in the NED frame. INS is a struct with the following fields:

Field	Definition
Position	Position in the scenario frame specified as a real-valued 1-by-3 vector. Units are in meters.
Velocity	Velocity in the scenario frame specified as a real-valued 1-by-3 vector. Units are in m/s.
Orientation	Orientation with respect to the scenario frame, specified as a 3-by-3 real-valued rotation matrix. The rotation is from the navigation frame to the current INS body frame. This is also referred to as a "parent to child" rotation.

#### Dependencies

To enable this port, select the **Enable INS** check box.

#### Time — Current simulation time

nonnegative scalar

Current simulation time, specified as a nonnegative scalar. The sensor only generates reports at simulation times corresponding to integer multiples of the update interval, which is given by the reciprocal of the **Update rate (Hz)** parameter. Units are in seconds.

#### Dependencies

To enable this port, set the **Source of target truth time** to Input port.

If this port is not enabled, then the time is taken from the time on the Target poses input bus. If time is not on this bus, then the current Simulink simulation time is used.

Data Types: double

**Output**

**Clustered detections – Clustered object detections**

Simulink bus containing MATLAB structure

Clustered object detections, returned as a Simulink bus containing a MATLAB structure. For more details about buses, see “Create Nonvirtual Buses” (Simulink).

With clustered detections, the block outputs a single detection per target, where each detection is the centroid of the unclustered detections for that target.

You can pass object detections from these sensors and other sensors to a tracker, such as the Global Nearest Neighbor Multi Object Tracker block in the Sensor Fusion and Tracking Toolbox.

The structure contains these fields.

Field	Description	Type
NumDetections	Number of valid detections	Nonnegative integer
IsValidTime	False when updates are requested at times that are between block invocation intervals	Boolean
Detections	Object detections	Array of object detection structures of length set by the <b>Maximum number of target reports</b> parameter. Only NumDetections of these are actual detections.

Each object detection structure contains these properties.

Property	Definition
Time	Measurement time
Measurement	Object measurements
MeasurementNoise	Measurement noise covariance matrix
SensorIndex	Unique ID of the sensor
ObjectClassID	Object classification
ObjectAttributes	Additional information passed to tracker
MeasurementParameters	Parameters used by initialization functions of nonlinear Kalman tracking filters

- For rectangular coordinates, Measurement and MeasurementNoise are reported in the rectangular coordinate system specified by the **Coordinate system** parameter.
- For spherical coordinates, Measurement and MeasurementNoise are reported in the spherical coordinate system, which is based on the sensor rectangular coordinate system.

**Measurement and MeasurementNoise**

Coordinate System	Measurement and MeasurementNoise Coordinates																
Scenario	This table shows how coordinates are affected by the <b>Enable range rate measurements</b> parameter.																
Body																	
Sensor rectangular			<table border="1"> <thead> <tr> <th data-bbox="870 508 1159 569">Enable range rate measurements</th> <th data-bbox="1172 499 1472 569">Coordinates</th> </tr> </thead> <tbody> <tr> <td data-bbox="870 577 1159 615">on</td> <td data-bbox="1172 577 1472 615">[x;y;z;vx;vy;vz]</td> </tr> <tr> <td data-bbox="870 623 1159 661">off</td> <td data-bbox="1172 623 1472 661">[x;y;z]</td> </tr> </tbody> </table>	Enable range rate measurements	Coordinates	on	[x;y;z;vx;vy;vz]	off	[x;y;z]								
Enable range rate measurements	Coordinates																
on	[x;y;z;vx;vy;vz]																
off	[x;y;z]																
Sensor spherical	This table shows how coordinates are affected by the <b>Enable elevation angle measurements</b> and <b>Enable range rate measurements</b> parameters.																
	<table border="1"> <thead> <tr> <th data-bbox="870 800 1055 961">Enable range rate measurements</th> <th data-bbox="1068 791 1263 961">Enable elevation angle measurements</th> <th data-bbox="1269 791 1472 961">Coordinates</th> </tr> </thead> <tbody> <tr> <td data-bbox="870 970 1062 1008">on</td> <td data-bbox="1068 970 1263 1008">on</td> <td data-bbox="1269 970 1472 1039">[az;el;rng;rr]</td> </tr> <tr> <td data-bbox="870 1047 1062 1085">on</td> <td data-bbox="1068 1047 1263 1085">off</td> <td data-bbox="1269 1047 1472 1085">[az;rng;rr]</td> </tr> <tr> <td data-bbox="870 1094 1062 1131">off</td> <td data-bbox="1068 1094 1263 1131">on</td> <td data-bbox="1269 1094 1472 1131">[az;el;rng]</td> </tr> <tr> <td data-bbox="870 1140 1062 1178">off</td> <td data-bbox="1068 1140 1263 1178">off</td> <td data-bbox="1269 1140 1472 1178">[az;rng]</td> </tr> </tbody> </table>	Enable range rate measurements	Enable elevation angle measurements	Coordinates	on	on	[az;el;rng;rr]	on	off	[az;rng;rr]	off	on	[az;el;rng]	off	off	[az;rng]	
Enable range rate measurements	Enable elevation angle measurements	Coordinates															
on	on	[az;el;rng;rr]															
on	off	[az;rng;rr]															
off	on	[az;el;rng]															
off	off	[az;rng]															

For ObjectAttributes, this table describes the additional information used for tracking.

### ObjectAttributes

Attribute	Definition										
TargetIndex	Identifier of the ActorID or PlatformID of the target that generated the detection. For false alarms, this value is negative.										
SNR	Signal-to-noise ratio of the detection. Units are in dB.										
BounceTargetIndex	Identifier of the target generating the multipath bounce that produced the ghost target report. Only present when <b>HasGhosts</b> is true.										
BouncePathIndex	<p>Index of the bounce path associated with the target report. Only present when <b>HasGhosts</b> is true.</p> <p><b>Bounce-Path Index</b></p> <table border="1"> <thead> <tr> <th>BouncePathIndex</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Direct-path target report</td> </tr> <tr> <td>1</td> <td>First 2-bounce path detection</td> </tr> <tr> <td>2</td> <td>Second 2-bounce path</td> </tr> <tr> <td>3</td> <td>3-bounce path</td> </tr> </tbody> </table>	BouncePathIndex	Description	0	Direct-path target report	1	First 2-bounce path detection	2	Second 2-bounce path	3	3-bounce path
BouncePathIndex	Description										
0	Direct-path target report										
1	First 2-bounce path detection										
2	Second 2-bounce path										
3	3-bounce path										

For MeasurementParameters, the measurements are relative to the parent frame. When you set the **Coordinate system** parameter to Body, the parent frame is the platform body. When you set **Coordinate system** to Sensor rectangular or Sensor spherical, the parent frame is the sensor.

### MeasurementParameters

Parameter	Definition
Frame	Enumerated type indicating the frame used to report measurements. When Frame is set to 'rectangular', detections are reported in Cartesian coordinates. When Frame is set to 'spherical', detections are reported in spherical coordinates.
OriginPosition	3-D vector offset of the sensor origin from the parent frame origin.
Orientation	Orientation of the radar sensor coordinate system with respect to the parent frame.
HasVelocity	Indicates whether measurements contain velocity or range rate components.
HasElevation	Indicates whether measurements contain elevation components.

## Dependencies

To enable this port, select the **Target reporting format** pull-down menu as **Clustered detections**.

## Tracks — Object tracks

Simulink bus containing MATLAB structure

Object tracks, returned as a Simulink bus containing a MATLAB structure. See “Create Nonvirtual Buses” (Simulink).

This table shows the structure fields.

Field	Description
NumTracks	Number of tracks
IsValidTime	False when updates are requested at times that are between block invocation intervals
Tracks	Array of track structures of a length set by the <b>Maximum number of target reports</b> parameter. Only the first NumTracks of these are actual tracks.

This table shows the fields of each track structure.

Field	Definition
TrackID	Unique track identifier used to distinguish multiple tracks.
BranchID	Unique track branch identifier used to distinguish multiple track branches.
SourceIndex	Unique source index used to distinguish tracking sources in a multiple tracker environment.
UpdateTime	Time at which the track is updated. Units are in seconds.
Age	Number of times the track was updated.
State	Value of state vector at the update time.
StateCovariance	Uncertainty covariance matrix.
ObjectClassID	Integer value representing the object classification. The value 0 represents an unknown classification. Nonzero classifications apply only to confirmed tracks.
TrackLogic	Confirmation and deletion logic type. This value is always 'History' for radar sensors, to indicate history-based logic.
TrackLogicState	Current state of the track logic type, returned as a 1-by-K logical array. K is the number of latest track logical states recorded. In the array, 1 denotes a hit and 0 denotes a miss.



Field	Definition
IsConfirmed	Confirmation status. This field is <code>true</code> if the track is confirmed to be a real target.
IsCoasted	Coasting status. This field is <code>true</code> if the track is updated without a new detection.
IsSelfReported	Indicate if the track is reported by the tracker. This field is used in a track fusion environment. It is returned as <code>true</code> by default.
ObjectAttributes	Additional information about the track.

For more details about these fields, see `objectTrack`.

The block outputs only confirmed tracks, which are tracks to which the block assigns at least  $M$  detections during the first  $N$  updates after track initialization. To specify the values  $M$  and  $N$ , use the **M and N for the M-out-of-N confirmation** parameter.

#### Dependencies

To enable this port, on the **Parameters** tab, set the **Target reporting format** parameter to `Tracks`.

#### Detections – Unclustered object detections

Simulink bus containing MATLAB structure

Unclustered object detections, returned as a Simulink bus containing a MATLAB structure. For more details about buses, see “Create Nonvirtual Buses” (Simulink).

With unclustered detections, the block outputs all detections, and a target can have multiple detections.

You can pass object detections from these sensors and other sensors to a tracker, such as a Multi-Object Tracker block, and generate tracks.

The structure must contain these fields:

Field	Description	Type
NumDetections	Number of valid detections	integer
IsValidTime	False when updates are requested at times that are between block invocation intervals	Boolean
Detections	Object detections	Array of object detection structures of length set by the <b>Maximum number of target reports</b> parameter. Only <code>NumDetections</code> of these are actual detections.

Each object detection structure contains these properties.

Property	Definition
Time	Measurement time
Measurement	Object measurements
MeasurementNoise	Measurement noise covariance matrix
SensorIndex	Unique ID of the sensor
ObjectClassID	Object classification
ObjectAttributes	Additional information passed to tracker
MeasurementParameters	Parameters used by initialization functions of nonlinear Kalman tracking filters

- For rectangular coordinates, **Measurement** and **MeasurementNoise** are reported in the rectangular coordinate system specified by the **Coordinate system** parameter.
- For spherical coordinates, **Measurement** and **MeasurementNoise** are reported in the spherical coordinate system, which is based on the sensor rectangular coordinate system.

### Measurement and MeasurementNoise

Coordinate System	Measurement and MeasurementNoise Coordinates		
Scenario	This table shows how coordinates are affected by the <b>Enable range rate measurements</b> parameter.		
Body			
Sensor rectangular			
	<b>Enable range rate measurements</b>	<b>Coordinates</b>	
	on	[x;y;z;vx;vy;vz]	
	off	[x;y;z]	
Sensor spherical	This table shows how coordinates are affected by the <b>Enable elevation angle measurements</b> and <b>Enable range rate measurements</b> parameters.		
	<b>Enable range rate measurements</b>	<b>Enable elevation angle measurements</b>	<b>Coordinates</b>
	on	on	[az;el;rng;rr]
	on	off	[az;rng;rr]
	off	on	[az;el;rng]
	off	off	[az;rng]

For **ObjectAttributes**, this table describes the additional information used for tracking.

### ObjectAttributes

Attribute	Definition										
TargetIndex	Identifier of the ActorID or PlatformID of the target that generated the detection. For false alarms, this value is negative.										
SNR	Signal-to-noise ratio of the detection. Units are in dB.										
BounceTargetIndex	Identifier of the target generating the multipath bounce that produced the ghost target report. Only present when <b>HasGhosts</b> is true.										
BouncePathIndex	<p>Index of the bounce path associated with the target report. Only present when <b>HasGhosts</b> is true.</p> <p><b>Bounce-Path Index</b></p> <table border="1"> <thead> <tr> <th>BouncePathIndex</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Direct-path target report</td> </tr> <tr> <td>1</td> <td>First 2-bounce path detection</td> </tr> <tr> <td>2</td> <td>Second 2-bounce path</td> </tr> <tr> <td>3</td> <td>3-bounce path</td> </tr> </tbody> </table>	BouncePathIndex	Description	0	Direct-path target report	1	First 2-bounce path detection	2	Second 2-bounce path	3	3-bounce path
BouncePathIndex	Description										
0	Direct-path target report										
1	First 2-bounce path detection										
2	Second 2-bounce path										
3	3-bounce path										

For MeasurementParameters, the measurements are relative to the parent frame. When you set the **Coordinate system** parameter to Body, the parent frame is the platform body. When you set **Coordinate system** to Sensor rectangular or Sensor spherical, the parent frame is the sensor.

### MeasurementParameters

Parameter	Definition
Frame	Enumerated type indicating the frame used to report measurements. When Frame is set to 'rectangular', detections are reported in Cartesian coordinates. When Frame is set to 'spherical', detections are reported in spherical coordinates.
OriginPosition	3-D vector offset of the sensor origin from the parent frame origin.
Orientation	Orientation of the radar sensor coordinate system with respect to the parent frame.
HasVelocity	Indicates whether measurements contain velocity or range rate components.
HasElevation	Indicates whether measurements contain elevation components.

## Dependencies

To enable this port, set the **Target reporting format** parameter to **Detections**.

## Configuration – Current sensor configuration

Simulink bus containing MATLAB structure

Configuration, returned as a Simulink bus containing a MATLAB structure. This output can be used to determine which objects fall within the radar beam during object execution. The structure fields are:

Field	Description	Type
NumConfigurations	Number of valid configurations	integer
Configurations	Configuration structure	Array of NumConfigurations configuration structures

The configuration structure has these fields:

Field	Description
SensorIndex	Unique sensor index, returned as a positive integer.
IsValidTime	Valid detection time, returned as <code>true</code> or <code>false</code> . <code>IsValidTime</code> is <code>false</code> when detection updates are requested between update intervals specified by the update rate.
IsScanDone	<code>IsScanDone</code> is <code>true</code> when the sensor has completed a scan.
FieldOfView	Field of view of the sensor, returned as a 2-by-1 vector of positive real values, <code>[azfov;elfov]</code> . <code>azfov</code> and <code>elfov</code> represent the field of view in azimuth and elevation, respectively.
RangeLimits	Minimum and maximum range of sensor, in meters, specified as a 1-by-2 nonnegative real-valued vector of the form <code>[rmin, rmax]</code> .
RangeRateLimits	Minimum and maximum range rate of sensor, in meters per second, specified as a 1-by-2 real-valued vector of the form <code>[rrmin, rrmx]</code> .
MeasurementParameters	Sensor measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current sensor frame. For details on <code>MeasurementParameters</code> , see “Measurement Parameters” on page 4-205.

## Dependencies

To enable this port, select the **Enable radar configuration output** check box.

## Parameters

### Parameters

#### Sensor Identification

##### Unique identifier of sensor – Unique sensor identifier

0 (default) | positive integer

Unique sensor identifier, specified as a positive integer. Use this parameter to distinguish between detections or tracks that come from different sensors in a multisensor system. Specify a unique value for each sensor. If you do not update **Unique identifier of sensor** from the default value of 0, then the radar returns an error at the start of simulation.

##### Update rate (Hz) – Sensor update rate

10 (default) | positive real scalar

Update rate, specified as a positive real scalar. The radar generates new reports at intervals defined by this reciprocal value. Any sensor update requested between update intervals contains no detections or tracks. Units are in Hz.

#### Sensor Mounting

##### Translation [ X, Y, Z ] relative to ego origin (m) – Mounting location of radar on platform

[3.4, 0, 0.2] (default) | 1-by-3 real-valued vector of form [x, y, z]

Sensor location on the radar on the platform, specified as a 1-by-3 real-valued vector of the form [x, y, z]. This parameter defines the coordinates of the sensor along the x-axis, y-axis, and z-axis relative to the platform origin. Units are in meters.

##### Rotation [Yaw,Pitch,Roll] relative to ego's frame (deg) – Mounting rotation angles of radar

[0 0 0] (default) | 1-by-3 real-valued vector of form [ $z_{yaw}$   $y_{pitch}$   $x_{roll}$ ]

Mounting rotation angles of the radar, specified as a 1-by-3 real-valued vector of form [ $z_{yaw}$   $y_{pitch}$   $x_{roll}$ ]. This parameter defines the intrinsic Euler angle rotation of the sensor around the z-axis, y-axis, and x-axis with respect to the platform frame, where:

- $z_{yaw}$ , or yaw angle, rotates the sensor around the z-axis of the platform frame.
- $y_{pitch}$ , or pitch angle, rotates the sensor around the y-axis of the platform frame. This rotation is relative to the sensor position that results from the  $z_{yaw}$  rotation.
- $x_{roll}$ , or roll angle, rotates the sensor about the x-axis of the platform frame. This rotation is relative to the sensor position that results from the  $z_{yaw}$  and  $y_{pitch}$  rotations.

These angles are clockwise-positive when looking in the forward direction of the z-axis, y-axis, and x-axis, respectively. Units are in degrees.

#### Detection Reporting

##### Enable elevation angle measurements – Enable radar to measure target elevation angles

off (default) | on

Select this check box to model a radar sensor that can estimate target elevation.

**Enable range rate measurements — Enable radar to measure target range rates**

on (default) | off

Select this check box to enable the radar to measure range rates from target detections.

**Add noise to measurements — Enable addition of noise to radar sensor measurements**

on (default) | off

Select this parameter to add noise to the radar measurements. Otherwise, the measurements have no noise. Even if you clear this parameter, the measurement noise covariance matrix, which is reported in the `MeasurementNoise` field of the generated detections output, represents the measurement noise that is added when **Add noise to measurements** is selected.

**Enable false reports — Enable creating false alarm radar detections**

on (default) | off

Select this parameter to enable creating false alarm radar measurements. If you clear this parameter, the radar reports only actual detections.

**Enable occlusion — Enable line-of-sight occlusion**

on (default) | off

Select this parameter to enable line-of-sight occlusion, where the radar generates detection only from objects for which the radar has a direct line of sight. For example, with this parameter enabled, the radar does not generate a detection for an object that is behind another object and blocked from view.

**Enable ghosts — Enable ghost targets**

off (default) | on

Select this parameter to generate ghost targets for multipath propagation paths having up to three reflections between transmission and reception of the radar signal.

**Maximum number of target reports — Maximum number of detections or tracks**

50 (default) | positive integer

Maximum number of detections or tracks that the sensor reports, specified as a positive integer. The sensor reports detections in order of increasing distance from the sensor until reaching this maximum number.

**Target reporting format — Format of generated target reports****Clustered detections** (default) | **Tracks** | **Detections**

Format of generated target reports, specified as one of these options:

- **Clustered detections** — The block generates target reports as clustered detections, where each target is reported as a single detection that is the centroid of the unclustered target detections. The block returns clustered detections at the **Clustered detections** output port.
- **Tracks** — The block generates target reports as tracks, which are clustered detections that have been processed by a tracking filter. The block returns clustered detections at the **Tracks** output port.
- **Detections** — The block generates target reports as unclustered detections, where each target can have multiple detections. The block returns clustered detections at the **Detections** output port.

**Coordinate system — Coordinate system of reported detections****Body** (default) | **Sensor rectangular** | **Sensor spherical** | **Scenario**

Coordinate system of reported detections, specified as one of these options:

- **Body** — Detections are reported in the rectangular body system of the sensor platform.
- **Sensor rectangular** — Detections are reported in the sensor rectangular body coordinate system.
- **Sensor spherical** — Detections are reported in a spherical coordinate system that is centered at the radar sensor and aligned with the orientation of the radar on the platform.
- **Scenario** — Detections are reported in the rectangular scenario coordinate frame. The scenario coordinate system is defined as the local navigation frame at simulation start time.

**Port Settings****Source of target truth time — Source of target truth time****Auto** (default) | **Input port**

Source of output truth time, specified as one of these options:

- **Auto** — The block uses the time provided on the target bus, or if not present, the current Simulink simulation time.
- **Input port** — The block uses the time provided on the Time input port of the block.

**Enable INS — Enable INS input port****off** (default) | **on**

Select this parameter to allow input of INS data using the INS input port.

**Source of output target report bus name — Source of output target report bus name****Auto** (default) | **Property**

Source of output target report bus name, specified as one of these options:

- **Auto** — The block automatically creates a bus name.
- **Property** — Specify the bus name by using the **Specify an output target report bus name** parameter.

This bus contains Clustered detections, Tracks, or Detections output port data.

**Specify an output target report bus name — Name of target report output bus****BusRadarDataGenerator** (default) | **valid bus name**

Name of the target report bus to be returned in output port, specified as a valid bus name.

**Dependencies**

To enable this parameter, set the **Source of output target report bus name** parameter to **Property**.

**Enable radar configuration output — Enable radar configuration output****off** (default) | **on**

Enable the Configuration output port.

**Source of output config bus name — Source of output config bus name**

Auto (default) | Property

Source of output config bus name, specified as one of these options:

- Auto — The block automatically creates a bus name.
- Property — Specify the bus name by using the **Specify an output config bus name** parameter.

**Specify an output config bus name — Name of target report output bus**

BusRadarDataGeneratorConfig (default) | valid bus name

Specify the name of the config bus returned in the output port.

**Dependencies**

To enable this parameter, set the **Source of output config bus name** parameter to Property.

**Measurements****Resolution Settings****Azimuth resolution (deg) — Azimuth resolution of radar**

4 (default) | positive real scalar

Azimuth resolution of the radar, specified as a positive scalar. The azimuth resolution defines the minimum separation in azimuth angle at which the radar can distinguish between two targets. The azimuth resolution is typically the 3 dB downpoint of the azimuth angle beamwidth of the radar. Units are in degrees.

**Elevation resolution (deg) — Elevation resolution of radar**

5 (default) | positive real scalar

Elevation resolution of the radar, specified as a positive scalar. The elevation resolution defines the minimum separation in elevation angle at which the radar can distinguish between two targets. The elevation resolution is typically the 3 dB downpoint of the elevation angle beamwidth of the radar. Units are in degrees.

**Dependencies**

To enable this parameter, select the **Enable elevation angle measurements** check box.

**Range resolution (m) — Range resolution of radar**

2.5 (default) | positive real scalar

Range resolution of the radar in meters, specified as a positive real scalar. The range resolution defines the minimum separation in range at which the radar can distinguish between two targets. Units are in meters.

**Range rate resolution (m/s) — Range rate resolution of radar**

0.5 (default) | positive real scalar

Range rate resolution of the radar, specified as a positive real scalar. The range rate resolution defines the minimum separation in range rate at which the radar can distinguish between two targets. Units are in meters per second.



**Dependencies**

To enable this parameter, on the **Parameters** tab, select the **Enable range rate measurements** check box.

**Bias Settings****Azimuth bias fraction — Azimuth bias fraction of radar**

0.1 (default) | nonnegative scalar

Azimuth bias fraction of the radar, specified as a nonnegative scalar. Azimuth bias is expressed as a fraction of the azimuth resolution specified in the **Azimuth resolution (deg)** parameter. This value sets a lower bound on the azimuthal accuracy of the radar and is dimensionless.

**Elevation bias fraction — Elevation bias fraction of radar**

0.1 (default) | nonnegative scalar

Elevation bias fraction of the radar, specified as a nonnegative scalar. Elevation bias is expressed as a fraction of the elevation resolution specified in the **Elevation resolution (deg)** parameter. This value sets a lower bound on the elevation accuracy of the radar and is dimensionless.

**Dependencies**

To enable this parameter, select the **Enable elevation angle measurements** check box.

**Range bias fraction — Range bias fraction**

0.05 (default) | nonnegative scalar

Range bias fraction of the radar, specified as a nonnegative scalar. Range bias is expressed as a fraction of the range resolution specified by the **Range resolution (m)** property. This property sets a lower bound on the range accuracy of the radar and is dimensionless.

**Range rate bias fraction — Range rate bias fraction**

0.05 (default) | nonnegative scalar

Range rate bias fraction of the radar, specified as a nonnegative scalar. Range rate bias is expressed as a fraction of the range rate resolution specified by the **Range rate resolution (m/s)** parameter. This property sets a lower bound on the range rate accuracy of the radar and is dimensionless.

**Dependencies**

To enable this parameter, select the **Enable range rate measurements** check box.

**Detector Settings****Total angular field of view [AZ, EL] (deg) — Angular field of view of radar**

[20 5] (default) | 1-by-2 positive real-valued vector of form [azfov, elfov]

Angular field of view of the radar, specified as a 1-by-2 positive real-valued vector of the form [azfov elfov]. The field of view defines the total angular extent spanned by the sensor. The azimuth field of view, azfov, must lie in the interval (0, 360]. The elevation field of view, elfov, must lie in the interval (0, 180]. Units are in degrees

**Range limits [MIN, MAX] (m) — Minimum and maximum range of radar**

[0 150] (default) | 1-by-2 nonnegative real-valued vector of form [min max]

Minimum and maximum range of the radar, specified as a 1-by-2 nonnegative real-valued vector of the form `[min max]`. The radar does not detect targets that are outside this range. The maximum range, `max`, must be greater than the minimum range, `min`. Units are in meters.

**Range rate limits [MIN, MAX] (m/s) – Minimum and maximum range rate of radar (m/s)**

`[-100 100]` (default) | 1-by-2 real-valued vector of form `[min max]`

Minimum and maximum range rate of radar as a 1-by-2 real-valued vector of the form `[min max]`. The radar does not detect targets that are outside this range rate. The maximum range rate, `max`, must be greater than the minimum range rate, `min`. Units are in meters per second.

**Dependencies**

To enable this parameter, select the **Enable range rate measurements** check box.

**Detection probability – Probability of detecting a target**

`0.9` (default) | scalar in range (0, 1]

Probability of detecting a target as a scalar, specified as a scalar in the range (0, 1]. This quantity defines the probability of detecting a target with a radar cross-section, with the radar cross-section specified by the **Reference target RCS (dBsm)** parameter at the reference detection range specified by the **Reference target range (m)** parameter. Units are dimensionless.

**False alarm rate – False alarm report rate**

`1e-06` (default) | positive real scalar in range  $[10^{-7}, 10^{-3}]$

False alarm report rate within each radar resolution cell, specified as a positive real scalar in the range  $[10^{-7}, 10^{-3}]$ . The block determines resolution cells from the **Azimuth resolution (deg)** and **Range resolution (m)** parameters and, when enabled, from the **Elevation resolution (deg)** and **Range rate resolution (m/s)** parameters. Units are dimensionless.

**Reference target range (m) – Reference range for given probability of detection**

`100` (default) | positive real scalar

Reference range for the given probability of detection and the given reference radar cross-section (RCS), specified as a positive real scalar. The reference range is the range at which a target having a radar cross-section specified by the **Reference target RCS (dBsm)** parameter is detected with a probability of detection specified by the **Detection probability** parameter. Units are in meters.

**Reference target RCS (dBsm) – Reference radar cross-section for given probability of detection**

`0` (default) | real scalar

Reference radar cross-section (RCS) for a given probability of detection and reference range, specified as a real scalar. The reference RCS is the RCS value at which a target is detected with a probability specified by the **Detection probability** parameter at the specified **Reference target range (m)** parameter value. Values are expressed in dBsm.

**Center frequency (Hz) – Center frequency of radar band**

`77e9` (default) | positive real scalar

Center frequency of the radar band, specified as a positive scalar. Units are in Hz.

## Tracker Setting

### Filter initialization function name — Kalman filter initialization function

`initcvekf` (default) | function handle

Kalman filter initialization function, specified as a character vector or string scalar of the name of a valid Kalman filter initialization function.

The table shows the initialization functions that you can use to specify **Filter initialization function name**.

Initialization Function	Function Definition
<code>initcaabf</code>	Initialize constant-acceleration alpha-beta Kalman filter
<code>initcvabf</code>	Initialize constant-velocity alpha-beta Kalman filter
<code>initcakf</code>	Initialize constant-acceleration linear Kalman filter.
<code>initcvkf</code>	Initialize constant-velocity linear Kalman filter.
<code>initcaekf</code>	Initialize constant-acceleration extended Kalman filter.
<code>initctekf</code>	Initialize constant-turnrate extended Kalman filter.
<code>initcvekf</code>	Initialize constant-velocity extended Kalman filter.
<code>initcaukf</code>	Initialize constant-acceleration unscented Kalman filter.
<code>initctukf</code>	Initialize constant-turnrate unscented Kalman filter.
<code>initcvukf</code>	Initialize constant-velocity unscented Kalman filter.

You can also write your own initialization function. The function must have the following syntax:

```
filter = filterInitializationFcn(detection)
```

The input to this function is a detection report like those created by an `objectDetection` object. The output of this function must be a tracking filter object, such as `trackingKF`, `trackingEKF`, `trackingUKF`, or `trackingABF`.

To guide you in writing this function, you can examine the details of the supplied functions from within MATLAB. For example:

```
type initcvekf
```

### Dependencies

To enable this parameter, set the **Target reporting format** parameter to 'Tracks'.

### M and N for the M-out-of-N confirmation — Threshold for track confirmation

`[2 3]` (default) | 1-by-2 vector of positive integers

Threshold for track confirmation, specified as a 1-by-2 vector of positive integers of the form [M N]. A track is confirmed if it receives at least M detections in the last N updates. M must be less than or equal to N.

- When setting M, take into account the probability of object detection for the sensors. The probability of detection depends on factors such as occlusion or clutter. You can reduce M when tracks fail to be confirmed or increase M when too many false detections are assigned to tracks.
- When setting N, consider the number of times you want the tracker to update before it makes a confirmation decision. For example, if a tracker updates every 0.05 seconds, and you want to allow 0.5 seconds to make a confirmation decision, set  $N = 10$ .

#### Dependencies

To enable this parameter, set the **Target reporting format** parameter to 'Tracks'.

#### P and R for the P-out-of-R deletion — Threshold for track deletion

[5 5] (default) | 1-by-2 vector of positive integers

Threshold for track deletion, specified as a 1-by-2 vector of positive integers of the form [P R]. If a confirmed track is not assigned to any detection P times in the last R tracker updates, then the track is deleted. P must be less than or equal to R.

- To reduce how long the radar maintains tracks, decrease R or increase P.
- To maintain tracks for a longer time, increase R or decrease P.

#### Dependencies

To enable this parameter, set the **Target reporting format** parameter to 'Tracks'.

#### Random Number Generator Settings

##### Random number generation — Method to specify random number generator seed

Repeatable (default) | Specify seed | Not repeatable

Method to set the random number generator seed as one of the options in the table.

Option	Description
Repeatable	The block generates a random initial seed for the first simulation and reuses this seed for all subsequent simulations. Select this parameter to generate repeatable results from the statistical sensor model. To change this initial seed, at the MATLAB command prompt, enter: <code>clear all</code> .
Specify seed	Specify your own random initial seed for reproducible results by using the <b>Initial seed</b> parameter.
Not repeatable	The block generates a new random initial seed after each simulation run. Select this parameter to generate nonrepeatable results from the statistical sensor model.

##### Initial seed — Random number generator seed

0 (default) | nonnegative integer less than  $2^{32}$

Random number generator seed, specified as a nonnegative integer less than  $2^{32}$ .

#### Dependencies

To enable this parameter, set the **Random number generation** parameter to Specify seed.

#### Target Profiles

##### Target profiles definition — Method to specify target profiles

From Scenario Reader block (default) | MATLAB expression | Parameters

Method to specify target profiles, as one of Parameters, MATLAB expression, From Scenario Reader block. Profiles are the physical and radar characteristics of targets in the scenario.

- Parameters — The block obtains the target profiles from these parameters:
  - **Unique target identifiers**
  - **Target classification identifiers**
  - **Length of target cuboids (m)**
  - **Width of target cuboids (m)**
  - **Height of target cuboids (m)**
  - **Rotational center of target cuboids (m)**
  - **Target signatures**
- MATLAB expression — The block obtains the target profiles from the MATLAB expression specified by the **MATLAB expression for target profiles** parameter.
- From Scenario Reader block — The block obtains the actor profiles from the scenario specified by a scenario reader block such as Scenario Reader.

##### MATLAB expression for target profiles — MATLAB expression for target profiles

MATLAB structure | MATLAB structure array | valid MATLAB expression

Specify the MATLAB expression for target profiles, as a MATLAB structure, a MATLAB structure array, or a valid MATLAB expression that produces such a structure or structure array.

If your Scenario Reader block reads data from a drivingScenario object, to obtain the actor profiles directly from this object, set this expression to call the actorProfiles function on the object. For example: actorProfiles(scenario).

The default target profile expression produces a MATLAB structure and has this form:

```
struct('ClassID',0,'Length',4.7,'Width',1.8,'Height',1.4, ...
'OriginOffset',[-1.35 0 0],'RCSPattern',[10 10;10 10], ...
'RCSAzimuthAngles',[-180 180],'RCSElevationAngles',[-90 90])
```

#### Dependencies

To enable this parameter, set the **Target profiles definition** parameter to MATLAB expression.

##### Unique target identifiers — Scenario-defined target identifier

[] (default) | positive integer | length-*L* vector of unique positive integers

Specify the scenario-defined target identifier as a positive integer or length-*L* vector of unique positive integers. *L* must equal the number of targets input into the **Targets** input port. The vector

elements must match TargetID values of the targets. You can specify **Unique target identifiers** as [ ]. In this case, the same target profile parameters apply to all targets.

Example: [ 1 2 ]

#### Dependencies

To enable this parameter, set the **Target profiles definition** parameter to Parameters.

#### Target classification identifiers – User-defined classification identifier

0 (default) | integer | length-*L* vector of integers

Specify the user-defined classification identifier as an integer or length-*L* vector of integers. When **Unique target identifiers** is a vector, this parameter is a vector of the same length with elements in one-to-one correspondence to the targets in **Unique target identifiers**. When **Unique target identifiers** is empty, [ ], you must specify this parameter as a single integer whose value applies to all targets.

Example: 2

#### Dependencies

To enable this parameter, set the **Target profiles definition** parameter to Parameters.

#### Length of target cuboids (m) – Length of target cuboids

4.7 (default) | positive real scalar | length-*L* vector of positive values

Specify the length of target cuboids as a positive real scalar or length-*L* vector of positive values. When **Unique target identifiers** is a vector, this parameter is a vector of the same length with elements in one-to-one correspondence to the targets in **Unique target identifiers**. When **Unique target identifiers** is empty, [ ], you must specify this parameter as a positive real scalar whose value applies to all targets. Units are in meters.

Example: 6.3

#### Dependencies

To enable this parameter, set the **Target profiles definition** parameter to Parameters.

#### Width of target cuboids (m) – Width of target cuboids

1.8 (default) | positive real scalar | length-*L* vector of positive values

Specify the width of target cuboids as a positive real scalar or length-*L* vector of positive values. When **Unique target identifiers** is a vector, this parameter is a vector of the same length with elements in one-to-one correspondence to the targets in **Unique target identifiers**. When **Unique target identifiers** is empty, [ ], you must specify this parameter as a positive real scalar whose value applies to all targets. Units are in meters.

Example: 4.7

#### Dependencies

To enable this parameter, set the **Target profiles definition** parameter to Parameters.

#### Height of heights cuboids (m) – Height of actor cuboids

1.4 (default) | positive real scalar | length-*L* vector of positive values

Specify the height of target cuboids as a positive real scalar or length-*L* vector of positive values. When **Unique target identifiers** is a vector, this parameter is a vector of the same length with

elements in one-to-one correspondence to the targets in **Unique target identifiers**. When **Unique target identifiers** is empty, [], you must specify this parameter as a positive real scalar whose value applies to all targets. Units are in meters.

Example: 2.0

#### Dependencies

To enable this parameter, set the **Target profiles definition** parameter to Parameters.

#### Rotational center of target cuboids (m) — Rotational center of target cuboids

{[-1.35, 0, 0]} (default) | length-*L* cell array of real-valued 1-by-3 vectors

Specify the rotational center of target cuboids as a length-*L* cell array of real-valued 1-by-3 vectors. Each vector represents the offset of the rotational center of an target cuboid from the bottom-center of the target. When **Unique target identifiers** is a vector, this parameter is a cell array of vectors with cells in one-to-one correspondence to the targets in **Unique target identifiers**. When **Unique target identifiers** is empty, [], you must specify this parameter as a cell array of one element containing an offset vector whose values apply to all targets. Units are in meters.

Example: {[-1.35, 0.2, 0.3]}

#### Dependencies

To enable this parameter, set the **Target profiles definition** parameter to Parameters.

#### Target signatures — Target signatures

cell array

Target signatures, specified as a cell array of `rcsSignature` objects, which specify the RCS signature of the target.

#### Dependencies

#### Dependencies

To enable this parameter, set the **Target profiles definition** parameter to Parameters.

### See Also

`radarDataGenerator` | `objectDetection` | `objectTrack` | `rcsSignature` | Scenario Reader | Tracking Scenario Reader

#### Introduced in R2021b

## Two-Ray Channel

Two-ray channel environment



## Library

Environment and Target

phasedenvlib

## Description

The Two-Ray Channel block propagates narrowband signals from one point in space to multiple points or from multiple points back to one point via both the direct path and the ground reflection path. The block models propagation time, free-space propagation loss, and Doppler shift. The block assumes that the propagation speed is much greater than the object's speed in which case the stop-and-hop model is valid.

## Parameters

### Signal Propagation speed (m/s)

Specify the propagation speed of the signal, in meters per second, as a positive scalar. You can use the function `physconst` to specify the speed of light.

### Signal carrier frequency (Hz)

Specify the carrier frequency of the signal in hertz of the narrowband signal as a positive scalar.

### Specify atmospheric parameters

Select this check box to enable atmospheric attenuation modeling.

### Temperature (degrees Celsius)

Ambient atmospheric temperature, specified as a real-valued scalar. Units are degrees Celsius. This parameter appears when you select the **Specify atmospheric parameters** check box. Units are degrees Celsius.

### Dry air pressure (Pa)

Atmospheric dry air pressure, specified as a positive real-valued scalar. Units are Pascals (Pa). The value 101325 for this property corresponds to one standard atmosphere. This parameter appears when you select the **Specify atmospheric parameters** check box.

### Water vapour density (g/m<sup>3</sup>)

Atmospheric water vapor density, specified as a positive real-valued scalar. Units are gm/m<sup>3</sup>. This parameter appears when you select the **Specify atmospheric parameters** check box.



**Liquid water density (g/m<sup>3</sup>)**

Liquid water density of fog or clouds, specified as a non-negative real-valued scalar. Units are gm/m<sup>3</sup>. Typical values for liquid water density are 0.05 for medium fog and 0.5 for thick fog. This parameter appears when you select the **Specify atmospheric parameters** check box.

**Rain rate (mm/hr)**

Rainfall rate, specified as a non-negative real-valued scalar. Units are in mm/hour. This parameter appears when you select the **Specify atmospheric parameters** check box.

**Inherit sample rate**

Select this check box to inherit the sample rate from upstream blocks. Otherwise, specify the sample rate using the **Sample rate (Hz)** parameter.

**Sample rate (Hz)**

Specify the signal sampling rate (in hertz) as a positive scalar. This parameter appears only when the **Inherit sample rate** parameter is not selected.

**Ground reflection coefficient**

Fraction of incident signal amplitude reflected towards receiver.

**Combine two rays at output**

Select this check box to coherently sum the direct-path and reflected-path signals at output. Clear the check box to keep the two rays separate.

**Maximum one-way propagation distance (m)**

The maximum distance between the signal origin and the destination, specified as a positive scalar. Units are in meters. Amplitudes of any signals that propagate beyond this distance will be set to zero.

**Simulate using**

Block simulation method, specified as **Interpreted Execution** or **Code Generation**. If you want your block to use the MATLAB interpreter, choose **Interpreted Execution**. If you want your block to run as compiled code, choose **Code Generation**. Compiled code requires time to compile but usually runs faster.

Interpreted execution is useful when you are developing and tuning a model. The block runs the underlying System object in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using **Code Generation**. Long simulations run faster than they would in interpreted execution. You can run repeated executions without recompiling. However, if you change any block parameters, then the block automatically recompiles before execution.

When setting this parameter, you must take into account the overall model simulation mode. The table shows how the **Simulate using** parameter interacts with the overall simulation mode.

When the Simulink model is in **Accelerator** mode, the block mode specified using **Simulate using** overrides the simulation mode.

### Acceleration Modes

Block Simulation	Simulation Behavior		
	Normal	Accelerator	Rapid Accelerator
Interpreted Execution	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
Code Generation	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink).

### Ports

**Note** The block input and output ports correspond to the input and output parameters described in the `step` method of the underlying System object. See link at the bottom of this page.

Port	Description	Supported Data Types
X	Input signal.	Double-precision floating point
Pos1	Signal source position.	Double-precision floating point
Pos2	Signal destination position.	Double-precision floating point
Vel1	Signal source velocity.	Double-precision floating point
Vel2	Signal destination velocity.	Double-precision floating point
Out	Propagated signal.	Double-precision floating point

### Algorithms

When the origin and destination are stationary relative to each other, the block output can be written as  $y(t) = x(t - \tau)/L$ . The quantity  $\tau$  is the delay and  $L$  is the propagation loss. The delay is computed from  $\tau = R/c$  where  $R$  is the propagation distance and  $c$  is the propagation speed. The free space path loss is given by

$$L_{fsp} = \frac{(4\pi R)^2}{\lambda^2},$$

where  $\lambda$  is the signal wavelength.

This formula assumes that the target is in the far-field of the transmitting element or array. In the near-field, the free-space path loss formula is not valid and can result in losses smaller than one, equivalent to a signal gain. For this reason, the loss is set to unity for range values,  $R \leq \lambda/4\pi$ .

When there is relative motion between the origin and destination, the processing also introduces a frequency shift. This shift corresponds to the Doppler shift between the origin and destination. The frequency shift is  $v/\lambda$  for one-way propagation and  $2v/\lambda$  for two-way propagation. The parameter  $v$  is the relative speed of the destination with respect to the origin.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **See Also**

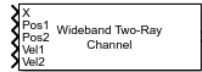
[phased.FreeSpace](#) | [twoRayChannel](#) | [widebandTwoRayChannel](#) | [Wideband Two-Ray Channel](#)

### **Introduced in R2021a**

## Wideband Two-Ray Channel

Wideband two-ray channel environment

**Library:** Radar Toolbox



### Description

The Wideband Two-Ray Channel block propagates wideband signals from one point in space to multiple points or from multiple points back to one point via both the direct path and the ground reflection path. The block propagates wideband signals by (1) decomposing them into subbands, (2) propagating subbands independently, and (3) recombining the propagated subbands. The block models propagation time, propagation loss, and Doppler shift. The block assumes that the propagation speed is much greater than the object's speed in which case the stop-and-hop model is valid.

### Ports

#### Input

##### **X** — Wideband input signal

$M$ -by- $N$  complex-valued matrix |  $M$ -by- $2N$  complex-valued matrix

- Wideband nonpolarized scalar signal, specified as an
  - $M$ -by- $N$  complex-valued matrix. The quantity  $M$  is the number of samples in the signal and  $N$  is the number of two-ray channels. Each channel corresponds to a source-destination pair. Each column contains an identical signal that is propagated along the line-of-sight and reflected paths.
  - $M$ -by- $2N$  complex-valued matrix. The quantity  $M$  is the number of samples of the signal and  $N$  is the number of two-ray channels. Each channel corresponds to a source-destination pair. Each adjacent pair of columns represents a different channel. Within each pair, the first column represents the signal propagated along the line-of-sight path and the second column represents the signal propagated along the reflected path.

The quantity  $M$  is the number of samples of the signal and  $N$  is the number of two-ray channels. Each channel corresponds to a source-destination pair.

The size of the first dimension of the input matrix can vary to simulate a changing signal length. A size change can occur, for example, in the case of a pulse waveform with variable pulse repetition frequency.

Example: `[1,1;j,1;0.5,0]`

Data Types: `double`

Complex Number Support: Yes

##### **Pos1** — Position of signal origin

3-by-1 real-valued column vector | 3-by- $N$  real-valued matrix

Origin of the signal or signals, specified as a 3-by-1 real-valued column vector or 3-by- $N$  real-valued matrix. The quantity  $N$  is the number of two-ray channels. If **Pos1** is a column vector, it takes the form  $[x; y; z]$ . If **Pos1** is a matrix, each column specifies a different signal origin and has the form  $[x; y; z]$ . Position units are in meters.

**Pos1** and **Pos2** cannot both be specified as matrices — at least one must be a 3-by-1 column vector.

Example:  $[1000; 100; 500]$

Data Types: double

### **Pos2 — Position of signal destination**

3-by-1 real-valued column vector | 3-by- $N$  real-valued matrix

Origin of the signal or signals, specified as a 3-by-1 real-valued column vector or 3-by- $N$  real-valued matrix. The quantity  $N$  is the number of two-ray channels. If **Pos2** is a column vector, it takes the form  $[x; y; z]$ . If **Pos2** is a matrix, each column specifies a different signal origin and has the form  $[x; y; z]$ . Position units are in meters.

**Pos1** and **Pos2** cannot both be specified as matrices — at least one must be a 3-by-1 column vector.

Example:  $[-100; 300; 50]$

Data Types: double

### **Vel1 — Velocity of signal origin**

3-by-1 real-valued column vector | 3-by- $N$  real-valued matrix

Velocity of signal origin, specified as a 3-by-1 real-valued column vector or 3-by- $N$  real-valued matrix. The dimensions of **Vel1** must match the dimensions of **Pos1**. If **Vel1** is a column vector, it takes the form  $[V_x; V_y; V_z]$ . If **Vel1** is a 3-by- $N$  matrix, each column specifies a different origin velocity and has the form  $[V_x; V_y; V_z]$ . Velocity units are in meters per second.

Example:  $[-10; 3; 5]$

Data Types: double

### **Vel2 — Velocity of signal destination**

3-by-1 real-valued column vector | 3-by- $N$  real-valued matrix

Velocity of signal origin, specified as a 3-by-1 real-valued column vector or 3-by- $N$  real-valued matrix. The dimensions of **Vel2** must match the dimensions of **Pos2**. If **Vel2** is a column vector, it takes the form  $[V_x; V_y; V_z]$ . If **Vel2** is a 3-by- $N$  matrix, each column specifies a different origin velocity and has the form  $[V_x; V_y; V_z]$ . Velocity units are in meters per second.

Example:  $[-1000; 300; 550]$

Data Types: double

## **Output**

### **Out — Propagated signal**

$M$ -by- $N$  complex-valued matrix |  $M$ -by- $2N$  complex-valued matrix

- $M$ -by- $N$  complex-valued matrix. To return this format, set the `CombinedRaysOutput` property to `true`. Each matrix column contains the coherently combined signals from the line-of-sight path and the reflected path.

- $M$ -by- $2N$  complex-valued matrix. To return this format set the `CombinedRaysOutput` property to `false`. Alternate columns of the matrix contain the signals from the line-of-sight path and the reflected path.

The output **Out** contains signal samples arriving at the signal destination within the current input time frame. Whenever it takes longer than the current time frame for the signal to propagate from the origin to the destination, the output may not contain all contributions from the input of the current time frame. The remaining output will appear in the next execution of the block.

## Parameters

### Signal propagation speed (m/s) — Signal propagation speed

`physconst('LightSpeed')` (default) | real-valued positive scalar

Signal propagation speed, specified as a real-valued positive scalar. The default value of the speed of light is the value returned by `physconst('LightSpeed')`. Units are in meters per second.

Example: `3e8`

Data Types: `double`

### Signal carrier frequency (Hz) — Signal carrier frequency

`300e6` (default) | positive real-valued scalar

Signal carrier frequency, specified as a positive real-valued scalar. Units are in hertz.

Data Types: `double`

### Number of subbands — Number of processing subbands

`64` (default) | positive integer

Number of processing subbands, specified as a positive integer.

Example: `128`

### Specify atmospheric parameters — Enable atmospheric attenuation model

`off` (default) | `on`

Select this parameter to enable to add signal attenuation caused by atmospheric gases, rain, fog, or clouds. When you select this parameter, the **Temperature (degrees Celsius)**, **Dry air pressure (Pa)**, **Water vapour density (g/m<sup>3</sup>)**, **Liquid water density (g/m<sup>3</sup>)**, and **Rain rate (mm/hr)** parameters appear in the dialog box.

Data Types: `Boolean`

### Temperature (degrees Celsius) — Ambient temperature

`15` (default) | real-valued scalar

Ambient temperature, specified as a real-valued scalar. Units are in degrees Celsius.

### Dependencies

To enable this parameter, select the **Specify atmospheric parameters** check box.

Data Types: `double`

### Dry air pressure (Pa) — Atmospheric dry air pressure

`101.325e3` (default) | positive real-valued scalar

Atmospheric dry air pressure, specified as a positive real-valued scalar. Units are in pascals (Pa). The default value of this parameter corresponds to one standard atmosphere.

#### Dependencies

To enable this parameter, select the **Specify atmospheric parameters** check box.

Data Types: double

#### Water vapour density (g/m<sup>3</sup>) – Atmospheric water vapor density

7.5 (default) | positive real-valued scalar

Atmospheric water vapor density, specified as a positive real-valued scalar. Units are in g/m<sup>3</sup>.

#### Dependencies

To enable this parameter, select the **Specify atmospheric parameters** check box.

Data Types: double

#### Liquid water density (g/m<sup>3</sup>) – Liquid water density

0.0 (default) | nonnegative real-valued scalar

Liquid water density of fog or clouds, specified as a nonnegative real-valued scalar. Units are in g/m<sup>3</sup>. Typical values for liquid water density are 0.05 for medium fog and 0.5 for thick fog.

#### Dependencies

To enable this parameter, select the **Specify atmospheric parameters** check box.

Data Types: double

#### Rain rate (mm/hr) – Rainfall rate

0.0 (default) | non-negative real-valued scalar

Rainfall rate, specified as a nonnegative real-valued scalar. Units are in mm/hr.

#### Dependencies

To enable this parameter, select the **Specify atmospheric parameters** check box.

Data Types: double

#### Inherit sample rate – Inherit sample rate from upstream blocks

on (default) | off

Select this parameter to inherit the sample rate from upstream blocks. Otherwise, specify the sample rate using the **Sample rate (Hz)** parameter.

Data Types: Boolean

#### Sample rate (Hz) – Sampling rate of signal

1e6 (default) | positive real-valued scalar

Specify the signal sampling rate as a positive scalar. Units are in Hz.

#### Dependencies

To enable this parameter, clear the **Inherit sample rate** check box.

Data Types: double

### Ground reflection coefficient — Ground reflection coefficient

-1 (default) | complex-valued scalar | complex-valued 1-by- $N$  row vector

Ground reflection coefficient for the field at the reflection point, specified as a complex-valued scalar or a complex-valued 1-by- $N$  row vector. Coefficients have an absolute value less than or equal to one. The quantity  $N$  is the number of two-ray channels. Units are dimensionless.

Example: -0.5

### Combine two rays at output — Option to combine two rays at output

on (default) | off

Select this parameter to combine the two rays at channel output. Combining the two rays coherently adds the line-of-sight propagated signal and the reflected path signal to form the output signal. You can use this mode when you do not need to include the directional gain of an antenna or array in your simulation.

Example: on

### Maximum one-way propagation distance (m) — Maximum one-way propagation distance

10.0e3 (default) | positive real-valued scalar

Maximum one-way propagation distance, specified as a real-valued positive scalar. Units are in meters. Any signal that propagates more than the maximum one-way distance is ignored. The maximum distance must be greater than or equal to the largest position-to-position distance.

Example: 5000.0

### Simulate using — Block simulation method

Interpreted Execution (default) | Code Generation

Block simulation, specified as Interpreted Execution or Code Generation. If you want your block to use the MATLAB interpreter, choose Interpreted Execution. If you want your block to run as compiled code, choose Code Generation. Compiled code requires time to compile but usually runs faster.

Interpreted execution is useful when you are developing and tuning a model. The block runs the underlying System object in MATLAB. You can change and execute your model quickly. When you are satisfied with your results, you can then run the block using Code Generation. Long simulations run faster with generated code than in interpreted execution. You can run repeated executions without recompiling, but if you change any block parameters, then the block automatically recompiles before execution.

This table shows how the **Simulate using** parameter affects the overall simulation behavior.

When the Simulink model is in Accelerator mode, the block mode specified using **Simulate using** overrides the simulation mode.



## Acceleration Modes

Block Simulation	Simulation Behavior		
	Normal	Accelerator	Rapid Accelerator
Interpreted Execution	The block executes using the MATLAB interpreter.	The block executes using the MATLAB interpreter.	Creates a standalone executable from the model.
Code Generation	The block is compiled.	All blocks in the model are compiled.	

For more information, see “Choosing a Simulation Mode” (Simulink).

## Algorithms

When the origin and destination are stationary relative to each other, the block output can be written as  $y(t) = x(t - \tau)/L$ . The quantity  $\tau$  is the delay and  $L$  is the propagation loss. The delay is computed from  $\tau = R/c$  where  $R$  is the propagation distance and  $c$  is the propagation speed. The free space path loss is given by

$$L_{fsp} = \frac{(4\pi R)^2}{\lambda^2},$$

where  $\lambda$  is the signal wavelength.

This formula assumes that the target is in the far-field of the transmitting element or array. In the near-field, the free-space path loss formula is not valid and can result in losses smaller than one, equivalent to a signal gain. For this reason, the loss is set to unity for range values,  $R \leq \lambda/4\pi$ .

When there is relative motion between the origin and destination, the processing also introduces a frequency shift. This shift corresponds to the Doppler shift between the origin and destination. The frequency shift is  $v/\lambda$  for one-way propagation and  $2v/\lambda$  for two-way propagation. The parameter  $v$  is the relative speed of the destination with respect to the origin.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

### Objects

`phased.FreeSpace` | `phased.LOSChannel` | `twoRayChannel` | `phased.WidebandFreeSpace` | `phased.WidebandLOSChannel` | `widebandTwoRayChannel`

### Functions

`fogpl` | `fspl` | `gaspl` | `rangeangle` | `rainpl`

### Blocks

### Topics

Two-Ray Channel

**Introduced in R2021a**

# Apps

---

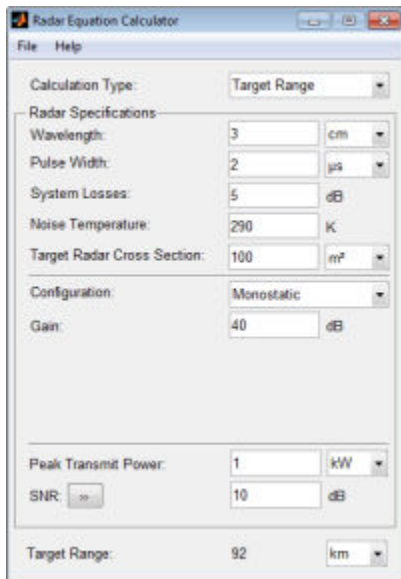
## Radar Equation Calculator

Estimate maximum range, peak power, and SNR of a radar system

### Description

The **Radar Equation Calculator** app solves the basic radar equation for monostatic or bistatic radar systems. The radar equation relates target range, transmitted power, and received signal SNR. Using this app, you can:

- Solve for maximum target range based on the transmit power of the radar and specified received SNR
- Calculate required transmit power based on known target range and specified received SNR
- Calculate the received SNR value based on known range and transmit power



### Open the Radar Equation Calculator App

- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `radarEquationCalculator`.

### Examples

#### Maximum Detection Range of a Monostatic Radar

This example shows how to compute the maximum detection range of a 10 GHz, 1 kW, monostatic radar with a 40 dB antenna gain and a detection threshold of 10 dB.

From the **Calculation Type** drop-down list, choose **Target Range** as the solution.

Choose **Configuration** as monostatic.

Enter 40 dB for the antenna **Gain**.

Set the **Wavelength** to 3 cm.

Set the **SNR** detection threshold parameter to 10 dB.

Assuming the target is a large airplane, set the **Target Radar Cross Section** value to 100 m<sup>2</sup>.

Specify the **Peak Transmit Power** as 1 kW

Specify the **Pulse Width** as 2  $\mu$ s.

Assume a total of 5 dB **System Losses**.

The screenshot shows the 'Radar Equation Calculator' window. The 'Calculation Type' is set to 'Target Range'. Under 'Radar Specifications', the inputs are: Wavelength: 3 cm, Pulse Width: 2  $\mu$ s, System Losses: 5 dB, Noise Temperature: 290 K, and Target Radar Cross Section: 100 m<sup>2</sup>. The 'Configuration' is set to 'Monostatic' and 'Gain' is 40 dB. At the bottom, 'Peak Transmit Power' is 1 kW and 'SNR' is 10 dB. The calculated 'Target Range' is 92 km.

Parameter	Value	Unit
Calculation Type	Target Range	
Wavelength	3	cm
Pulse Width	2	$\mu$ s
System Losses	5	dB
Noise Temperature	290	K
Target Radar Cross Section	100	m <sup>2</sup>
Configuration	Monostatic	
Gain	40	dB
Peak Transmit Power	1	kW
SNR	10	dB
Target Range	92	km

The maximum target detection range is 92 km.

### **Maximum Detection Range of a Monostatic Radar Using Multiple Pulses**

This example shows how to use multiple pulses to reduce the transmitted power while maintaining the same maximum target range.

Continue with the results from the previous example.

Click the arrows to the right of the **SNR** label.

The **Detection Specifications for SNR** menu opens.

Set **Probability of Detection** to 0.95.

Set **Probability of False Alarm** to  $10^{-6}$ .

Set **Number of Pulses** to 4.

Reduce **Peak Transmit Power** to 0.75 kW.

Assume a nonfluctuating target model, and set the **Swerling Case Number** to 0.

Calculation Type: Target Range

Radar Specifications

Wavelength: 3 cm

Pulse Width: 2 μs

System Losses: 5 dB

Noise Temperature: 290 K

Target Radar Cross Section: 100 m<sup>2</sup>

Configuration: Monostatic

Gain: 40 dB

Peak Transmit Power: .75 kW

SNR: << 8.741 dB

Detection Specifications for SNR

Probability of Detection: 0.95

Probability of False Alarm: 1e-06

Number of Pulses: 4

Swerling Case Number: 0

Target Range: 92.05 km

The maximum detection range is approximately the same as in the previous example, but the transmitted power is reduced by 25%.

**Maximum Detection Range of Bistatic Radar System**

This example shows how to solve for the geometric mean range of a target for a bistatic radar system.

Specify the **Calculation Type** as Target Range.

Specify the **Configuration** as bistatic.

Provide a **Transmitter Gain** and a **Receiver Gain** parameter, instead of the single gain needed in the monostatic case.

The screenshot shows the 'Radar Equation Calculator' window. The 'Calculation Type' is set to 'Target Range'. Under 'Radar Specifications', the values are: Wavelength: 0.3 m, Pulse Width: 1 μs, System Losses: 0 dB, Noise Temperature: 290 K, and Target Radar Cross Section: 1 m². The 'Configuration' is set to 'Bistatic'. The 'Transmitter Gain' is 20 dB and the 'Receiver Gain' is 20 dB. The 'Peak Transmit Power' is 1 kW and the 'SNR' is 10 dB. The 'Geometric Mean Range' is calculated as 10.32 km.

Parameter	Value	Unit
Calculation Type	Target Range	
Wavelength	0.3	m
Pulse Width	1	μs
System Losses	0	dB
Noise Temperature	290	K
Target Radar Cross Section	1	m²
Configuration	Bistatic	
Transmitter Gain	20	dB
Receiver Gain	20	dB
Peak Transmit Power	1	kW
SNR	10	dB
Geometric Mean Range	10.32	km

Alternatively, to achieve a particular probability of detection and probability of false alarm, open the **Detection Specifications for SNR** menu.

Enter values for **Probability of Detection** and **Probability of False Alarm**, **Number of Pulses**, and **Swerling Case Number**.



The screenshot shows a software window titled "Radar Equation Calculator" with a menu bar containing "File" and "Help". The interface is organized into several sections:

- Calculation Type:** A dropdown menu set to "Target Range".
- Radar Specifications:** A group box containing:
  - Wavelength: 0.3 m
  - Pulse Width: 1  $\mu$ s
  - System Losses: 0 dB
  - Noise Temperature: 290 K
  - Target Radar Cross Section: 1 m<sup>2</sup>
- Configuration:** A dropdown menu set to "Bistatic".
- Transmitter Gain:** 20 dB
- Receiver Gain:** 20 dB
- Peak Transmit Power:** 2.3 kW
- SNR:** A button with a double left arrow and a text box containing "13.5883" dB.
- Detection Specifications for SNR:**
  - Probability of Detection: 0.95
  - Probability of False Alarm: 1e-06
  - Number of Pulses: 1
  - Swerling Case Number: 0
- Geometric Mean Range:** 10.33 km

**Required Transmit Power for a Bistatic Radar**

This example shows how to compute the required peak transmit power of a 10 GHz, bistatic X-band radar for a 80 km total bistatic range, and 10 dB received SNR.

The system has a 40 dB transmitter gain and a 20 dB receiver gain. The required receiver SNR is 10 dB.

From the **Calculation Type** drop-down list, choose **Peak Transmit Power** as the solution type.

Choose **Configuration** as bistatic.

From the system specifications, set **Transmitter Gain** to 40 dB and **Receiver Gain** to 20 dB.

Set the **SNR** detection threshold to 10 dB and the **Wavelength** to 0.3 m.

Assume the target is a fighter aircraft having a **Target Radar Cross Section** value of 2 m<sup>2</sup>.

Choose **Range from Transmitter** as 50 km, and **Range from Receiver** as 30 km.

Set the **Pulse Width** to 2  $\mu$ s and the **System Losses** to 0 dB.

The screenshot shows the 'Radar Equation Calculator' window. The interface includes a menu bar with 'File' and 'Help'. The main area is divided into several sections:

- Calculation Type:** A dropdown menu set to 'Peak Transmit Power'.
- Radar Specifications:** A group box containing:
  - Wavelength:** 0.3 m
  - Pulse Width:** 2  $\mu$ s
  - System Losses:** 0 dB
  - Noise Temperature:** 290 K
  - Target Radar Cross Section:** 2 m<sup>2</sup>
- Configuration:** A dropdown menu set to 'Bistatic'.
- Transmitter Gain:** 40 dB
- Range from Transmitter:** 50 km
- Receiver Gain:** 20 dB
- Range from Receiver:** 30 km
- SNR:** A button with '>>' and a text box containing '10' dB.

At the bottom, the calculated result is displayed: **Peak Transmit Power:** 0.4966 kW.

The required Peak Transmit Power is about 0.5 kW.

### **Receiver SNR for a Monostatic Radar**

This example shows how to compute the received SNR for a monostatic radar with 1 kW peak transmit power with a target at a range of 2 km.

Assume a 2 GHz radar frequency and 20 dB antenna gain.

From the **Calculation Type** drop-down list, choose **SNR** as the solution type and set the **Configuration** as monostatic.

Set the **Gain** to 20, the **Peak Transmit Power** to 1 kW, and the **Target Range** to 2000 m.

Set the **Wavelength** to 15 cm.

Find the received SNR of a small boat having a **Target Radar Cross Section** value of 0.5 m<sup>2</sup>.

The **Pulse Width** is 1  $\mu$ s and **System Losses** are 0 dB.

Calculation Type: SNR

Radar Specifications

Wavelength: 15 cm

Pulse Width: 1  $\mu$ s

System Losses: 0 dB

Noise Temperature: 290 K

Target Radar Cross Section: 0.5 m<sup>2</sup>

Configuration: Monostatic

Gain: 20 dB

Target Range: 2000 m

Peak Transmit Power: 1 kW

SNR: 29.47 dB

- “Detection, Range and Doppler Estimation”

## Parameters

### Calculation Type – Type of calculation to perform

Target Range (default) | Peak Transmit Power | SNR

Target Range - solves for maximum target range based on transmit power of the radar and desired received SNR.

Peak Transmit – Power computes power needed to transmit based on known target range and desired received SNR.

SNR - calculates the received SNR value based on known range and transmit power.

### Wavelength – Wavelength of radar operating frequency

0.3 m (default) | m | cm | mm

Specify the wavelength of radar operating frequency in m, cm, or mm.

The wavelength is the ratio of the wave propagation speed to frequency. For electromagnetic waves, the speed of propagation is the speed of light.

Denoting the speed of light by  $c$  and the frequency (in hertz) of the wave by  $f$ , the equation for wavelength is  $\lambda = c/f$ .

#### **Pulse Width — Single pulse duration**

1  $\mu$ s (default) |  $\mu$ s | ms | s

Specify the single pulse duration in  $\mu$ s, ms, or s.

#### **System Losses — System loss in decibels (dB)**

0 dB (default)

System Losses represents a general loss factor that comprises losses incurred in the system components and in the propagation to and from the target.

#### **Noise Temperature — System noise temperature in kelvins**

290 K (default)

The system noise temperature is the product of the system temperature and the noise figure.

#### **Target Radar Cross Section — Radar cross section (RCS)**

1 m<sup>2</sup> (default) | m<sup>2</sup> | dBsm

Specify the target radar cross section in m<sup>2</sup>, or dBsm.

The target radar cross section is nonfluctuating.

#### **Configuration — Type of radar system**

Monostatic (default) | Bistatic

**Monostatic** - Transmitter and receiver are co-located (monostatic radar).

**Bistatic** - Transmitter and receiver are not co-located (bistatic radar).

#### **Gain — Transmitter and receiver gain in decibels (dB)**

20 dB (default)

When the transmitter and receiver are co-located (monostatic radar), the transmit and receive gains are equal.

This parameter is enabled only if the **Configuration** is set to Monostatic.

#### **Peak Transmit Power — Transmitter peak power**

1 kw (default) | kW | mW | W | dBW

Specify the transmitter peak power in kW, mW, W, or dBW.

This parameter is enabled only if the **Calculation Type** is set to Target Range or SNR.

#### **SNR — Minimum output signal-to-noise ratio at the receiver in decibels**

10 dB (default)

Specify an SNR value, or calculate an SNR value using Detection Specifications for SNR.

You can calculate the SNR required to achieve a particular probability of detection and probability of false alarm using Shnidman's equation. To calculate the SNR value:

- 1 Click the arrows to the right of the **SNR** label to open the Detection Specifications for SNR menu.
- 2 Enter values for Probability of Detection, Probability of False Alarm, Number of Pulses, and Swerling Case Number.

This parameter is enabled only if the **Calculation Type** is set to Target Range or Peak Transmit Power.

#### **Probability of Detection — Detection probability used to estimate SNR**

0.81029 (default)

Specify the detection probability used to estimate SNR using Shnidman's equation.

This parameter is enabled only when the **Calculation Type** is set to Peak Transmit Power or Target Range, and you select the **Detection Specifications for SNR** button for the **SNR** parameter.

#### **Probability of False Alarm — False alarm probability used to estimate SNR**

0.001 (default)

Specify the false-alarm probability used to estimate SNR using Shnidman's equation.

This parameter is enabled only when the **Calculation Type** is set to Peak Transmit Power or Target Range, and you select the **Detection Specifications for SNR** button for the **SNR** parameter.

#### **Number of Pulses — Number of pulses used to estimate SNR**

1 (default)

Specify a single pulse, or the number of pulses used for noncoherent integration in Shnidman's equation.

Use multiple pulses to reduce the transmitted power while maintaining the same maximum target range.

This parameter is enabled only when the **Calculation Type** is set to Peak Transmit Power or Target Range, and you select the **Detection Specifications for SNR** button for the **SNR** parameter.

#### **Swerling Case Number — Swerling case number used to estimate SNR**

0 (default) | 1 | 2 | 3 | 4

Specify the Swerling case number used to estimate SNR using Shnidman's equation:

- **0** - Nonfluctuating pulses.
- **1** - Scan-to-scan decorrelation. Rayleigh/exponential PDF-A number of randomly distributed scatterers with no dominant scatterer.
- **2** - Pulse-to-pulse decorrelation. Rayleigh/exponential PDF- A number of randomly distributed scatterers with no dominant scatterer.

- **3** – Scan-to-scan decorrelation. Chi-square PDF with 4 degrees of freedom. A number of scatterers with one dominant.
- **4** – Pulse-to-pulse decorrelation. Chi-square PDF with 4 degrees of freedom. A number of scatterers with one dominant.

Swerling case numbers characterize the detection problem for fluctuating pulses in terms of:

- A decorrelation model for the received pulses.
- The distribution of scatterers affecting the probability density function (PDF) of the target radar cross section (RCS).

The Swerling case numbers consider all combinations of two decorrelation models (scan-to-scan; pulse-to-pulse) and two RCS PDFs (based on the presence or absence of a dominant scatterer).

This parameter is enabled only when the **Calculation Type** is set to **Peak Transmit Power** or **Target Range**, and you select the **Detection Specifications for SNR** button for the **SNR** parameter.

#### **Target Range — Range to target**

10 km (default) | km | m | mi | nmi

Specify target range in m, km, mi, or nmi.

This parameter is enabled only when the **Calculation Type** is set to **Peak Transmit Power** or **SNR**, and the **Configuration** is set to **Monostatic**.

#### **Transmitter Gain — Transmitter gain in decibels (dB)**

20 dB (default)

When the transmitter and receiver are not co-located (bistatic radar), specify the transmitter gain separately from the receiver gain.

This parameter is enabled only if the **Configuration** is set to **Bistatic**.

#### **Range from Transmitter — Range from the transmitter to the target**

10 km (default) | km | m | mi | nmi

When the transmitter and receiver are not co-located (bistatic radar), specify the transmitter range separately from the receiver range.

You can specify range in m, km, mi, or nmi.

This parameter is enabled only when the **Calculation Type** is set to **Peak Transmit Power** or **SNR**, and the **Configuration** is set to **Bistatic**.

#### **Receiver Gain — Receiver gain in decibels (dB)**

20 dB (default)

When the transmitter and receiver are not co-located (bistatic radar), specify the receiver gain separately from the transmitter gain.

This parameter is enabled only if the **Configuration** is set to **Bistatic**.

#### **Range from Receiver — Range from the target to the receiver**

10 km (default) | km | m | mi | nmi

When the transmitter and receiver are not co-located (bistatic radar), specify the receiver range separately from the transmitter range.

You can specify range in m, km, mi, or nmi.

This parameter is enabled only when the **Calculation Type** is set to Peak Transmit Power or SNR, and the **Configuration** is set to Bistatic.

## **See Also**

### **Apps**

**Radar Designer** | **Pulse Waveform Analyzer** | **Sensor Array Analyzer**

### **Functions**

radareqpow | radareqrng | radareqsnr | shnidman

### **Topics**

“Detection, Range and Doppler Estimation”

### **Introduced in R2021a**



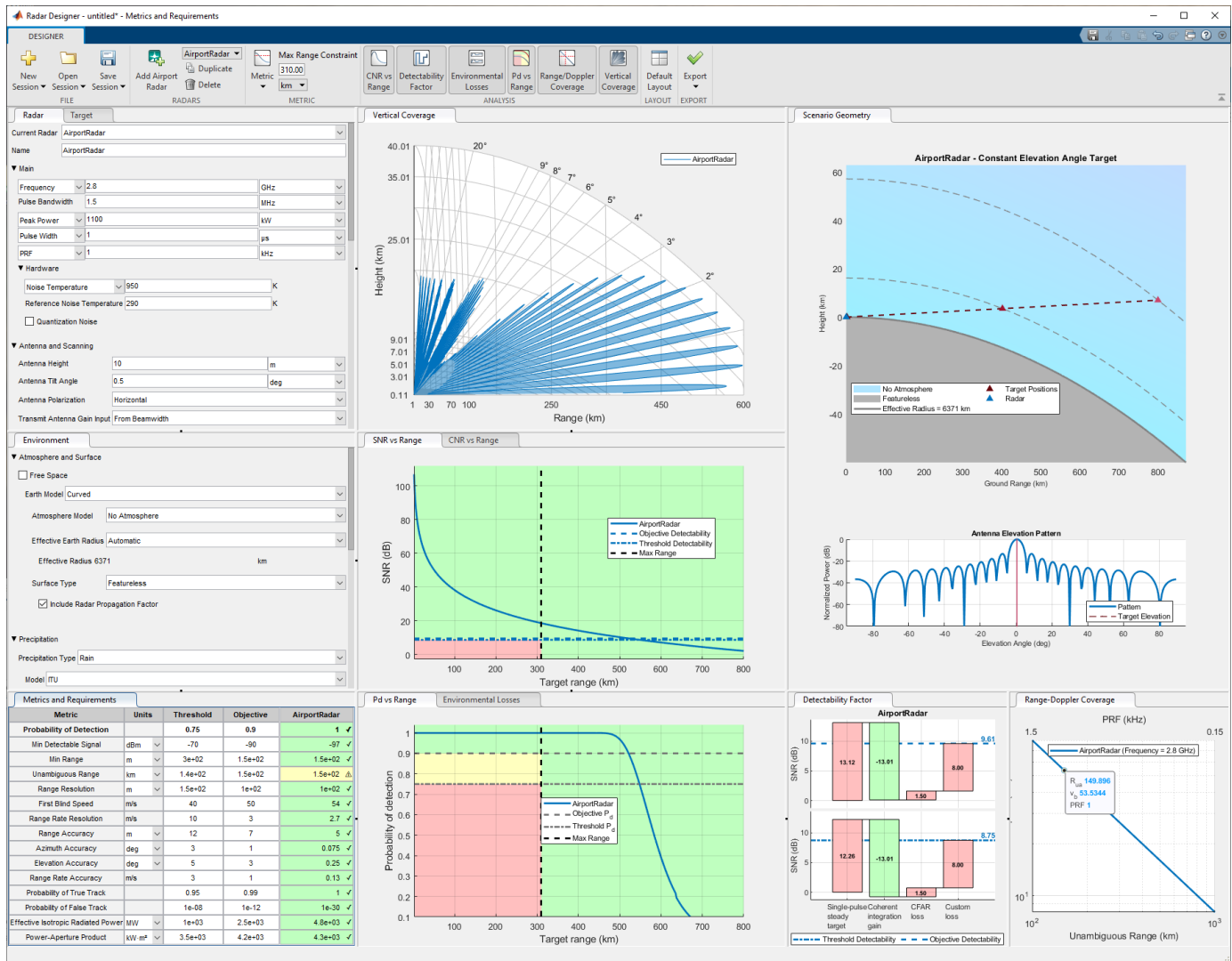
# Radar Designer

Model radar gains and losses and assess performance in different environments

## Description

The **Radar Designer** app is an interactive tool that assists engineers and system analysts with high-level design and assessment of radar systems at the early stage of radar development. Using the app, you can:

- Assess and compare multiple radar designs in a single session
- Add smart radar, environment, and target “Radar Designer Configurations” on page 3-50 to jump-start your analysis
- Incorporate environmental effects due to Earth's curvature, atmosphere, terrain, and precipitation
- Add custom target radar cross-sections, antenna/array models, and both range-independent and range-dependent losses
- Export and save results, sessions, models, and plots to continue your analysis



## Open the Radar Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `radarDesigner`.

## Examples

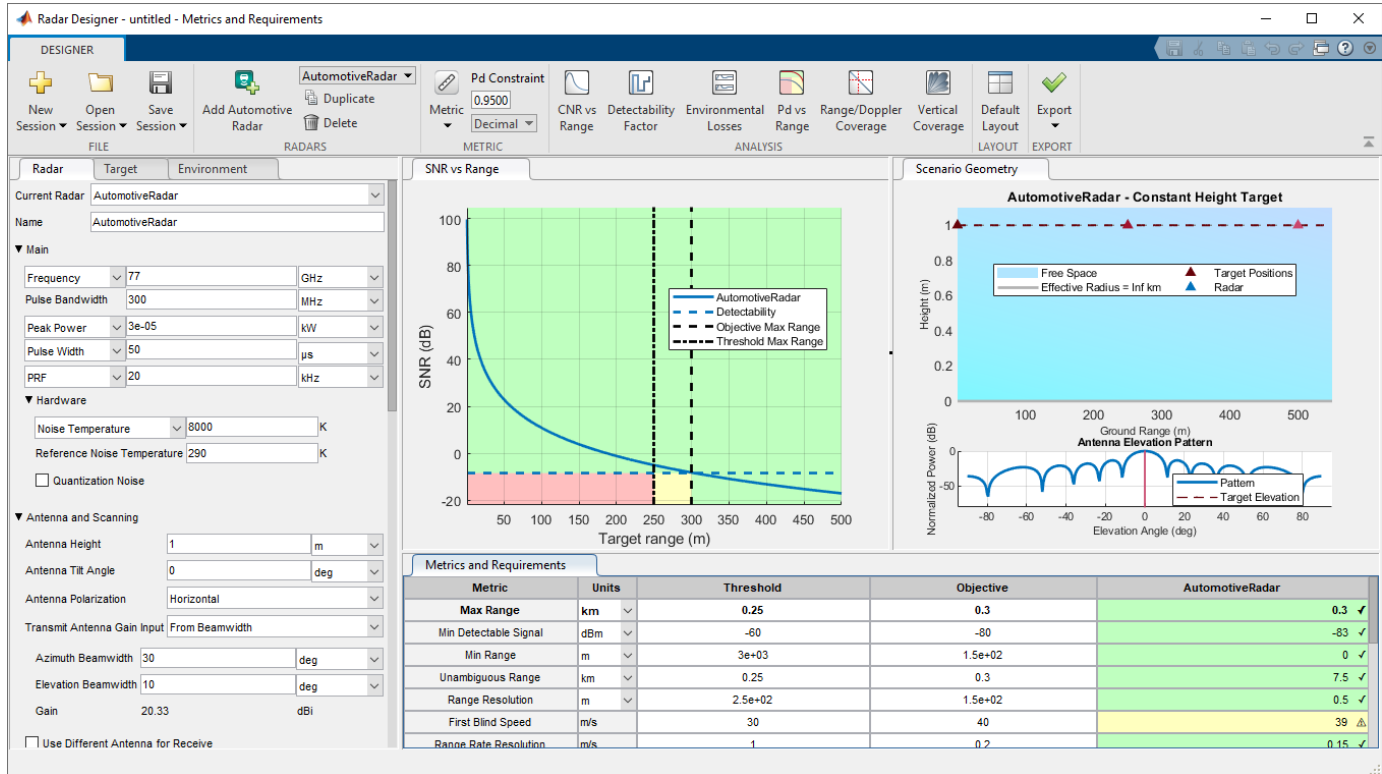
### Design Automotive Radar

Design a radar to install on top of a truck. Adjust the design parameters so the radar can work in foggy conditions and still make the objective range. Export the design session to the MATLAB Workspace.

Open **Radar Designer**. At the command line, type

## radarDesigner

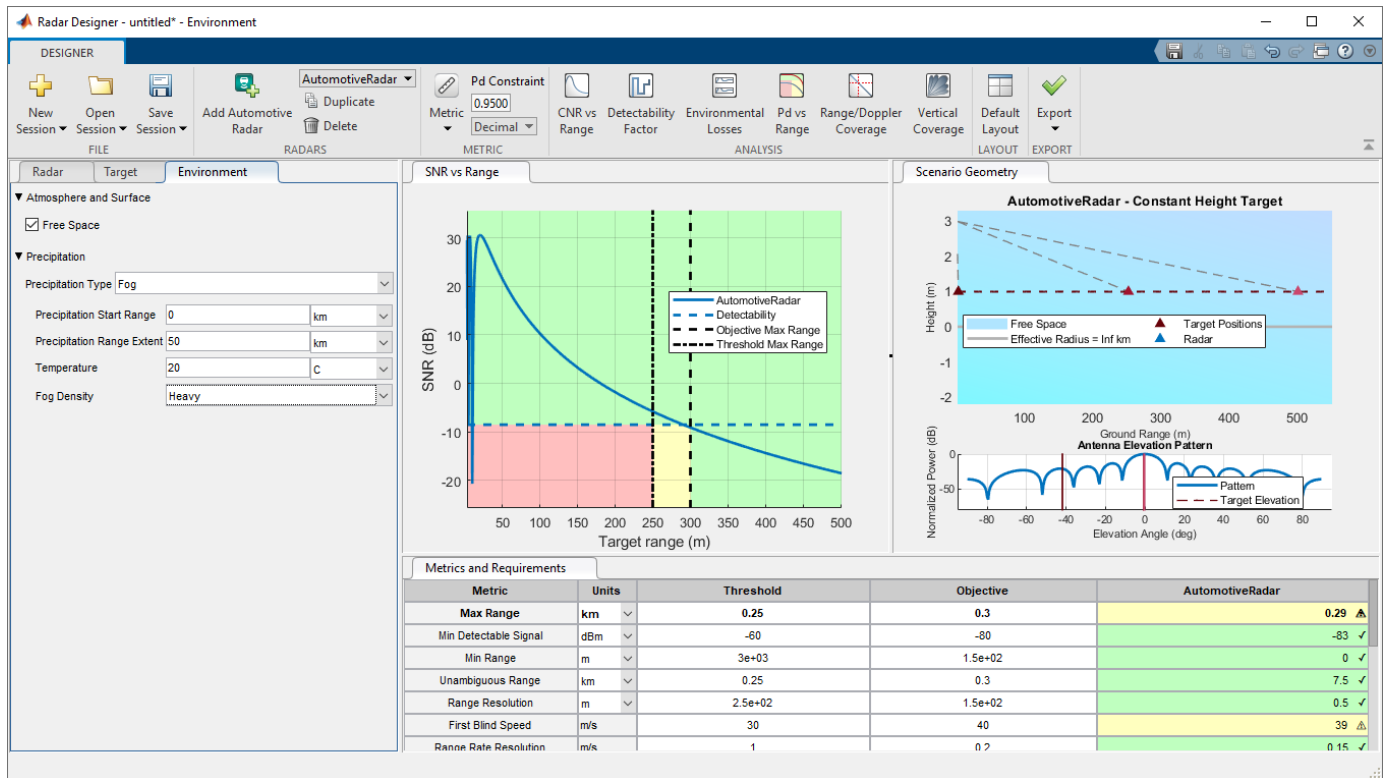
Start a radar design session. On the toolstrip, click **New Session** and select the **Automotive Radar** option. The app specifies typical radar design, target, and environment parameters.



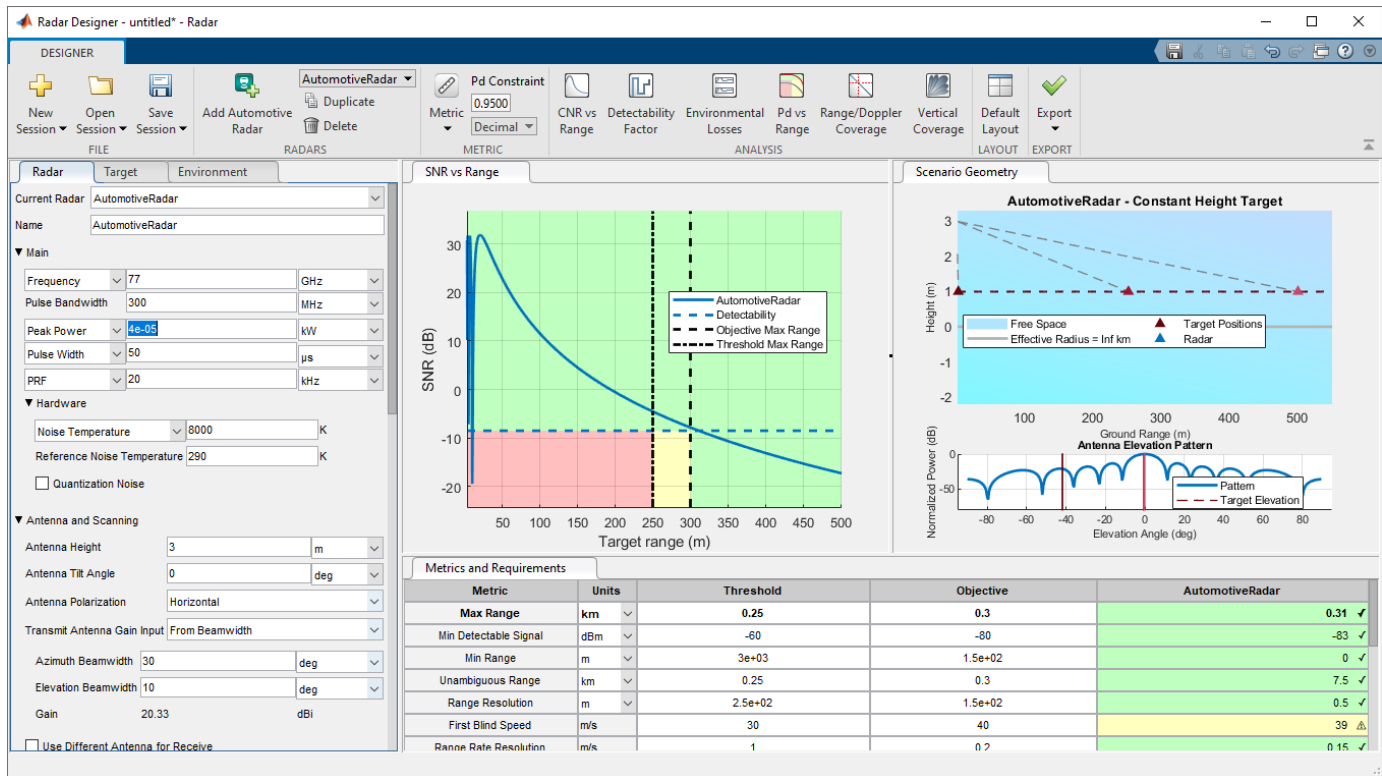
The radar you are designing must be set 3 meters above the ground. On the Radar tab, in the Antenna and Scanning section, change the **Antenna Height** from 1 meter to 3 meters.

On the Environment tab, in the Precipitation section, specify the **Precipitation Type** as Fog and set the **Fog Density** to Heavy.

As the SNR vs Range plot and Metrics and Requirements table show, the radar satisfies the threshold maximum range but falls short of the desired maximum range of 300 meters.



Increase the transmitted power to attain a higher maximum range. On the Radar tab, in the Main section, increase the Peak Power to  $4e-05$  kW. The plot and table show that the radar satisfies the requirement with the new power value.



Export the radar design to the MATLAB Workspace. On the toolstrip, click **Export** and select **Generate Metrics Report** to generate a formatted report of numeric metrics.

- “Radar Link Budget Analysis”

## Parameters

### Radar, Target, and Environment

#### Radar — Design parameters

tab

To enable the **Radar** parameters, click **New Session** on the app toolstrip to load one of the built-in “Radar Designer Configurations” on page 3-50. Use the **Radars** section of the app toolstrip to add, duplicate, or delete radar designs during a session.

- Use the **Current Radar** list to switch between different radar designs within a single session.
- Use the **Name** box to change the name of the currently selected radar.

#### Main — Pulse and carrier settings

tab section

Use these parameters to specify pulse and carrier settings, such as the carrier frequency and the transmitted power.

Parameter	Description
Carrier wave Frequency (default) or Wavelength	Carrier frequency or carrier wavelength, specified as a scalar. <ul style="list-style-type: none"> <li>Specify Frequency as a scalar in Hz, kHz, MHz, or GHz.</li> <li>Specify Wavelength as a scalar in m, cm, or mm.</li> </ul>
<b>Pulse Bandwidth</b>	Bandwidth of the transmitted pulse, specified as a scalar in Hz, kHz, MHz, or GHz.
Average Power (default) or Peak Power	Average transmitted power or peak transmitted power, specified as a scalar. <ul style="list-style-type: none"> <li>Specify Average Power as a scalar in W, kW, MW, dBw, or dBm.</li> <li>Specify Peak Power as a scalar in W, kW, MW, dBw, or dBm.</li> </ul>
Pulse Width (default) or Duty Cycle	Radar pulse width or radar duty cycle, specified as a scalar. <ul style="list-style-type: none"> <li>Specify Pulse Width, the duration of the transmitted pulse, as a scalar in s, ms, or <math>\mu</math>s.</li> <li>Specify Duty Cycle, fraction of the time the radar is transmitting, as a dimensionless scalar from 0 to 1.</li> </ul>
PRF (default) or PRI	Pulse repetition frequency (PRF) or pulse repetition interval (PRI), specified as a scalar. <ul style="list-style-type: none"> <li>Specify PRF, the number of pulses transmitted per second, as a scalar in Hz, kHz, or MHz.</li> <li>Specify PRI, the time between two consecutive transmitted pulses, as a scalar in s, ms, or <math>\mu</math>s.</li> </ul>

### Hardware — Noise settings

tab subsection

Use these parameters to specify noise settings, such as noise temperature or dynamic range.

Parameter	Description
Noise Temperature or Noise Figure	System noise temperature or noise figure, specified as a scalar. <ul style="list-style-type: none"> <li>Specify Noise Temperature as a scalar in K.</li> <li>Specify Noise Figure as a scalar in dB or in linear units.</li> </ul>
<b>Reference Noise Temperature</b>	Reference noise temperature, specified as a scalar in K.

Parameter	Description
<b>Quantization Noise</b>	Select <b>Quantization Noise</b> to include quantization noise.
<b>Number of Bits</b>	Number of bits in the analog-to-digital (A/D) converter, specified as a dimensionless scalar.  This parameter applies only if <b>Quantization Noise</b> is selected.
<b>Dynamic Range</b>	Dynamic range of the A/D converter, specified as a scalar in dB or in linear units.  This parameter applies only if <b>Quantization Noise</b> is selected.

### Antenna and Scanning — Position, beamwidth, and gain settings

tab section

Use these parameters to specify position, beamwidth, and gain settings, such as antenna height, antenna polarization, or azimuth beamwidth.

Parameter	Description
<b>Antenna Height</b>	Height of the antenna above the surface, specified as a scalar in m, km, ft, or kft.  This parameter applies to both the transmit antenna and the receive antenna.
<b>Antenna Tilt Angle</b>	Angle between the electric axis of the antenna and the ground plane, specified as a scalar in deg, rad, or mrad.  This parameter applies to both the transmit antenna and the receive antenna.
<b>Antenna Polarization</b>	Specify the antenna polarization as <b>Horizontal</b> or <b>Vertical</b> .  This parameter applies to both the transmit antenna and the receive antenna.

### Transmit Antenna Gain Input — Transmit antenna gain

tab subsection

Specify the **Transmit Antenna Gain Input** as one of these:

- **Manual** — Use the **Gain** box to enter a custom value for the transmit antenna in dBi.
- **From Beamwidth** — Compute the transmit antenna gain from the beamwidths assuming an ideal Gaussian beam pattern with no sidelobes. You can set these parameters.

Parameter	Description
<b>Azimuth Beamwidth</b>	Azimuth beamwidth of the transmit antenna, specified as a scalar in deg, rad, or mrad.

Parameter	Description
<b>Elevation Beamwidth</b>	Elevation beamwidth of the transmit antenna, specified as a scalar in deg, rad, or mrad.

**Radar Designer** computes and displays the receive antenna gain in dBi.

### Receive Antenna Gain Input – Receive antenna gain if different from transmit antenna tab subsection

Select **Use Different Antenna for Receive** to indicate that the receive and transmit antennas have different gains. If you use a different antenna for receive, you can specify the **Receive Antenna Gain Input** as one of these:

- **Manual** — Use the **Gain** box to enter a custom value for the receive antenna in dBi.
- **From Beamwidth** — Compute the receive antenna gain from the beamwidths assuming an ideal Gaussian beam pattern with no sidelobes. You can set these parameters.

Parameter	Description
<b>Azimuth Beamwidth</b>	Azimuth beamwidth of the receive antenna, specified as a scalar in deg, rad, or mrad.
<b>Elevation Beamwidth</b>	Elevation beamwidth of the receive antenna, specified as a scalar in deg, rad, or mrad.

**Radar Designer** computes and displays the receive antenna gain in dBi.

### Scan Mode – Scan mode settings

tab subsection

Specify the scan mode for your design as one of these:

- **None** — The radar performs no scanning. **Radar Designer** does not incorporate scanning-related losses into the analysis.
- **Mechanical** — The radar performs mechanical scanning. **Radar Designer** incorporates beam shape loss and beam-dwell factor (range-dependent loss for rapidly scanning beam) into the analysis.
- **Electronic** — The radar uses a phased array to perform electronic scanning. **Radar Designer** incorporates beam shape loss and scan sector loss into the analysis.

If you specify **Scan Mode** as **Mechanical** or **Electronic**, you can set these parameters.

Parameter	Description
<b>Azimuth Scan Sector Size</b>	Azimuth span of the search volume, specified as a scalar in deg, rad, or mrad.
<b>Elevation Scan Limits</b>	Initial and final elevations of the scan volume, specified as two scalars in deg, rad, or mrad.

Based on the chosen parameters, **Radar Designer** computes and displays these settings:

- **Max Scan Rate**, the maximum scan rate in degrees per second given the selected PRF, the number of transmitted pulses, and the antenna beamwidth. This setting is displayed if **Scan Mode** is specified as **Mechanical**.



- **Search Volume Size**, the size of the solid angular search volume in steradians.
- **Search Time**, the time in seconds it takes to scan the search volume given the selected PRF, the number of transmitted pulses, and the antenna beamwidth.

### Detection and Tracking — $P_{fa}$ , CPI, and $M$ -of- $N$ settings

tab section

Use these parameters to specify  $P_{fa}$ , CPI, and  $M$ -of- $N$  settings, such as probability of false alarm or track confirmation logic threshold.

Parameter	Description
<b>Probability of False Alarm</b>	Desired probability of false alarm ( $P_{fa}$ ) at the output of the detector, specified as a dimensionless scalar. The default value is $10^{-6}$ (1e-06).
<b>Number of Pulses</b>	Number of pulses within a coherent processing interval (CPI), specified as a positive integer scalar.
<b>Pulse Integration</b>	Pulse integration, specified as Coherent or Noncoherent.

### Moving Target Indicator (MTI) — Moving target indicator

tab subsection

Select **Moving Target Indicator (MTI)** to include moving target indicator processing in your design. If you enable moving target indicator processing, you can set these parameters.

Parameter	Description
<b>Canceler</b>	Canceler, specified as one of these: <ul style="list-style-type: none"> <li>• Two-pulse — First-order canceler</li> <li>• Three-pulse — Second-order canceler</li> <li>• Four-pulse — Third-order canceler</li> </ul>
<b>Null Velocity</b>	Clutter velocity to which the MTI filter is adjusted, specified as a scalar in m/s, km/hr, mi/hr, or kts.
<b>Method</b>	Method to perform MTI processing, specified as one of these: <ul style="list-style-type: none"> <li>• Sequential — <b>Radar Designer</b> processes pulses sequentially.</li> <li>• Batch — <b>Radar Designer</b> processes pulses in batches.</li> </ul>
<b>Quadrature Processing</b>	Select <b>Quadrature Processing</b> to enable quadrature-channel (vector) MTI processing for your design. If this parameter is not selected, <b>Radar Designer</b> performs single-channel MTI processing.

This option is available if **Pulse Integration** is set to Noncoherent.

**Binary Pulse Integration – Binary pulse integration**

tab subsection

Specify how to perform binary ( $M$ -of- $N$ ) pulse integration as one of these:

- **None** — **Radar Designer** does not apply binary integration.
- **Automatic** — **Radar Designer** applies binary integration and computes the optimal number of detected pulses ( $M$ ) out of the total number of pulses ( $N$ ).
- **Custom** — **Radar Designer** applies binary integration with a manually specified number of detected pulses. If you choose this option, specify the **Number of Detected Pulses** ( $M$ ) out of the total number of pulses ( $N$ ) as a positive integer.

This option is available if **Pulse Integration** is set to Noncoherent.

**Constant False Alarm Rate (CFAR) – Include constant false alarm rate detection**

tab subsection

Select **Constant False Alarm Rate (CFAR)** to enable constant false alarm rate (CFAR) detection. If you enable CFAR detection, you can set these parameters.

Parameter	Description
<b>Number of Reference Cells</b>	Total number of CFAR reference (training) cells, specified as a positive integer scalar.
<b>Method</b>	CFAR detection method, specified as one of these: <ul style="list-style-type: none"> <li>• <b>Cell Averaging</b> — <b>Radar Designer</b> sets the detection threshold by computing the average output of the surrounding range and Doppler cells.</li> <li>• <b>Greatest-of Cell Averaging</b> — <b>Radar Designer</b> sets the detection threshold by computing separate averages for leading and lagging cells and choosing the greatest value.</li> </ul>

**Number of CPIs – Number of coherent processing intervals**

tab subsection

Specify the number of coherent processing intervals (CPIs) as a positive integer scalar.

**M-of-N CPI Integration – Enable  $M$ -of- $N$  integration of CPIs**

tab subsection

Select **M-of-N CPI Integration** to enable  $M$ -of- $N$  integration of coherent processing intervals (CPIs). If you enable  $M$ -of- $N$  integration of CPIs, you can set this parameter.

Parameter	Description
<b>Number of CPIs with Detection</b>	Number of coherent processing intervals with a declared detection ( $M$ ) out of the total number of CPIs ( $N$ ), specified as a dimensionless scalar.

**Sensitivity Time Control (STC) – Sensitivity time control**

tab subsection

Select **Sensitivity Time Control** to enable sensitivity time control in your design. If you enable sensitivity time control, you can set these parameters.

Parameter	Description
<b>Cutoff Range</b>	Cutoff range beyond which the full receiver gain is used, specified as a scalar in m, km, nmi, ft, or kft. Default: 50 km.
<b>Exponent</b>	Exponent selected to maintain target detectability for ranges inside the cutoff range. Default: 3.5.

### Track Confirmation Logic – Track confirmation probabilities

tab subsection

Use the “Common Gate History Algorithm” on page 1-283 to compute track confirmation probabilities. You can set these parameters.

Parameter	Description
<b>Confirmation Threshold</b>	Confirmation threshold, specified as two positive integer scalars that represent an <i>M-of-N</i> or <i>M/N</i> confirmation logic. Default: 2/3.
Update Rate or Update Time	Update rate or update time: <ul style="list-style-type: none"> <li>Specify <b>Update Rate</b>, the number of track updates per second, as a scalar in Hz.</li> <li>Specify <b>Update Time</b>, the time interval between two consecutive track updates, as a scalar in seconds.</li> </ul> Default: 1 Hz or 1 s.

### Loss Factors – Loss factors

tab section

Use these parameters to specify loss factors.

Parameter	Description
<b>Eclipsing</b>	Eclipsing loss, specified as <b>None</b> (default), <b>Range-Dependent Factor</b> , or <b>Statistical Loss</b> .
<b>Custom Loss</b>	Custom loss, specified as a scalar in dB or linear units. Default: 4 dB.

### Target – Target characteristics

tab

To enable the **Target** parameters, add at least one radar to the app.

Parameter	Description
<b>Radar Cross Section</b>	Radar cross section, specified as a scalar in m <sup>2</sup> or dBsm.
<b>Swerling Model</b>	Swerling model, specified as Swerling 0/5, Swerling 1, Swerling 2, Swerling 3, or Swerling 4.
Height or Elevation Angle	Height or elevation angle, specified as a scalar. <ul style="list-style-type: none"> <li>Specify Height in m, km, nmi, ft, or kft.</li> <li>Specify Elevation Angle in deg, rad, or mrad.</li> </ul>
<b>Max Acceleration</b>	Maximum acceleration, specified as a scalar in m <sup>2</sup> or in units of g.

### Environment – Landscape and precipitation

tab

Use the **Environment** tab to incorporate effects due to earth's curvature, atmosphere, terrain, and precipitation.

### Atmosphere and Surface – Atmosphere and surface characteristics

tab section

Specify atmosphere and surface characteristics to use seasonal latitude models, surface, and surface clutter settings.

By default, **Radar Designer** has the **Free Space** parameter selected. This option corresponds to propagation in a vacuum, and the only variable you can control is the Precipitation. To access other options, clear the box.

### Earth Model – Earth model

tab section

Specify the **Earth Model** as Curved or Flat. Using a curved Earth model gives access to more atmosphere models and enables you to control the Effective Earth Radius.

### Atmosphere Model – Type of atmosphere

tab section

Specify the type of atmosphere through which the radar signal propagates as No Atmosphere, Uniform, Standard, Low Latitude, Mid Latitude, or High Latitude.

### No Atmosphere – No atmosphere

tab subsection

Specify No Atmosphere to use a constant index of refraction of 1. This model does not incorporate atmospheric gas loss or lens effect loss.

### Uniform – Uniform atmosphere

tab subsection

Specify **Uniform** for an atmosphere with uniform temperature, pressure, and water vapor density. This model can incorporate atmospheric gas loss but not lens effect loss. You can set these parameters.

Parameter	Description
<b>Ambient Temperature</b>	Temperature of uniform atmosphere, specified as a scalar in C or K. Default: 15 °C.
<b>Dry Air Pressure</b>	Dry air pressure of uniform atmosphere, specified as a scalar in hPa, Pa, or mbar. Default: 1013 hPa.
<b>Water Vapor Density</b>	Water vapor density of uniform atmosphere, specified as a scalar in g/m <sup>3</sup> or g/cm <sup>3</sup> . Default: 7.5 g/m <sup>3</sup> .
<b>Include Atmospheric Gases Loss</b>	Select to incorporate the path loss due to atmosphere gaseous absorption.

### Standard — ITU Mean Annual Global Reference Atmosphere

tab subsection

Specify **Standard** to use the ITU Mean Annual Global Reference Atmosphere (MAGRA) recommended in ITU-R P.835-6 [1]. This option applies only if **Earth Model** is specified as Curved. You can set these parameters.

Parameter	Description
<b>Water Vapor Density Profile</b>	Water vapor density profile, specified as <b>Automatic</b> or <b>Custom</b> . Use this parameter to use the settings recommended in ITU-R P.835-6 or to use your own settings of water vapor density and scale height.
<b>Surface Water Vapor Density</b>	Surface water vapor density, specified as a scalar in g/m <sup>3</sup> or g/cm <sup>3</sup> .  This parameter applies only if <b>Water Vapor Density Profile</b> is specified as <b>Custom</b> . The recommended value is 7.5 g/m <sup>3</sup> .
<b>Scale Height</b>	Scale height, specified as a scalar in m, km, nmi, ft, or kft.  This parameter applies only if <b>Water Vapor Density Profile</b> is specified as <b>Custom</b> . The recommended value is 2 km for typical atmospheric conditions and 6 km for dry atmospheric conditions.
<b>Include Atmospheric Gases Loss</b>	Select to incorporate the path loss due to atmosphere gaseous absorption.
<b>Include Lens Effect Loss</b>	Select to incorporate the lens effect loss due to the changing index of refraction in the atmosphere. This effect is significant only at small grazing angles.

**Low Latitude – ITU atmosphere model for latitudes less than 22 degrees**

tab subsection

Specify **Low Latitude** to use the ITU atmosphere model for latitudes less than 22° recommended in ITU-R P.835-6 [1]. This option applies only if **Earth Model** is specified as Curved. You can set these parameters.

Parameter	Description
<b>Include Atmospheric Gases Loss</b>	Select to incorporate the path loss due to atmosphere gaseous absorption.
<b>Include Lens Effect Loss</b>	Select to incorporate the lens effect loss due to the changing index of refraction in the atmosphere. This effect is significant only at small grazing angles.

**Mid Latitude – ITU atmosphere model for latitudes from 22 degrees to 45 degrees**

tab subsection

Specify **Mid Latitude** to use the ITU atmosphere model for latitudes from 22° to 45° recommended in ITU-R P.835-6 [1]. This option applies only if **Earth Model** is specified as Curved. You can set these parameters.

Parameter	Description
<b>Season</b>	Season, specified as Summer or Winter.
<b>Include Atmospheric Gases Loss</b>	Select to incorporate the path loss due to atmosphere gaseous absorption.
<b>Include Lens Effect Loss</b>	Select to incorporate the lens effect loss due to the changing index of refraction in the atmosphere. This effect is significant only at small grazing angles.

**High Latitude – ITU atmosphere model for latitudes greater than 45 degrees**

tab subsection

Specify **High Latitude** to use the ITU atmosphere model for latitudes greater than 45° recommended in ITU-R P.835-6 [1]. This option applies only if **Earth Model** is specified as Curved. You can set these parameters.

Parameter	Description
<b>Season</b>	Season, specified as Summer or Winter.
<b>Include Atmospheric Gases Loss</b>	Select to incorporate the path loss due to atmosphere gaseous absorption.
<b>Include Lens Effect Loss</b>	Select to incorporate the lens effect loss due to the changing index of refraction in the atmosphere. This effect is significant only at small grazing angles.

**Effective Earth Radius – Effective Earth radius**

tab section

Specify **Effective Earth Radius** as one of these:

- Automatic — **Radar Designer** computes the radius automatically based on the reference atmosphere.

Atmosphere Model	Effective Earth Radius
No Atmosphere	6371 km
Uniform	6371 km
Standard	8719 km
Low Latitude	9540 km
Mid Latitude	8262 km
High Latitude	8308 km

- Custom — This option is recommended for high-altitude geometries. Specify the effective radius of the Earth as a scalar in m, km, nmi, ft, or kft. This parameter is often set to 4/3 of the Earth's actual radius.

### Surface Type — Type of surface

tab section

Specify the type of surface on which the radar signal propagates as Featureless, Sea, Land, or Custom.

#### Featureless — Characteristics of perfectly smooth, perfectly reflective surface

tab subsection

If you specify the **Surface Type** as Featureless, you can set the **Propagation Factor** parameter, which is available only if you set **Earth Model** to Curved. **Propagation Factor** is off by default.

#### Sea — Sea characteristics

tab subsection

If you specify the **Surface Type** as Sea, you can set these parameters.

Parameter	Description
<b>Sea State Number</b>	<p>Sea state number, specified as one of these:</p> <ul style="list-style-type: none"> <li>• 0 - Glassy (Default) — Calm, glassy sea surface. No waves.</li> <li>• 1 - Ripples — Calm, rippled sea surface. Wave heights from 0 to 0.1 m.</li> <li>• 2 - Smooth — Smooth sea surface. Wave heights from 0.1 m to 0.5 m.</li> <li>• 3 - Slight — Slight waves. Wave heights from 0.5 m to 1.25 m.</li> <li>• 4 - Moderate — Moderate waves. Wave heights from 1.25 m to 2.5 m.</li> <li>• 5 - Rough — Rough waves. Wave heights from 2.5 m to 4 m.</li> <li>• 6 - Very Rough — Very rough waves. Wave heights from 4 m to 6 m.</li> <li>• 7 - High — High waves. Wave heights from 6 m to 9 m.</li> <li>• 8 - Very High — Very high waves. Wave heights from 9 m to 14 m.</li> </ul>
<b>Include Radar Propagation Factor</b>	<p>The radar propagation factor is the ratio of the magnitude of the actual magnetic field at a point in space to the magnitude of the magnetic field at the same point in free space.</p> <p>This parameter is available only if you set <b>Earth Model</b> to Curved. The parameter is off by default.</p>



Parameter	Description
<b>Permittivity Model</b>	<p>Permittivity model, specified as one of these:</p> <ul style="list-style-type: none"> <li>• <b>Blake's Model (Default)</b> — Blake's model is applicable in the frequency range from 100 MHz to 10 GHz.</li> <li>• <b>Sea Water</b> — ITU seawater permittivity model. Uses a temperature of 20 °C and a salinity of 35 g/kg.</li> <li>• <b>Pure Water</b> — ITU pure water permittivity model. Uses a temperature of 20 °C.</li> <li>• <b>Wet Ice</b> — ITU wet ice permittivity model. Uses a liquid water fraction of 0.5.</li> <li>• <b>Dry Ice</b> — ITU dry ice permittivity model. Uses a temperature of -10 °C</li> <li>• <b>Custom</b> — Specify a frequency-independent custom sea surface permittivity.</li> </ul> <p>This parameter applies only if <b>Include Radar Propagation Factor</b> is selected.</p>

### Land — Land characteristics

tab subsection

If you specify the **Surface Type** as Land, you can set these parameters.

<b>Land Type</b>	<p>Land type, specified as one of these:</p> <ul style="list-style-type: none"> <li>• <b>Smooth</b> — <b>Vegetation Type</b> set to None.</li> <li>• <b>Flatland (Default)</b> — <b>Vegetation Type</b> set to Thin Grass.</li> <li>• <b>Desert</b> — <b>Vegetation Type</b> set to Thin Grass.</li> <li>• <b>Farm</b> — <b>Vegetation Type</b> set to Thin Grass.</li> <li>• <b>Rolling Hills</b> — <b>Vegetation Type</b> set to Dense Brush.</li> <li>• <b>Wooded Hills</b> — <b>Vegetation Type</b> set to Dense Trees.</li> <li>• <b>Urban</b> — <b>Vegetation Type</b> set to None.</li> <li>• <b>Metropolitan</b> — <b>Vegetation Type</b> set to None.</li> <li>• <b>Mountains</b> — <b>Vegetation Type</b> set to Dense Trees.</li> <li>• <b>Rugged Mountains</b> — <b>Vegetation Type</b> set to Dense Trees.</li> </ul>

<b>Include Radar Propagation Factor</b>	<p>The radar propagation factor is the ratio of the magnitude of the actual magnetic field at a point in space to the magnitude of the magnetic field at the same point in free space.</p> <p>This parameter is available only if you set <b>Earth Model</b> to Curved. The parameter is off by default.</p>
<b>Vegetation Type</b>	<p>Vegetation type, specified as one of these:</p> <ul style="list-style-type: none"><li>• None</li><li>• Thin Grass</li><li>• Dense Weeds</li><li>• Dense Brush</li><li>• Dense Trees</li></ul> <p>This parameter applies only if <b>Include Radar Propagation Factor</b> is selected.</p>

<p><b>Permittivity Model</b></p>	<p>Permittivity model, specified as one of these:</p> <ul style="list-style-type: none"> <li>• <b>Sandy Loam (Default)</b> — Uses a default temperature of 20 °C and a water content of 0.5. Specify the temperature as a scalar in C or K and the water content as a dimensionless scalar.</li> <li>• <b>Loam</b> — Uses a default temperature of 20 °C and a water content of 0.5. Specify the temperature as a scalar in C or K and the water content as a dimensionless scalar.</li> <li>• <b>Silty Loam</b> — Uses a default temperature of 20 °C and a water content of 0.5. Specify the temperature as a scalar in C or K and the water content as a dimensionless scalar.</li> <li>• <b>Silty Clay</b> — Uses a temperature of 20 °C and a water content of 0.5. Specify the temperature as a scalar in C or K and the water content as a dimensionless scalar.</li> <li>• <b>Custom Soil</b> — Uses a default temperature of 20 °C and a water content of 0.5, and specifies these additional parameters: <ul style="list-style-type: none"> <li>• <b>Temperature</b> — Specify the temperature as a scalar in C or K. Default: 20 °C.</li> <li>• <b>Sand Percentage</b> — Specify the sand percentage as a dimensionless scalar from 0 to 100. Default: 51.52.</li> <li>• <b>Clay Percentage</b> — Specify the clay percentage as a dimensionless scalar from 0 to 100. Default: 13.42.</li> <li>• <b>Specific Gravity</b> — Specify the specific gravity as a dimensionless scalar. Default: 2.66.</li> <li>• <b>Bulk Density Model</b> — Specify Automatic to use the value chosen by <b>Radar Designer</b> or Custom to use your own value.</li> <li>• <b>Bulk Density</b> — Specify the bulk density as a scalar in g/m<sup>3</sup> or g/cm<sup>3</sup>. Default: 1.601 g/cm<sup>3</sup>.</li> </ul> <p>This parameter applies only if <b>Bulk Density Model</b> is specified as Custom.</p> </li> <li>• <b>Vegetation</b> — Uses a default temperature of 20 °C and a water content of 0.5. Specify the temperature as a scalar in C or K and the water content as a dimensionless scalar.</li> </ul>
----------------------------------	--

	<ul style="list-style-type: none"> <li>• <b>Custom</b> — Uses a default permittivity of <math>(28.5 - j11.5)</math> F/m. Specify the permittivity as a complex-valued scalar in F/m.</li> </ul> <p>This parameter applies only if <b>Include Radar Propagation Factor</b> is selected.</p>
--	--

### Custom — Custom surface

tab subsection

If you specify the **Surface Type** as Custom, you can set these parameters.

Parameter	Description
<b>Height Standard Deviation</b>	Surface height standard deviation, specified as a scalar in m, km, nmi, ft, or kft.
<b>Include Radar Propagation Factor</b>	<p>The radar propagation factor is the ratio of the magnitude of the actual magnetic field at a point in space to the magnitude of the magnetic field at the same point in free space.</p> <p>This parameter is available only if you set <b>Earth Model</b> to Curved. The parameter is off by default.</p>
<b>Slope</b>	<p>Surface slope, specified as a scalar in deg, rad, or mrad. Default: <math>3.151^\circ</math>.</p> <p>This parameter applies only if <b>Include Radar Propagation Factor</b> is selected.</p>
<b>Permittivity</b>	Surface permittivity, specified as a complex-valued scalar in F/m. Default: $(28.5 - j11.5)$ F/m.

The properties of the Custom **Surface Type** have no dependence on frequency.

### Clutter Properties — Clutter characteristics

tab section

You can specify these clutter properties.

Parameter	Description
<b>Gamma</b>	<p>Surface gamma (<math>\gamma</math>) parameter, specified as a scalar in dB or linear units.</p> <p>The <math>\gamma</math> value for a system operating at a frequency <math>f</math> is</p> $\gamma = \gamma_0 + 5 \log_{10}(f/f_0),$ <p>where <math>\gamma_0</math> is the value of <math>\gamma</math> at <math>f_0 = 10</math> GHz and is determined by measurement.</p> <p>This parameter applies only if <b>Surface Type</b> is specified as Custom.</p>

Parameter	Description
<b>Clutter Velocity Specification</b>	<p>Clutter velocity, specified as one of these:</p> <ul style="list-style-type: none"> <li>• <b>Automatic</b> — <b>Radar Designer</b> chooses values for the other parameters in this table.</li> <li>• <b>Custom</b> — You can specify the other parameters in this table.</li> </ul> <p>This parameter applies only if <b>Surface Type</b> is specified as <b>Sea</b>.</p>
<b>Polarization Dependence</b>	<p>Polarization dependence, specified as <b>Dependent</b> or <b>Independent</b>.</p> <p>This parameter applies only if <b>Surface Type</b> is specified as <b>Sea</b> and <b>Clutter Velocity Specification</b> is specified as <b>Custom</b>, or if <b>Surface Type</b> is specified as <b>Custom</b>.</p>
<b>Clutter Velocity</b>	<p>Clutter velocity, specified as a scalar in m/s, km/hr, mi/hr, or kts.</p> <p>This parameter applies only if <b>Polarization Dependence</b> is specified as <b>Independent</b>.</p>
<b>H-pol Clutter Velocity</b>	<p>Clutter velocity for horizontal polarization, specified as a scalar in m/s, km/hr, mi/hr, or kts.</p> <p>This parameter applies only if <b>Polarization Dependence</b> is specified as <b>Dependent</b>.</p>
<b>V-pol Clutter Velocity</b>	<p>Clutter velocity for vertical polarization, specified as a scalar in m/s, km/hr, mi/hr, or kts.</p> <p>This parameter applies only if <b>Polarization Dependence</b> is specified as <b>Dependent</b>.</p>
<b>Clutter Velocity Standard Deviation</b>	<p>Clutter velocity standard deviation (clutter velocity spread), specified as a scalar in m/s, km/hr, mi/hr, or kts.</p>

### Precipitation — Precipitation characteristics

tab section

Specify the **Precipitation Type** during the propagation of the radar signal as **None**, **Rain**, **Snow**, **Fog**, or **Clouds** to use rain, snow, fog, and cloud models with range settings.

### Rain — Rain characteristics

tab subsection

If you specify the **Precipitation Type** as **Rain**, you can set these parameters.

Parameter	Description
<b>Model</b>	Rain model, specified as one of these: <ul style="list-style-type: none"> <li>• ITU — Compute the path loss due to rain using the model from ITU-R P.530-17.</li> <li>• Crane — Compute the path loss due to rain using the Crane rain model.</li> </ul>
<b>Precipitation Start Range</b>	Start range of the precipitation patch, specified as a scalar in m, km, nmi, ft, or kft.
<b>Precipitation Range Extent</b>	Range extent of the precipitation patch, specified as a positive scalar in m, km, nmi, ft, or kft.
<b>Rain Rate</b>	Long-term statistical rain rate, specified as a scalar in mm/hr.
<b>Statistical Percentage</b>	Statistical Percentage, specified as a dimensionless scalar no smaller than 0.001 and no larger than 1. This parameter returns the attenuation for the specified percentage of time and applies only if <b>Model</b> is specified as ITU.

### Snow — Snow characteristics

tab subsection

If you specify the **Precipitation Type** as Snow, you can set these parameters.

Parameter	Description
<b>Precipitation Start Range</b>	Start range of the precipitation patch, specified as a scalar in m, km, nmi, ft, or kft.
<b>Precipitation Range Extent</b>	Range extent of the precipitation patch, specified as a positive scalar in m, km, nmi, ft, or kft.
<b>Snow Rate</b>	Snow rate, specified as: <ul style="list-style-type: none"> <li>• <b>Light</b> — Light snow with an equivalent liquid water content of 0.5 mm/hr</li> <li>• <b>Moderate</b> — Moderate snow with an equivalent liquid water content of 2 mm/hr</li> <li>• <b>Heavy</b> — Heavy snow with an equivalent liquid water content of 3 mm/hr</li> <li>• <b>Custom</b> — Your own equivalent liquid water content</li> </ul>
<b>Liquid Water Content</b>	Liquid water content, specified as a scalar in mm/hr. This parameter applies only if <b>Snow Rate</b> is specified as Custom. A moderate snow rate is from 1 mm/hr to 2.5 mm/hr.

**Radar Designer** uses the Gunn-East model [3] to compute snow loss.

### Fog — Fog characteristics

tab subsection

If you specify the **Precipitation Type** as Fog, you can set these parameters.

Parameter	Description
<b>Precipitation Start Range</b>	Start range of the precipitation patch, specified as a scalar in m, km, nmi, ft, or kft.
<b>Precipitation Range Extent</b>	Range extent of the precipitation patch, specified as a positive scalar in m, km, nmi, ft, or kft.
<b>Temperature</b>	Fog ambient temperature, specified as a scalar in C or K.
<b>Fog Density</b>	Fog liquid water density, specified one of these: <ul style="list-style-type: none"> <li>• <b>Moderate</b> — Moderate fog with a liquid water density of 0.5 g/m<sup>3</sup>, corresponding to a visibility of about 300 m</li> <li>• <b>Heavy</b> — Heavy fog with a liquid water density of 0.05 g/m<sup>3</sup>, corresponding to a visibility of about 50 m</li> <li>• <b>Custom</b> — Your own liquid water density</li> </ul>
<b>Liquid Water Density</b>	Liquid water density, specified as a scalar in g/m <sup>3</sup> or g/cm <sup>3</sup> . This parameter applies only if <b>Fog Density</b> is specified as Custom.

**Radar Designer** uses the ITU fog/cloud model from ITU-R P.840-6. The model is not recommended for slant path propagation.

### Clouds — Cloud characteristics

tab subsection

If you specify the **Precipitation Type** as Clouds, you can set these parameters.

Parameter	Description
<b>Precipitation Start Range</b>	Start range of the precipitation patch, specified as a scalar in m, km, nmi, ft, or kft.
<b>Precipitation Range Extent</b>	Range extent of the precipitation patch, specified as a positive scalar in m, km, nmi, ft, or kft.

Parameter	Description
<b>Cloud Type</b>	Type of clouds, specified as one of these: <ul style="list-style-type: none"> <li>• <b>Cumulus</b> (default) — Liquid water density of 1 g/m<sup>3</sup> at an altitude of 3000 ft, with average heights in the range from 1000 ft to 5000 ft</li> <li>• <b>Stratus</b> — Liquid water density of 0.29 g/m<sup>3</sup> at an altitude of 1000 ft, with average heights in the range from 0 to 2000 ft</li> <li>• <b>Stratocumulus</b> — Liquid water density of 0.15 g/m<sup>3</sup> at an altitude of 2500 ft, with average heights in the range from 1000 ft to 4000 ft</li> <li>• <b>Altostratus</b> — Liquid water density of 0.41 g/m<sup>3</sup> at an altitude of 15,000 ft, with average heights in the range from 10,000 ft to 20,000 ft</li> <li>• <b>Nimbostratus</b> — Liquid water density of 0.65 g/m<sup>3</sup> at an altitude of 5000 ft, with average heights in the range from 0 to 10,000 ft</li> <li>• <b>Cirrus</b> — Liquid water density of 0.06405 g/m<sup>3</sup> at an altitude of 30,000 ft, with average heights in the range from 20,000 ft to 40,000 ft</li> <li>• <b>Custom</b> — Liquid water density of 1 g/m<sup>3</sup> and a temperature of 9 °C</li> </ul>
<b>Liquid Water Density</b>	Liquid water density, specified as a scalar in g/m <sup>3</sup> or g/cm <sup>3</sup> . This parameter applies only if <b>Fog Density</b> is specified as Custom.



**Radar Designer** uses the ITU fog/cloud model from ITU-R P.840-6. The model is not recommended for slant path propagation.

### Performance Metrics

#### Metric — Radar equation solution and constraint

toolstrip section

Specify the quantity for which to solve the radar equation and the quantity to keep fixed when solving.

- **Probability of Detection**  — Compute probability of detection ( $P_d$ ) and other metrics with a maximum range constraint. Specify the maximum range as a scalar in m, km, nmi, ft, or kft.
- **Maximum Range**  — Compute maximum range and other metrics with a probability-of-detection ( $P_d$ ) constraint. Specify the probability of detection as a scalar in decimal units.

The chosen constraint appears at the top of the table in the **Metrics and Requirements** tab.



## Metrics and Requirements — Radar design constraints

tab

Use the **Metrics and Requirements** tab to adjust and modify the metrics required for the tradeoff analysis to obtain the desired performance and satisfy your radar design requirements. The tab uses the same color coding as a “Stoplight Chart” on page 3-53 and shows the metrics in the table.

To generate a formatted report of numeric metrics, click **Export** on the toolbar and select **Generate Metrics Report**.

Metric	Description
<b>Probability of Detection</b>	<p>Probability of detection, specified as a dimensionless scalar. This is the first entry in the table if you specify <b>Metric</b> as <b>Probability of Detection</b>.</p> <p>Given the maximum range <math>R_{\max}</math> specified in <b>Metric</b>, the probability of detection is the value <math>P_d</math> such that</p> $\text{SNR}_{\text{av}}(R_{\max}) = D_x(P_d, P_{\text{fa}}, N, \text{SW}),$ <p>where <math>\text{SNR}_{\text{av}}</math> is the “Available Signal-to-Noise Ratio” on page 3-52, <math>D_x</math> is the effective “Detectability Factor” on page 3-52, <math>P_{\text{fa}}</math> is the chosen probability of false alarm, <math>N</math> is the number of received pulses, and <math>\text{SW}</math> is the Swerling signal model.</p>
<b>Max Range</b>	<p>Maximum range, specified as a scalar in m, km, nmi, ft, or kft. This is the first entry in the table if you specify <b>Metric</b> as <b>Maximum Range</b>.</p> <p>Given the desired probability of detection <math>P_d</math> specified in <b>Metric</b>, the radar maximum range is the value <math>R_{\max}</math> such that</p> $\text{SNR}_{\text{av}}(R_{\max}) = D_x(P_d, P_{\text{fa}}, N, \text{SW}),$ <p>where <math>\text{SNR}_{\text{av}}</math> is the “Available Signal-to-Noise Ratio” on page 3-52, <math>D_x</math> is the effective “Detectability Factor” on page 3-52, <math>P_{\text{fa}}</math> is the chosen probability of false alarm, <math>N</math> is the number of received pulses, and <math>\text{SW}</math> is the Swerling signal model.</p>
<b>Min Detectable Signal</b>	<p>Minimum detectable signal, specified as a scalar in W, kW, MW, dBw, or dBm.</p> <p>The minimum detectable signal is computed using</p> $\text{MDS} = kT_sBD_x,$ <p>where <math>k</math> is Boltzmann's constant, <math>T_s</math> is the system noise temperature, <math>B</math> is the bandwidth, and <math>D_x</math> is the detectability factor.</p>

Metric	Description
<b>Min Range</b>	<p>Minimum range, specified as a scalar in m, km, nmi, ft, or kft.</p> <p>The minimum range is computed using  <math display="block">R_{\min} = c\tau/2,</math> where <math>c</math> is the speed of light and <math>\tau</math> is the pulse duration.</p>
<b>Unambiguous Range</b>	<p>Unambiguous range, specified as a scalar in m, km, nmi, ft, or kft.</p> <p>The unambiguous range is computed using  <math display="block">R_{\text{ua}} = c \times \text{PRI}/2 = c/(2 \times \text{PRF}),</math> where <math>c</math> is the speed of light, PRI is the pulse repetition interval, and PRF is the pulse repetition frequency.</p>
<b>Range Resolution</b>	<p>Range resolution, specified as a scalar in m or ft.</p> <p>The range resolution is computed using  <math display="block">\delta R = c/(2 \times B),</math> where <math>c</math> is the speed of light and <math>B</math> is the pulse bandwidth.</p>
<b>First Blind Speed</b>	<p>First blind speed, specified as a scalar in m/s.</p> <p>The maximum unambiguous radial velocity (unambiguous Doppler) is computed using  <math display="block">V_r^{\max} = \lambda \times \text{PRF}/4,</math> where <math>\lambda</math> is the radar wavelength and PRF is the pulse repetition frequency.</p>
<b>Range Rate Resolution</b>	<p>Range rate resolution, specified as a scalar in m/s.</p> <p>The range rate resolution is computed using  <math display="block">\delta V_r = \lambda \times \text{PRF}/(2N),</math> where <math>\lambda</math> is the radar wavelength, PRF is the pulse repetition frequency, and <math>N</math> is the number of received pulses.</p>
<b>Range Accuracy</b>	<p>Range accuracy, specified as a scalar in m or ft.</p> <p>The range accuracy for a linear frequency modulated (LFM) pulse is computed using  <math display="block">e_r = \sqrt{\frac{3c^2}{8\pi^2 \times \text{SNR} \times B^2} + b_r^2},</math> where <math>c</math> is the speed of light, SNR is the available signal-to-noise ratio, <math>B</math> is the pulse bandwidth, and <math>b_r^2</math> is the range bias.</p>

Metric	Description
<b>Azimuth Accuracy</b>	<p>Azimuth accuracy, specified as a scalar in deg, rad, or mrad.</p> <p>The azimuth accuracy for an <math>M</math>-element uniform linear array (ULA) is computed using</p> $e_{\theta} = \sqrt{\frac{6\theta_e^2}{4\pi^2 \times \text{SNR} \times Mk^2} + b_{\theta}^2},$ <p>where <math>\theta_e</math> is the azimuth beamwidth, SNR is the available signal-to-noise ratio, <math>k</math> is the beamwidth factor (<math>k = 0.89</math> for a ULA), and <math>b_{\theta}</math> is the azimuth bias.</p>
<b>Elevation Accuracy</b>	<p>Elevation accuracy, specified as a scalar in deg, rad, or mrad.</p> <p>The elevation accuracy for an <math>M</math>-element uniform linear array (ULA) is computed using</p> $e_{\theta} = \sqrt{\frac{6\theta_e^2}{4\pi^2 \times \text{SNR} \times Mk^2} + b_{\theta}^2},$ <p>where <math>\theta_e</math> is the elevation beamwidth, SNR is the available signal-to-noise ratio, <math>k</math> is the beamwidth factor (<math>k = 0.89</math> for a ULA), and <math>b_{\theta}</math> is the elevation bias.</p>
<b>Range Rate Accuracy</b>	<p>Range rate accuracy, specified as a scalar in m/s.</p> <p>The range rate accuracy for <math>N</math> pulses coherently processed during a coherent processing interval is computed using</p> $e_{\text{rr}} = \sqrt{\frac{6 \times \text{PRF}^2 \times \lambda^2}{4\pi^2 \times \text{SNR} \times 4N^3} + b_{\text{rr}}^2},$ <p>where PRF is the pulse repetition frequency, <math>\lambda</math> is the radar wavelength, SNR is the available signal-to-noise ratio, <math>B</math> is the pulse bandwidth, and <math>b_{\text{rr}}</math> is the range rate bias.</p>
<b>Probability of True Track</b>	<p>Probability of true track, specified as a dimensionless scalar.</p> <p>The probability of true track is computed using the common gate history algorithm. For more details, see <code>toccgh</code>.</p>

Metric	Description
<b>Probability of False Track</b>	<p>Probability of false track, specified as a dimensionless scalar.</p> <p>The probability of false track is computed using the common gate history algorithm. For more details, see <code>toccgh</code>.</p>
<b>Effective Isotropic Radiated Power</b>	<p>Effective isotropic radiated power, specified as a scalar in W, kW, MW, dBw, or dBm.</p> <p>The effective radiated power is computed using <math>ERP = P_t G_{tx}</math>, where <math>P_t</math> is the peak transmitted power and <math>G_{tx}</math> is the transmitter antenna gain.</p>
<b>Power-Aperture Product</b>	<p>Power-aperture product, specified as a scalar in <math>W \cdot m^2</math>, <math>kW \cdot m^2</math>, or <math>MW \cdot m^2</math>.</p>

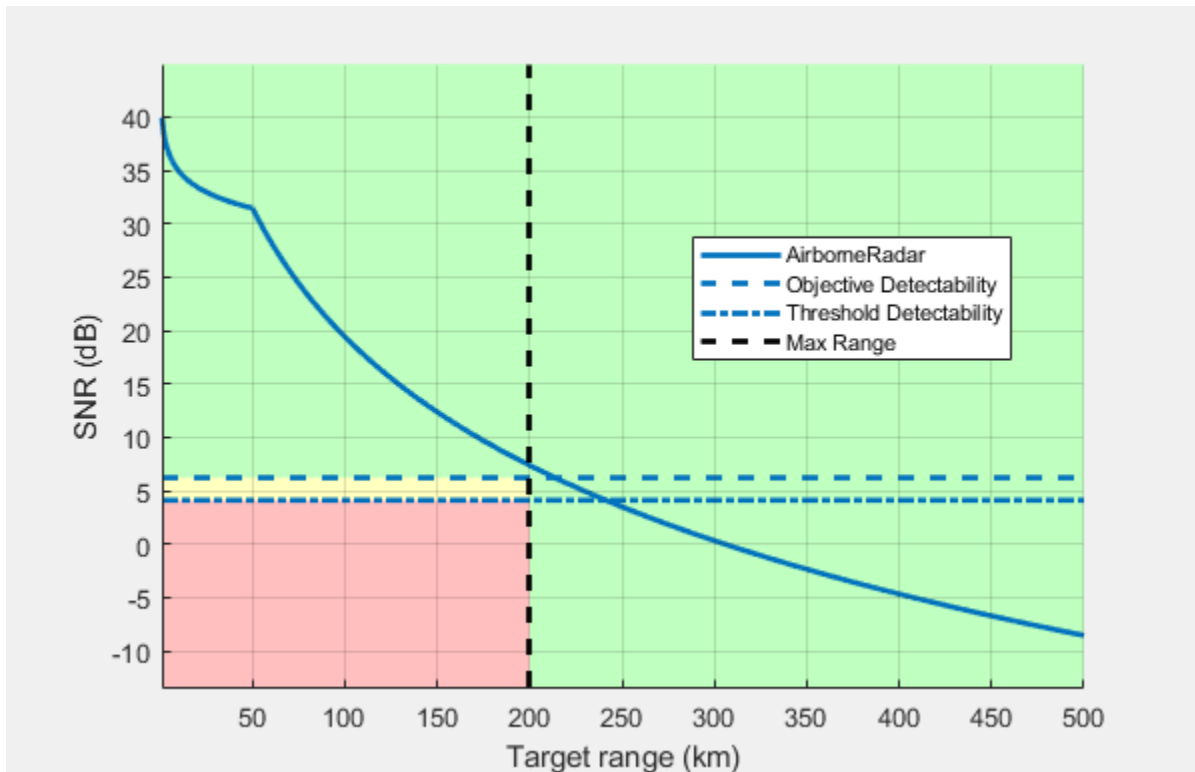
### Visualization

#### SNR vs Range — Available signal-to-noise ratio visualization

plot tab

For every radar design session, **Radar Designer** displays the “Available Signal-to-Noise Ratio” on page 3-52 (SNR) at the receiver input as a function of the target range. The plot shows the maximum range requirements and a “Stoplight Chart” on page 3-53 based on the detectability factor (required SNR) values.

This plot shows the signal-to-noise ratio plot for one airborne radar with the default settings. For more information, see “Radar Designer Configurations” on page 3-50.



To generate a script to recreate the signal-to-noise ratio plot for the currently selected radar, click **Export** on the toolstrip and select **Export SNR vs Range MATLAB Script**.

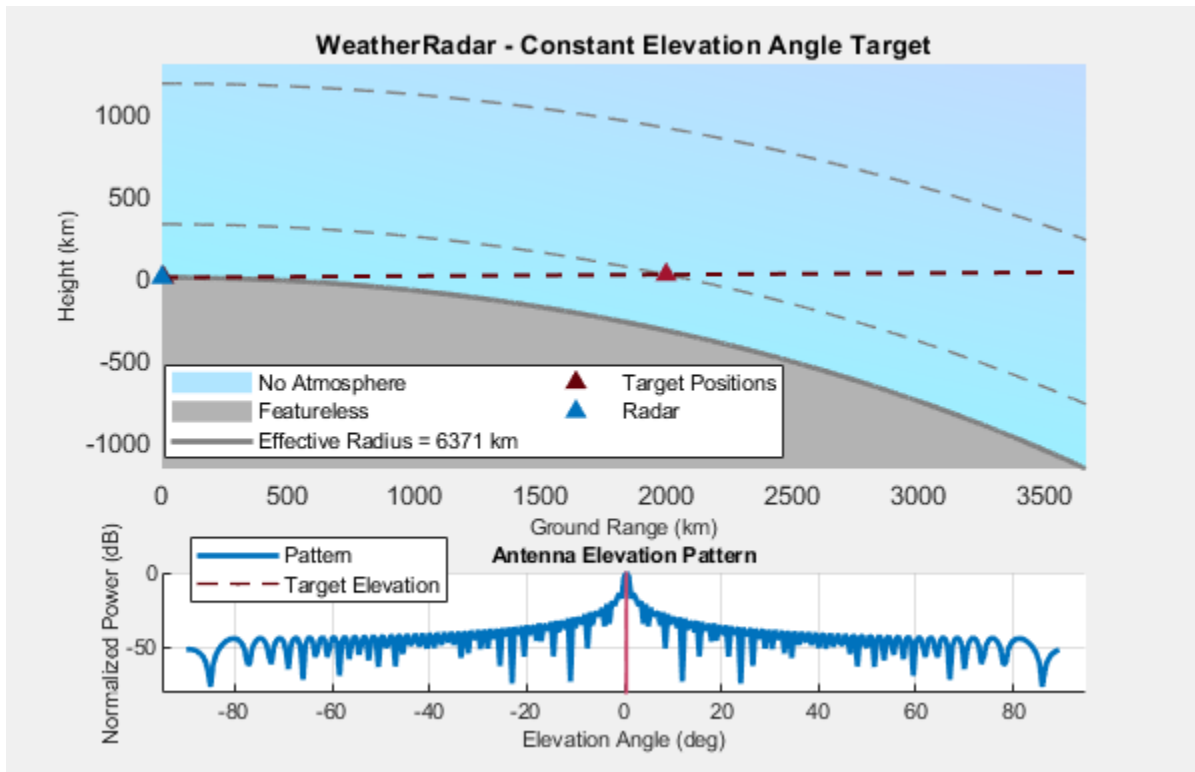
### Scenario Geometry – Geometric and environmental visualization

plot tab

For every radar design session, **Radar Designer** displays a **Scenario Geometry** tab that shows this information:

- Environment (curved Earth, flat Earth, free space)
- Radar antenna height
- Target height and position at various ranges (constant elevation or constant height)
- Radar antenna pattern demonstrating the applied tilt angle

This plot shows the scenario geometry plot for one weather radar with the default settings on a curved Earth. For more information, see “Radar Designer Configurations” on page 3-50.




### Analysis – Range/Doppler, detectability, and other plots

toolstrip button

Specify the plots to use to visualize and analyze your radar design.

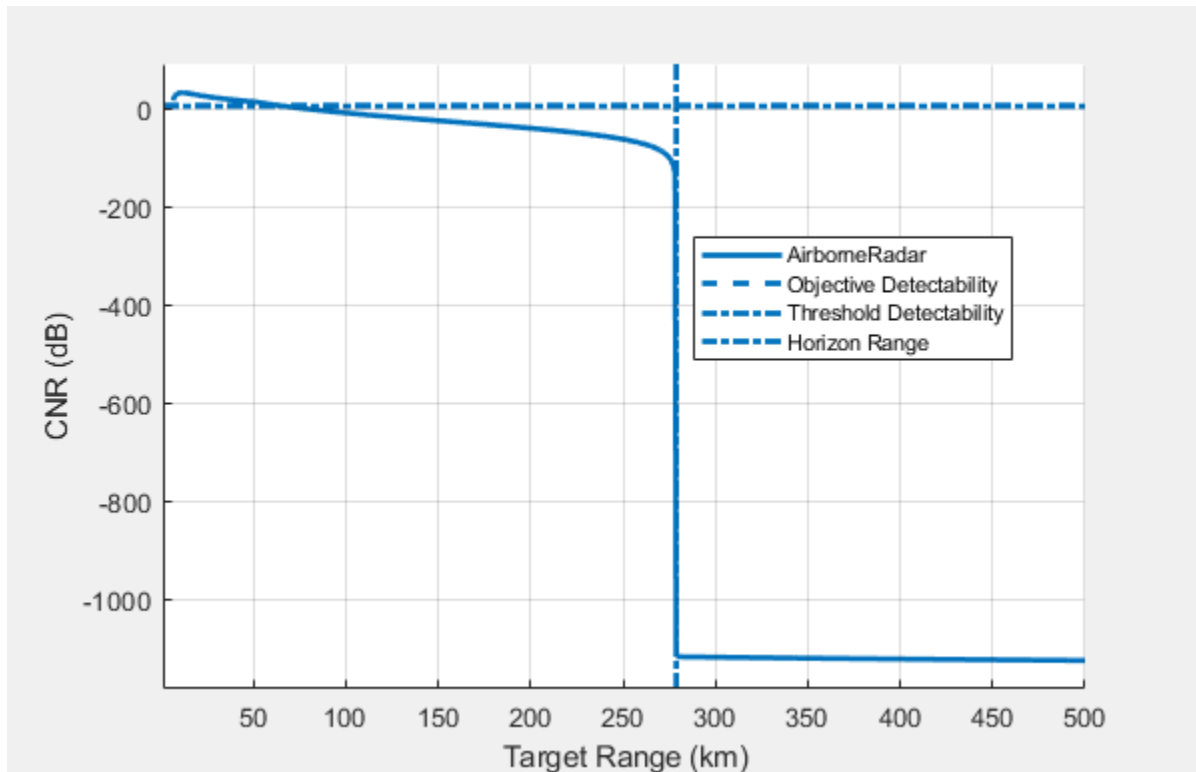
- 


**CNR vs Range**  – View clutter-to-noise ratio versus range for all designs

To visualize the clutter-to-noise ratio (CNR) as a function of range for your radar designs, click **CNR vs Range** on the toolstrip.

**Radar Designer** displays the CNR in dB and shows the horizon range.

This plot shows the clutter-to-noise ratio plot for one airborne radar with the default settings. For more information, see “Radar Designer Configurations” on page 3-50.



• **Detectability Factor**  — Inspect gains and losses of the currently selected radar

To visualize the gains and losses for your radar designs, click **Detectability Factor** on the toolstrip.

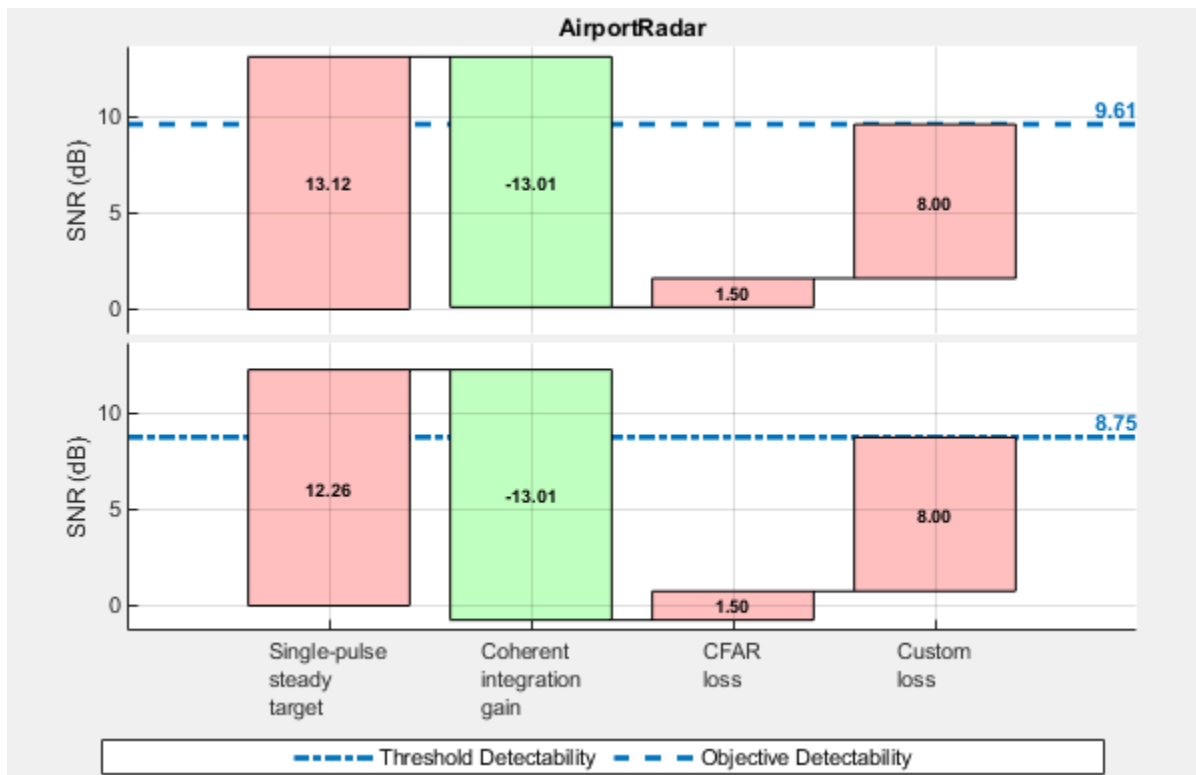
**Radar Designer** models several components of the radar signal processing chain that affect the resulting “Detectability Factor” on page 3-52. The app displays a waterfall chart that shows the individual losses and gains that contribute to increasing the required signal energy.


- The losses, represented in red, increase the required SNR threshold.
- The gains, represented in green, decrease the required SNR threshold.

Scan the plot left to right to see how the detectability factor changes as these components are added:

- Steady-target single-pulse detectability
- Integration gain
- Fluctuation loss
- Binary integration loss
- CFAR loss
- Eclipsing loss
- MTI loss
- Beam shape loss
- Scan sector loss

This plot shows the detectability factor plot for one airport radar with the default settings. For more information, see “Radar Designer Configurations” on page 3-50.



**Environmental Losses**  — View environmental losses for the currently selected radar

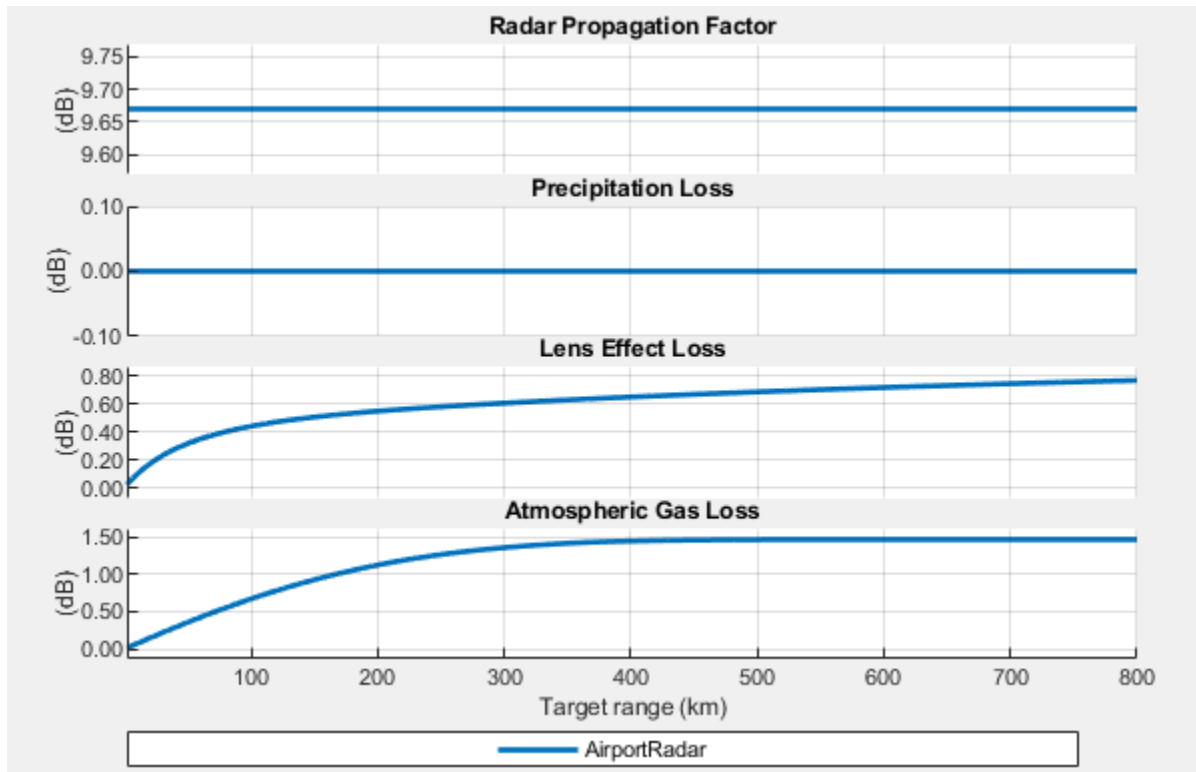
To visualize the range-dependent loss components for your radar designs in their operation environments, click **Environmental Losses** on the toolstrip.


**Radar Designer** displays four range-dependent loss components that correspond to different atmospheric and propagation effects:

- Precipitation loss
- Atmospheric gas loss
- Lens-effect loss
- Radar propagation factor

This plot shows the environmental losses plot for one airport radar with the default settings using a high-latitude atmosphere model. For more information, see “Radar Designer Configurations” on page 3-50.



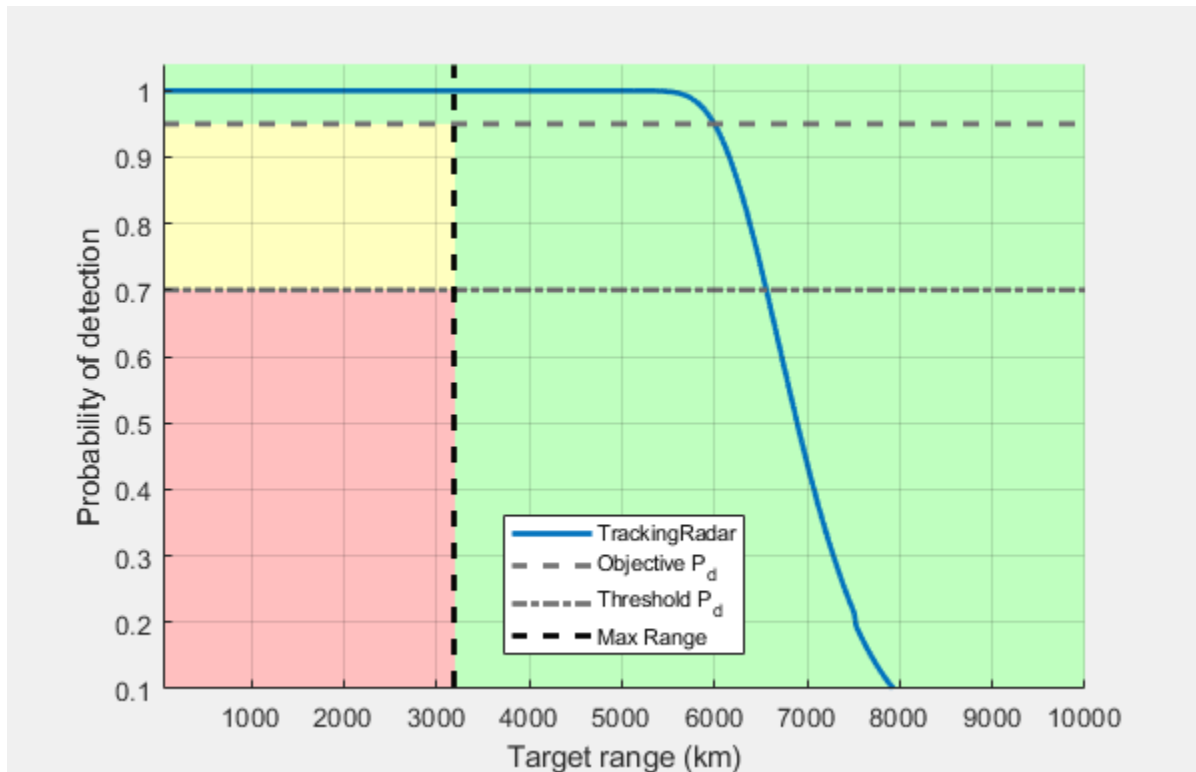



• **Pd vs Range**  — Show probability of detection ( $P_d$ ) versus range for all designs

To visualize the probability of detection as a function of range for your radar designs, click **Pd vs Range** on the toolstrip.

**Radar Designer** displays the probability of detection at the output of the receiver (effective  $P_d$ ) as a function of the target range. The plot shows the maximum range requirements and a “Stoplight Chart” on page 3-53 based on the desired  $P_d$  values.

This plot shows the probability of detection versus range plot for one tracking radar with the default settings. For more information, see “Radar Designer Configurations” on page 3-50.

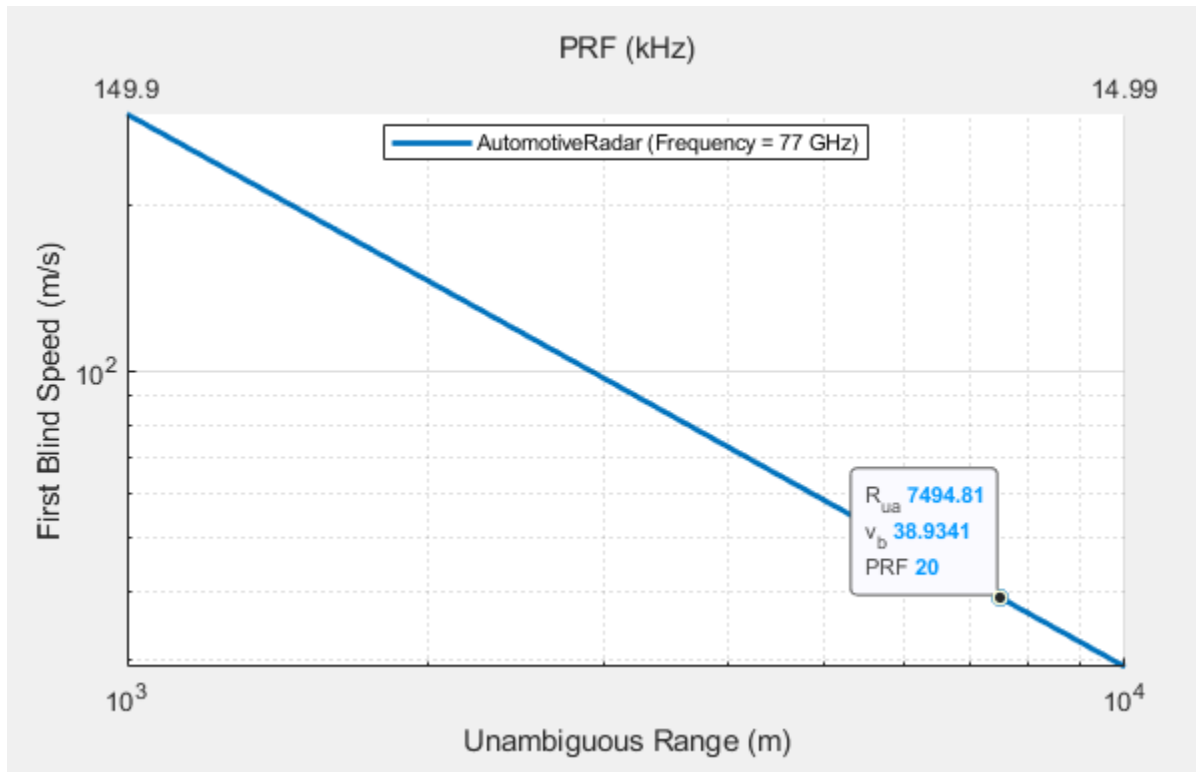



- **Range/Doppler Coverage**  — Explore range/Doppler space for the currently selected radar

To visualize the ambiguity-free range/Doppler coverage regions for your radar designs, click **Range/Doppler Coverage** on the toolstrip.

**Radar Designer** displays a log-log plot of first blind speed as a function of unambiguous range (lower x-axis) and PRF (upper x-axis). Each solid line on the plot represents a radar design. Designs with different carrier frequencies appear as parallel lines.

This plot shows the range/Doppler coverage plot for one automotive radar with the default settings. For more information, see “Radar Designer Configurations” on page 3-50.

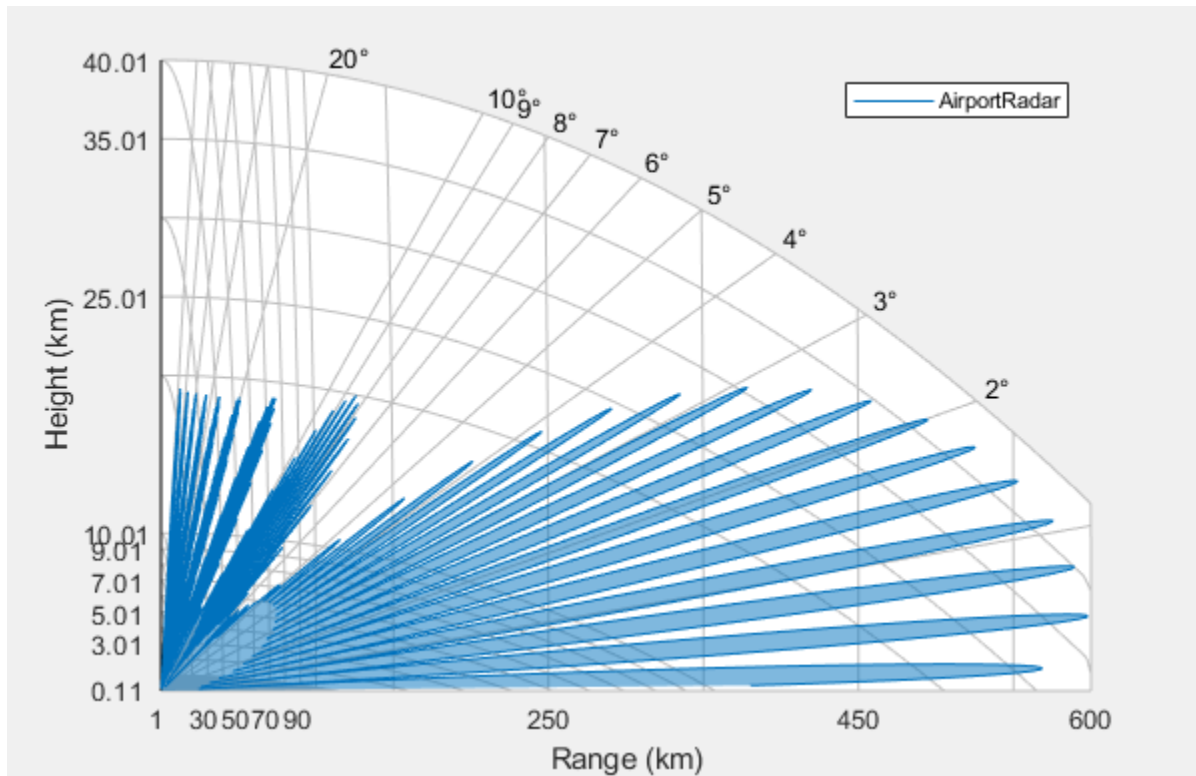


- **Vertical Coverage**  — Plot Blake chart for the currently selected radar

To visualize the range-height-angle relationships for your radar designs, click **Vertical Coverage** on the toolstrip.

**Radar Designer** displays a vertical coverage diagram of the selected radar. Vertical coverage diagrams, also known as range-height-angle charts or Blake charts, show the relationship between the range to a target, the height of the target, and the initial elevation angle of the transmitted rays for the sensor.

This plot shows the vertical coverage diagram for one airport radar with the default settings. For more information, see "Radar Designer Configurations" on page 3-50.



To generate a script to recreate the vertical coverage plot for the currently selected radar, click **Export** on the toolstrip and select **Export Vertical Coverage MATLAB Script**.

## Programmatic Use

`radarDesigner` opens the **Radar Designer** app for designing radars, targets, and environment.






`radarDesigner(sessionFileName)` opens the **Radar Designer** app and loads the specified radar file that was previously saved from the app.

## More About

### Radar Designer Configurations

**Radar Designer** includes radar configurations that enable you to switch between radar designs, duplicate radars, and delete radars.

This table shows the default parameter values for the built-in configurations.

Category	Property	Radar					
		Airborne Radar	Airport Radar	Automotive Radar	Tracking Radar	Weather Radar	
General	Icon						
	Description	Long-range airborne surveillance radar	Terminal airport surveillance	Automotive radar for use in applications such as automatic cruise control	Ground-based, cued tracking radar system	Clear air weather radar	
	Inspired By	Airborne scenario presented in [5]	ASR-9	Bosch LRR3, TI Radars	COBRA DANE	NEXRAD (VCP 32)	
Main	Frequency	450 MHz	2.8 GHz	77 GHz	1.25 GHz	2.8 GHz	
	Frequency band	UHF	S	W	L	S	
	Bandwidth	4 MHz	1.5 MHz	300 MHz	20 MHz	0.5 MHz	
	Peak power	200 kW	1.1 MW	30 mW	15 MW	500 kW	
	Pulse width	200 $\mu$ s	1 $\mu$ s	50 $\mu$ s	1 ms	1.5 $\mu$ s	
	PRF	300 Hz	1 kHz	20 kHz	1 kHz	320 Hz	
Hardware	Noise temperature	1500 K (8 dB noise figure with reference temperature of 290K)	950 K	8000 K	800 K	450 K	
Antenna and scanning	Antenna height	6096 m (20,000 ft)	10 m	1 m	75 m	20 m	
	Antenna tilt	-1°	0.5°	0	10°	0.5°	
	Polarization	Horizontal	Horizontal	Horizontal	Horizontal	Horizontal	
	Gain	From beamwidth	From beamwidth	From beamwidth	From beamwidth	From beamwidth	Manual
		Azimuth: 8°	Azimuth: 1.5°	Azimuth: 30°	Azimuth: 1°	45 dB	
		Elevation: 90°	Elevation: 5°	Elevation: 10°	Elevation: 1°		
Scan mode	Electronic	Mechanical	N/A	N/A	Mechanical		

Category	Property	Radar				
		Airborne Radar	Airport Radar	Automotive Radar	Tracking Radar	Weather Radar
		Azimuth $\pm 30^\circ$	Full $360^\circ$			Volume scan: Azimuth: Full $360^\circ$ . Elevation: $0.5^\circ$ to $5^\circ$
	Scan time	0.05 s	5 s	N/A	N/A	10 minutes
Detection	Probability of false alarm	$10^{-6}$	$10^{-6}$	$10^{-6}$	$10^{-6}$	$10^{-3}$
	Number of pulses in CPI	18	20	256	1	64
	Number of CPIs	1	1	1	1	1
Losses and other inputs	Custom loss	4 dB	8 dB	2 dB	2 dB	2 dB
	Other inputs	STC 'on' with default parameters	CFAR 'on' with default parameters	N/A	N/A	N/A
		CFAR 'on' with default parameters				
		MTI 'on' with default parameters	MTI 'on' with default parameters			
Receive gain: 10 dB						

### Available Signal-to-Noise Ratio

The available signal-to-noise ratio at a range  $R$ ,  $SNR_{av}(R)$ , is the SNR at the input to the radar receiver after the transmitted radar signal has traveled through the medium, bounced off the target, and traveled back to the radar.

The available SNR is range-dependent and can be computed from the radar equation. The available SNR depends on radar operating frequency, transmitter power, pulse width, antenna gain, system noise temperature, and also on propagation losses and factors including atmospheric losses, eclipsing effects, and so on. The available SNR tells how much energy there is available for signal detection at the receiver.

### Detectability Factor

The detectability factor or required SNR,  $D_x(P_d, P_{fa})$ , is the signal-to-noise ratio needed to detect a target with the desired probabilities of detection and false alarm.

The detectability factor is impacted by signal processing and scanning losses. Detection with the desired  $P_d$  and  $P_{fa}$  is possible when the available SNR is higher than the detectability factor. Plotting the available SNR and the detectability factor as a function of the range creates a clear image of the

radar detection performance and shows the ranges in which detection is possible and those in which it is not.

### Stoplight Chart

A radar system must meet a set of performance requirements that depend on the environment and scenarios in which the system is intended to operate. A number of such requirements can be fairly large and a design that satisfies all of them might be impractical. In this case a tradeoff analysis is applied. A subset of the requirements is satisfied at the expense of accepting lower values for the rest of the metrics. Such tradeoff analysis can be facilitated by specifying multiple requirement values for a single metric.

The requirement for each metric is specified as a pair of values:

- Objective — The desired level of the performance metric
- Threshold — The value of the metric below which the system's performance is considered unsatisfactory

The region between the Threshold and the Objective values is the trade-space. It defines a margin by which a metric can be below the Objective value while the system is still considered to have a satisfactory performance.

A stoplight chart color-codes the status of the performance metric for a radar system based on the specified requirements. The plot is divided into three zones:

- A Pass zone, colored green — At the ranges where the curve is in the Pass zone, the system performance satisfies the Objective value of the requirement.
- A Warn zone, colored yellow — At the ranges where the curve passes through the Warn zone, the system performance violates the Objective value of the specified requirement but still satisfies the Threshold value.
- A Fail zone, colored red — At the ranges where the curve passes through the Fail zone, the system performance violates the Threshold value of the specified requirement.

### Tips

- Use **Ctrl+Z** to undo a modification. Use **Ctrl+Y** to redo an undone modification.

### References

- [1] Recommendation ITU-R P835-6 (12/2017). "Reference Standard Atmospheres." Geneva: International Telecommunication Union, 2017, [https://www.itu.int/dms\\_pubrec/itu-r/rec/p/R-REC-P.835-6-201712-I!!PDF-E.pdf](https://www.itu.int/dms_pubrec/itu-r/rec/p/R-REC-P.835-6-201712-I!!PDF-E.pdf).
- [2] Barton, David K. *Radar Equations for Modern Radar*. Norwood, MA: Artech House, 2013.
- [3] Gunn, K. L. S., and T. W. R. East. "The Microwave Properties of Precipitation Particles." *Quarterly Journal of the Royal Meteorological Society* 80, no. 346 (October 1954): 522-45. <https://doi.org/10.1002/qj.49708034603>.
- [4] O'Donnell, R. M. "Radar Systems Engineering." IEEE AES Society, IEEE New Hampshire Section, Radar Systems Course, January 2010.

[5] Ward, J. "Space-Time Adaptive Processing for Airborne Radar." TR-1015, MIT Lincoln Laboratory, December 1994. <https://apps.dtic.mil/dtic/tr/fulltext/u2/a293032.pdf>

[6] Wasson, Charles S. *System Engineering Analysis, Design, and Development: Concepts, Principles, and Practices*. Second edition. Wiley Series in Systems Engineering and Management. Hoboken, New Jersey: John Wiley & Sons Inc, 2016.

## **See Also**

### **Apps**

**Radar Equation Calculator | Pulse Waveform Analyzer | Sensor Array Analyzer**

### **Functions**

radareqpow | radareqrng | radareqsnr | radarmetricplot

### **Topics**

"Radar Link Budget Analysis"

**Introduced in R2021a**



# Objects

---

## Platform

Platform object belonging to radar scenario

## Description

Platform defines a platform object belonging to a radar scenario.

## Creation

You can create Platform objects using the `platform` function of the `radarScenario` object.

## Properties

### PlatformID — Scenario-defined platform identifier

positive integer

This property is read-only.

Scenario-defined platform identifier, specified as a positive integer. The scenario automatically assigns PlatformID values to each platform, starting with 1 for the first platform and incrementing by 1 for each new platform.

Data Types: double

### ClassID — Platform classification identifier

0 (default) | nonnegative integer

Platform classification identifier, specified as a nonnegative integer. You can define your own platform classification scheme and assign ClassID values to platforms according to the scheme. The value of 0 is reserved for an object of unknown or unassigned class.

Example: 5

Data Types: double | single

### Position — Current position of platform

three-element numeric vector

This property is read-only.

Current position of the platform, specified as a three-element numeric vector.

- When the `IsEarthCentered` property of the scenario is set to `false`, the position is expressed as Cartesian coordinates `[x, y, z]` in meters.
- When the `IsEarthCentered` property of the scenario is set to `true`, the position is expressed as geodetic coordinates `[latitude, longitude, altitude]`, where `latitude` and `longitude` are in degrees and `altitude` is in meters.

Data Types: double

**Orientation – Current orientation of platform**

three-element numeric vector

This property is read-only.

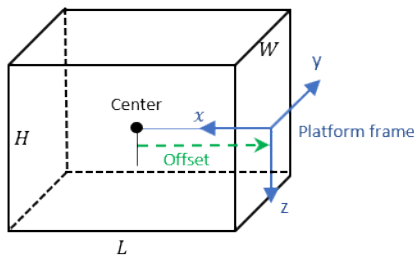
Current orientation of the platform, specified as a three-element numeric vector in degrees. The orientation is expressed as [yaw, pitch, roll] rotation angles from the local reference frame to the body frame of the platform.

Data Types: double

**Dimensions – Platform dimensions and origin offset**

structure

Platform dimensions and origin offset, specified as a structure. The structure contains the Length, Width, Height, and OriginOffset of a cuboid that approximates the dimensions of the platform. The OriginOffset is the position vector from the center of the cuboid to the origin of the platform coordinate frame. The OriginOffset is expressed in the platform coordinate system. For example, if the platform origin is at the center of the cuboid rear face as shown in the figure, then set OriginOffset as  $[-L/2, 0, 0]$ . The default value for Dimensions is a structure with all fields set to zero, which corresponds to a point model.

**Fields of Dimensions**

Fields	Description	Default
Length	Dimension of a cuboid along the x direction	0
Width	Dimension of a cuboid along the y direction	0
Height	Dimension of a cuboid along the z direction	0
OriginOffset	Position of the platform coordinate frame origin with respect to the cuboid center	[0 0 0 ]

Example: `struct('Length',5,'Width',2.5,'Height',3.5,'OriginOffset',[-2.5 0 0])`

Data Types: struct

**Trajectory – Platform motion**

kinematicTrajectory object | waypointTrajectory object | geoTrajectory object

Platform motion, specified as a kinematicTrajectory object, a waypointTrajectory object, or a geoTrajectory object. The trajectory object defines the time evolution of the position and velocity

of the platform frame origin, as well as the orientation of the platform frame relative to the scenario frame.

- When the `IsEarthCentered` property of the scenario is set to `false`, use the `kinematicTrajectory` or the `waypointTrajectory` object. By default, the platform uses a stationary `kinematicTrajectory` object.
- When the `IsEarthCentered` property of the scenario is set to `true`, use the `geoTrajectory` object. By default, the platform uses a stationary `geoTrajectory` object.

### **Signatures – Platform signatures**

cell array of signature objects | `{}`

Platform signatures, specified as a cell array of signature objects or an empty cell array (`{}`). The default value is a cell array containing an `rcsSignature` object with default property values. If you have Sensor Fusion and Tracking Toolbox, then the cell array can also include `irSignature` and `tsSignature` objects. The cell array contains at most one instance of each type of signature object. A signature represents the reflection or emission pattern of a platform, such as its radar cross-section, target strength, or IR intensity.

### **PoseEstimator – Platform pose estimator**

`insSensor` object (default) | pose estimator object

Platform pose estimator, specified as a pose-estimator object such as an `insSensor` object. The pose estimator determines the platform pose with respect to the local NED scenario coordinates. The interface of any pose estimator must match the interface of the `insSensor` object. By default, the pose-estimator accuracy properties are zero.

### **Emitters – Emitters mounted on platform**

cell array of emitter objects

Emitters mounted on the platform, specified as a cell array of emitter objects such as `radarEmitter` objects. If you have Sensor Fusion and Tracking Toolbox, then the cell array can also include `sonarEmitter` objects.

### **Sensors – Sensors mounted on platform**

cell array of sensor objects

Sensors mounted on the platform, specified as a cell array of sensor objects such as `radarDataGenerator` objects.

## **Object Functions**

<code>detect</code>	Collect detections from all sensors mounted on platform
<code>emit</code>	Collect emissions from all emitters mounted on platform
<code>pose</code>	Update pose for platform
<code>receive</code>	Receive IQ signal from radars mounted on platform
<code>targetPoses</code>	Target positions and orientations as seen from platform

## **Examples**

### **Create Radar Scenario with Two Platforms**

Create a radar scenario with two platforms that follow different trajectories.

```
sc = radarScenario('UpdateRate',100,'StopTime',1.2);
```

Create two platforms.

```
platfm1 = platform(sc);
platfm2 = platform(sc);
```

Platform 1 follows a circular path of radius 10 m for one second. This is accomplished by placing waypoints in a circular shape, ensuring that the first and last waypoint are the same.

```
wpts1 = [0 10 0; 10 0 0; 0 -10 0; -10 0 0; 0 10 0];
time1 = [0; 0.25; .5; .75; 1.0];
platfm1.Trajectory = waypointTrajectory(wpts1,time1);
```

Platform 2 follows a straight path for one second.

```
wpts2 = [-8 -8 0; 10 10 0];
time2 = [0; 1.0];
platfm2.Trajectory = waypointTrajectory(wpts2,time2);
```

Verify the number of platforms in the scenario.

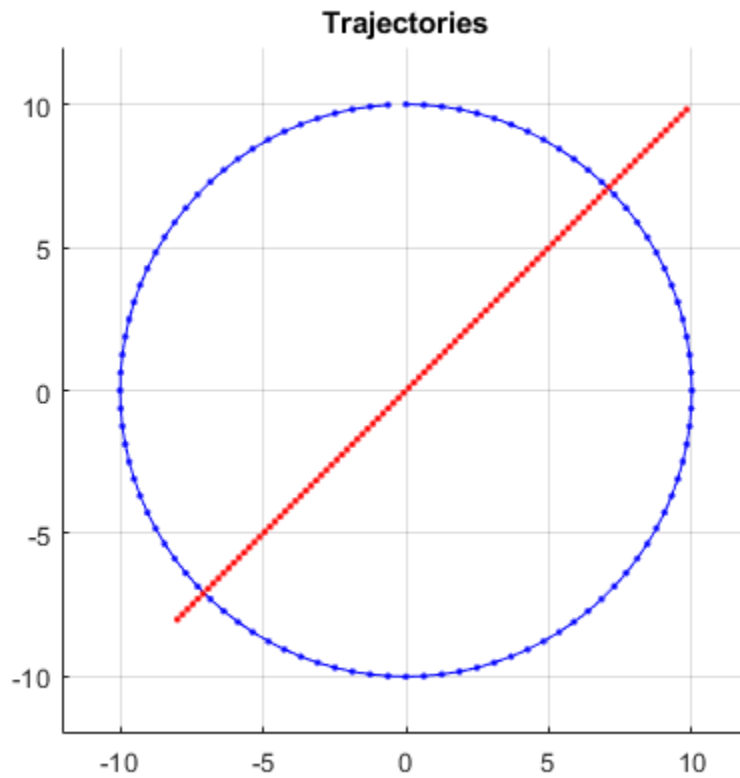
```
disp(sc.Platforms)

    {1x1 radar.scenario.Platform}    {1x1 radar.scenario.Platform}
```

Run the simulation and plot the current position of each platform using an animated line.

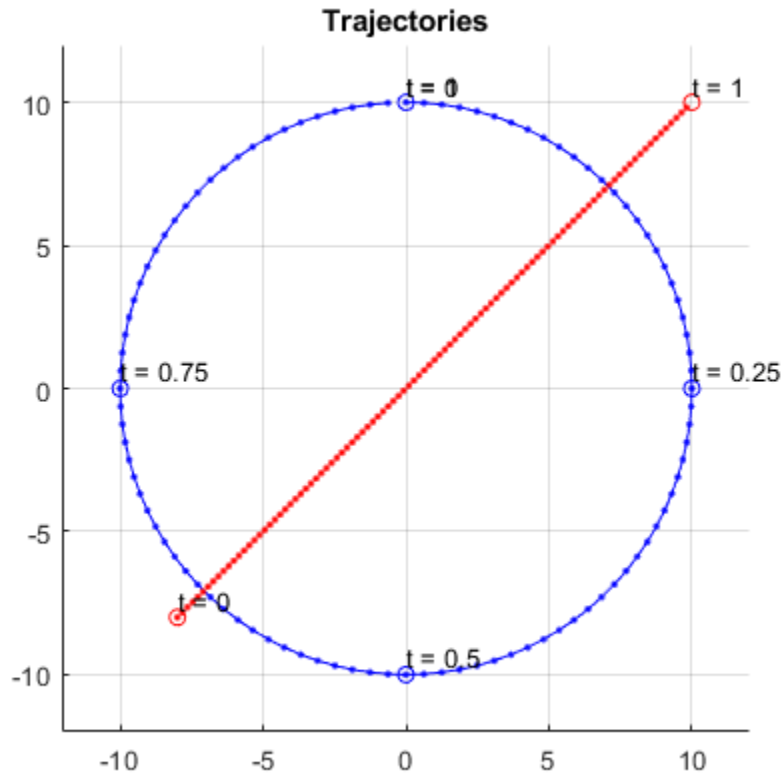
```
figure
grid
axis equal
axis([-12 12 -12 12])
line1 = animatedline('DisplayName','Trajectory 1','Color','b','Marker','.');
line2 = animatedline('DisplayName','Trajectory 2','Color','r','Marker','.');
title('Trajectories')
p1 = pose(platfm1);
p2 = pose(platfm2);
addpoints(line1,p1.Position(1),p1.Position(2));
addpoints(line2,p2.Position(2),p2.Position(2));

while advance(sc)
    p1 = pose(platfm1);
    p2 = pose(platfm2);
    addpoints(line1,p1.Position(1),p1.Position(2));
    addpoints(line2,p2.Position(2),p2.Position(2));
    pause(0.1)
end
```



Plot the waypoints for both platforms.

```
hold on
plot(wpts1(:,1),wpts1(:,2),'ob')
text(wpts1(:,1),wpts1(:,2),"t = " + string(time1),'HorizontalAlignment','left','VerticalAlignment','top')
plot(wpts2(:,1),wpts2(:,2),'or')
text(wpts2(:,1),wpts2(:,2),"t = " + string(time2),'HorizontalAlignment','left','VerticalAlignment','top')
hold off
```



### Create Cuboid Platforms with Circular Trajectory

Create a radar scenario.

```
rs = radarScenario;
```

Create a cuboid platform for a truck with dimensions 5 m by 2.5 m by 3.5 m.

```
dim1 = struct('Length',5,'Width',2.5,'Height',3.5,'OriginOffset',[0 0 0]);
truck = platform(rs,'Dimension',dim1);
```

Specify the trajectory of the truck as a circle with radius 20 m.

```
truck.Trajectory = waypointTrajectory('Waypoints', ...
    [20*cos(2*pi*(0:10)/10) 20*sin(2*pi*(0:10)/10) -1.75*ones(11,1)], ...
    'TimeOfArrival',linspace(0,50,11)');
```

Create the platform for a small quadcopter with dimensions 0.3 m by 0.3 m by 0.1 m.

```
dim2 = struct('Length',.3,'Width',.3,'Height',.1,'OriginOffset',[0 0 0]);
quad = platform(rs,'Dimension',dim2);
```

Specify the trajectory of the quadcopter as a circle 10 m above the truck with a small angular delay. Note that the negative z coordinates correspond to positive elevation.

```
quad.Trajectory = waypointTrajectory('Waypoints', ...
    [20*cos(2*pi*((0:10)'-.6)/10) 20*sin(2*pi*((0:10)'-.6)/10) -11.80*ones(11,1)], ...
    'TimeOfArrival', linspace(0,50,11)');
```

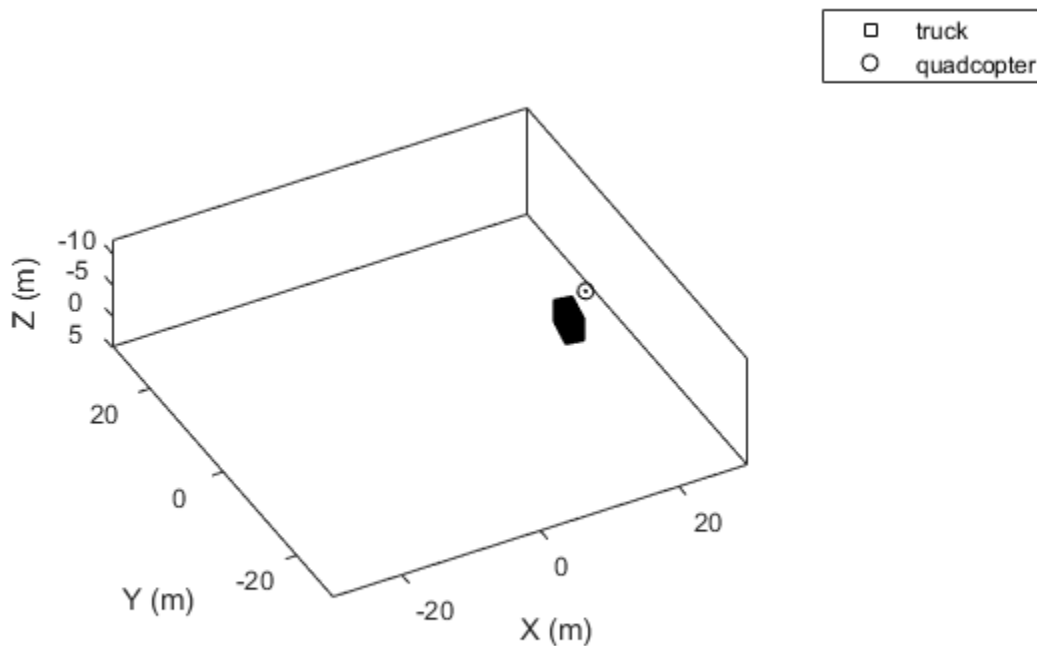
Visualize the results using theaterPlot.

```
tp = theaterPlot('XLim',[-30 30],'YLim',[-30 30],'Zlim',[-12 5]);
pp1 = platformPlotter(tp,'DisplayName','truck','Marker','s');
pp2 = platformPlotter(tp,'DisplayName','quadcopter','Marker','o');
```

Specify a view direction and run the simulation.

```
view(-28,37);
set(gca,'Zdir','reverse');
```

```
while advance(rs)
    poses = platformPoses(rs);
    plotPlatform(pp1,poses(1).Position,truck.Dimensions,poses(1).Orientation);
    plotPlatform(pp2,poses(2).Position,quad.Dimensions,poses(2).Orientation);
end
```



## See Also

### Classes

rscSignature



**Objects**

waypointTrajectory | kinematicTrajectory | geoTrajectory | insSensor | radarEmitter  
| radarDataGenerator

**Introduced in R2021a**

## detect

**Package:** radar.scenario

Collect detections from all sensors mounted on platform

### Syntax

```
detections = detect(plat,time)
detections = detect(plat,signals,time)
detections = detect(plat,signals,emitterConfigs,time)
[detections,numDets] = detect( ___ )
[detections,numDets,sensorConfigs] = detect( ___ )
```

### Description

`detections = detect(plat,time)` reports the detections from all sensors mounted on the platform, `plat`, at the specified `time`. Use this syntax when none of the sensors require information on signals present in the scenario.

`detections = detect(plat,signals,time)` also specifies any signals, `signals`, present in the scenario. Use this syntax when sensors require information on the signals.

`detections = detect(plat,signals,emitterConfigs,time)` also specifies emitter configurations, `emitterConfigs`. Use this syntax when sensors require information on the configurations of emitters generating signals in the scenario.

`[detections,numDets] = detect( ___ )` also returns the number of detections, `numDets`. This output argument can be used with any of the previous syntaxes.

`[detections,numDets,sensorConfigs] = detect( ___ )` also returns all sensor configurations, `sensorConfigs`. This output argument can be used with any of the previous syntaxes.

### Input Arguments

#### **plat** – Scenario platform

Platform object

Scenario platform, specified as a `Platform` object. To create platforms, use the `platform` function.

#### **time** – Simulation time

0 (default) | positive scalar

Simulation time, specified as a positive scalar.

Example: 1.5

Data Types: `single` | `double`

#### **signals** – Signal emissions

cell array of signal emission objects

Signal emissions, specified as a cell array of signal emission objects such as `radarEmission` objects.

### **emitterConfigs** — Emitter configurations

array of emitter configuration structures

Emitter configurations, specified as an array of emitter configuration structures. Each structure has these fields.

Field	Description
<code>EmitterIndex</code>	Unique emitter index, returned as a positive integer.
<code>IsValidTime</code>	Valid emission time, returned as 0 or 1. <code>IsValidTime</code> is 0 when emitter updates are requested at times that are between update intervals specified by the <code>UpdateInterval</code> property.
<code>IsScanDone</code>	Whether the emitter has completed a scan, returned as <code>true</code> or <code>false</code> .
<code>FieldOfView</code>	Field of view of the emitter, returned as a two-element vector [azimuth; elevation] in degrees.
<code>MeasurementParameters</code>	Emitter measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.

## Output Arguments

### **detections** — Sensor detections

cell array of `objectDetection` objects

Sensor detections, returned as a cell array of `objectDetection` objects.

### **numDets** — Number of detections

nonnegative integer

Number of detections reported, returned as a nonnegative integer.

Data Types: `double`

### **sensorConfigs** — Sensor configurations

array of sensor configuration structures

Sensor configurations, returned as an array of sensor configuration structures. Each structure has these fields.

Field	Description
<code>SensorIndex</code>	Unique sensor index, returned as a positive integer.

<code>IsValidTime</code>	Valid detection time, returned as <code>true</code> or <code>false</code> . <code>IsValidTime</code> is <code>false</code> when detection updates are requested between update intervals specified by the update rate.
<code>IsScanDone</code>	<code>IsScanDone</code> is <code>true</code> when the sensor has completed a scan.
<code>FieldOfView</code>	Field of view of the sensor, returned as a 2-by-1 vector of positive real values, <code>[azfov;elfov]</code> . <code>azfov</code> and <code>elfov</code> represent the field of view in azimuth and elevation, respectively.
<code>RangeLimits</code>	Minimum and maximum range of sensor, in meters, specified as a 1-by-2 nonnegative real-valued vector of the form <code>[rmin, rmax]</code> .
<code>RangeRateLimits</code>	Minimum and maximum range rate of sensor, in meters per second, specified as a 1-by-2 real-valued vector of the form <code>[rrmin, rrmx]</code> .
<code>MeasurementParameters</code>	Sensor measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current sensor frame.

**See Also**

`Platform` | `platform` | `objectDetection` | `radarDataGenerator` | `pose` | `emit`

**Introduced in R2021a**

# emit

**Package:** radar.scenario

Collect emissions from all emitters mounted on platform

## Syntax

```
emissions = emit(plat,time)
[emissions,emitterConfigs] = emit(plat)
```

## Description

`emissions = emit(plat,time)` reports signals emitted from all emitters mounted on the platform, `plat`, at the specified emission time, `time`.

`[emissions,emitterConfigs] = emit(plat)` also returns the configurations of all emitters at the emission time.

## Input Arguments

### **plat** – Scenario platform

Platform object

Scenario platform, specified as a Platform object. To create platforms, use the platform function.

### **time** – Simulation time

0 (default) | positive scalar

Simulation time, specified as a positive scalar.

Example: 1.5

Data Types: single | double

## Output Arguments

### **emissions** – Emissions of all emitters

cell array of emission objects

Emissions of all emitters mounted on the platform, returned as a cell array of emission objects such as radarEmission objects.

### **emitterConfigs** – Emitter configurations

array of sensor configuration structures

Emitter configurations, returned as an array of emitter configuration structures. Each structure has these fields.

Field	Description
-------	-------------

EmitterIndex	Unique emitter index, returned as a positive integer.
IsValidTime	Valid emission time, returned as 0 or 1. IsValidTime is 0 when emitter updates are requested at times that are between update intervals specified by the UpdateInterval property.
IsScanDone	Whether the emitter has completed a scan, returned as true or false.
FieldOfView	Field of view of the emitter, returned as a two-element vector [azimuth; elevation] in degrees.
MeasurementParameters	Emitter measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.

**See Also**

Platform | platform | pose | detect | radarEmitter

**Introduced in R2021a**

## pose

**Package:** radar.scenario

Update pose for platform

### Syntax

```
p = pose(plat)
p = pose(plat,type)
p = pose( ____, 'CoordinateSystem', coordinateSystem)
```

### Description

`p = pose(plat)` returns the estimated pose, `p`, of the platform `plat`, in scenario coordinates. The platform must already exist in the radar scenario. Add platforms to a scenario using the `platform` function. The pose is estimated by a pose estimator specified in the `PoseEstimator` property of the platform.

`p = pose(plat,type)` specifies the source of the platform pose information, `type`, as 'estimated' or 'true'.

`p = pose( ____, 'CoordinateSystem', coordinateSystem)` specifies the coordinate system of the pose. You can use this syntax only when the `IsEarthCentered` property of the radar scenario is set to true.

### Examples

#### Get Pose of Platform

Create a radar scenario.

```
rs = radarScenario;
```

Add a platform to the scenario.

```
plat = platform(rs);
plat.Trajectory.Position = [1 1 0];
plat.Trajectory.Orientation = quaternion([90 0 0], 'eulerd', 'ZYX', 'frame');
```

Extract the pose of the platform.

```
p = pose(plat)
p = struct with fields:
    Orientation: [1x1 quaternion]
    Position: [1 1 0]
    Velocity: [0 0 0]
    Acceleration: [0 0 0]
    AngularVelocity: [0 0 0]
```

## Input Arguments

### **plat** — Scenario platform

Platform object

Scenario platform, specified as a Platform object. To create platforms, use the platform function.

### **type** — Source of platform pose information

'estimated' (default) | 'true'

Source of the platform pose information, specified as one of these values:

- 'estimated' — Estimate poses using the pose estimator specified in the PoseEstimator property of the radar scenario.
- 'true' — Return the true pose of the platform.

Data Types: char

### **coordinateSystem** — Coordinate system to report pose

'Cartesian' (default) | 'Geodetic'

Coordinate system to report pose, specified as one of these values:

- 'Cartesian' — Report poses using Cartesian coordinates in the Earth-Centered-Earth-Fixed coordinate frame.
- 'Geodetic' — Report poses using geodetic coordinates (latitude, longitude, and altitude). Report orientation, velocity, and acceleration in the local reference frame (North-East-Down by default) corresponding to the current waypoint.

Specify this argument only when the IsEarthCentered property of the radar scenario is set to true.

## Output Arguments

### **p** — Pose of platform

structure

Pose of the platform, returned as a structure. Pose consists of the position, velocity, orientation, and angular velocity of the platform with respect to the radar scenario coordinates. The structure has these fields.

Field	Description
PlatformID	Unique identifier for the platform, specified as a positive integer. This is a required field with no default value.
ClassID	User-defined integer used to classify the type of target, specified as a nonnegative integer. Zero is reserved for unclassified platform types and is the default value.



Field	Description
Position	<p>Position of target in scenario coordinates, specified as a real-valued 1-by-3 row vector.</p> <ul style="list-style-type: none"> <li>If the <code>coordinateSystem</code> argument is specified as 'Cartesian', then <code>Position</code> is a three-element vector of Cartesian position coordinates in meters.</li> <li>If the <code>coordinateSystem</code> argument is specified as 'Geodetic', then <code>Position</code> is a three-element vector of geodetic coordinates: latitude in degrees, longitude in degrees, and altitude in meters.</li> </ul>
Velocity	<p>Velocity of platform in scenario coordinates, specified as a real-valued 1-by-3 row vector. Units are meters per second. The default value is <code>[0 0 0]</code>.</p>
Acceleration	<p>Acceleration of the platform in scenario coordinates, specified as a 1-by-3 row vector in meters per second squared. The default value is <code>[0 0 0]</code>.</p>
Orientation	<p>Orientation of the platform with respect to the local scenario navigation frame, specified as a scalar quaternion or a 3-by-3 rotation matrix. Orientation defines the frame rotation from the local navigation coordinate system to the current platform body coordinate system. Units are dimensionless. The default value is <code>quaternion(1,0,0,0)</code>.</p>
AngularVelocity	<p>Angular velocity of the platform in scenario coordinates, specified as a real-valued 1-by-3 vector. The magnitude of the vector defines the angular speed. The direction defines the axis of clockwise rotation. Units are degrees per second. The default value is <code>[0 0 0]</code>.</p>

## See Also

Platform | platform | insSensor | platformPoses

Introduced in R2021a

## receive

**Package:** radar.scenario

Receive IQ signal from radars mounted on platform

### Syntax

```
sig = receive(plat,t)
[sig,info] = receive(plat,t)
```

### Description

`sig = receive(plat,t)` returns the target echo, `sig`, received at radars mounted on the platform, `plat`, at time `t`.

`[sig,info] = receive(plat,t)` also returns the configuration information, `info`, of each radar when the signal is received.

### Input Arguments

**plat — Scenario platform**

Platform object

Scenario platform, specified as a Platform object. To create platforms, use the platform function.

**t — Detection time**

nonnegative scalar

Detection time, specified as a nonnegative scalar in seconds.

### Output Arguments

**sig — Signal received at radar receiver**

vector | array

Signal received at the radar receiver, returned as one of these values:

- *NS*-by-*NRE*-by-*N* array -- If the radar uses a regular antenna array for receiving, then the dimension of `sig` is *NS*-by-*NRE*-by-*N*, where *NRE* is the number of antenna elements in the receive antenna array of the radar, *NS* is the number of samples in each transmitted pulse or sweep, and *N* is the number of transmitted pulses or sweeps. In this case, *N* is the value of the NumRepetition property.
- *NS*-by-*NRS*-by-*N* array -- If the radar uses a subarray for receiving, then the dimension of `sig` is *NS*-by-*NRS*-by-*N*, where *NRS* is the number of subarrays in the receive antenna array of the radar. When multiple pulses or sweeps are simulated, the function assumes that targets move according to a constant velocity trajectory.

Data Types: double

**info — Simulation metadata**

structure

Simulation metadata, returned as a structure containing the following fields:

- `IsScanDone` -- Whether one period of mechanical scan is done
- `MechanicalAngle` -- Current antenna pointing angle due to mechanical scan
- `Origin` -- Radar location in the platform coordinate system
- `Orientation` -- Radar orientation axes in the platform coordinate system

Data Types: `struct`

**See Also**`radarTransceiver`**Introduced in R2021a**

## targetPoses

**Package:** radar.scenario

Target positions and orientations as seen from platform

### Syntax

```
poses = targetPoses(plat)
poses = targetPoses(plat, format)
```

### Description

`poses = targetPoses(plat)` returns the `poses` of all targets in a scenario with respect to the observing platform, `plat`. Targets are defined as platforms as seen by `plat`. Pose represents the position, velocity, and orientation of a target with respect to the coordinate system of `plat`. The targets must already exist in the radar scenario. Add targets to a scenario using the `platform` function.

`poses = targetPoses(plat, format)` also specifies the format of the returned platform orientation as 'quaternion' or 'rotmat'.

### Input Arguments

#### **plat** — Observing platform

Platform object

Observing platform, specified as a `Platform` object. To create platforms, use the `platform` function.

#### **format** — Pose orientation format

'quaternion' (default) | 'rotmat'

Pose orientation format, specified as 'quaternion' or 'rotmat'. When specified as 'quaternion', the `Orientation` field of the platform pose structure is a quaternion. When specified as 'rotmat', the `Orientation` field is a rotation matrix.

Data Types: char | string

### Output Arguments

#### **poses** — Poses of all targets

structure | array of structures

Poses for all targets, returned as a structure or an array of structures. The pose of the observing platform, `plat`, is not included. Pose consists of the position, velocity, orientation, and signature of a target in platform coordinates. Each structure has these fields.

Field	Description
PlatformID	Unique identifier for the platform, specified as a positive integer. This is a required field with no default value.
ClassID	User-defined integer used to classify the type of target, specified as a nonnegative integer. 0 is reserved for unclassified platform types and is the default value.
Position	Position of the target in platform coordinates, specified as a real-valued, 1-by-3 vector. This is a required field with no default value. Units are in meters.
Velocity	Velocity of the target in platform coordinates, specified as a real-valued, 1-by-3 vector. Units are in meters per second. The default is [0 0 0].
Acceleration	Acceleration of the target in platform coordinates specified as a 1-by-3 row vector. Units are in meters per second-squared. The default is [0 0 0].
Orientation	Orientation of the target with respect to platform coordinates, specified as a scalar quaternion or a 3-by-3 rotation matrix. Orientation defines the frame rotation from the platform coordinate system to the current target body coordinate system. Units are dimensionless. The default is quaternion(1,0,0,0).
AngularVelocity	Angular velocity of the target in platform coordinates, specified as a real-valued, 1-by-3 vector. The magnitude of the vector defines the angular speed. The direction defines the axis of clockwise rotation. Units are in degrees per second. The default is [0 0 0].

## See Also

Platform | platform | pose | detect | platformPoses

**Introduced in R2021a**

# radarTransceiver

Monostatic radar transceiver

## Description

The `radarTransceiver` System object creates a monostatic radar object that generates samples of the received target echo at the radar.

To generate samples of the received target echo:

- 1 Create the `radarTransceiver` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
radarTrans = radarTransceiver  
radarTrans = radarTransceiver(Name,Value)
```

### Description

`radarTrans = radarTransceiver` creates a monostatic radar object. This object generates samples of the received target echo at the radar.

`radarTrans = radarTransceiver(Name,Value)` creates a monostatic radar transceiver object with each specified property set to the specified value. Enclose each property name in single quotes.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### Waveform — Radar waveform

`phased.RectangularWaveform` (default) | `phased.LinearFMWaveform` | ...

Radar waveform used in the radar system, specified as one of the following objects:

- `phased.RectangularWaveform`
- `phased.LinearFMWaveform`

- `phased.PhaseCodedWaveform`
- `phased.SteppedFMWaveform`
- `phased.FMCWaveform`
- `phased.MFSKWaveform`

#### **Transmitter — Radar transmitter**

`phased.Transmitter` (default)

Radar system's transmitter, specified as a `phased.Transmitter` object.

#### **TransmitAntenna — Radar transmit antenna**

`phased.Radiator` (default) | `phased.WidebandRadiator`

Radar transmit antenna, specified as either a `phased.Radiator` object or `phased.WidebandRadiator` object.

#### **ReceiveAntenna — Radar receive antenna**

`phased.Collector` (default) | `phased.WidebandCollector`

Radar receive antenna, specified as either a `phased.Collector` object or `phased.WidebandCollector`.

#### **Receiver — Radar receiver**

`phased.ReceiverPreamp` (default)

Radar receiver, specified as a `phased.ReceiverPreamp` object.

#### **MechanicalScanMode — Radar mechanical scan mode**

'None' (default) | 'Circular' | 'Sector'

Radar mechanical scan mode, specified as one of the following:

- 'Circular' -- The radar scans counter-clockwise in the azimuth plane. The azimuth plane is defined in the xy plane.
- 'Sector' -- The radar scans back and forth within a sector in the azimuth plane, first in counter-clockwise direction, then in clockwise direction, and so on.
- 'None'

#### **InitialMechanicalScanAngle — Initial mechanical scan angle**

$\theta$  (default) | scalar

Initial mechanical scan angle measured in degrees, and specified as scalar.

#### **Dependencies**

This property applies only when you set the `MechanicalScanMode` property to 'Circular' or 'Sector'.

Data Types: `double`

#### **MechanicalScanLimits — Mechanical azimuth coverage for sector scanning**

`[-60 60]` | two-element row vector

Mechanical azimuth coverage for sector scanning measured in degrees, and specified as a two-element row vector.

**Dependencies**

This property applies only when you set the `MechanicalScanMode` property to 'Sector'.

Data Types: double

**MechanicalScanRate — Mechanical azimuth scanning rate**

10 (default) | positive scalar

Azimuth scanning rate for the mechanical scan measured in degrees per second, and specified as a positive scalar.

**Dependencies**

This property applies only when you set the `MechanicalScanMode` property to 'Circular' or 'Sector'.

Data Types: double

**ElectronicScanMode — Radar electronic scan mode**

'None' (default) | 'Sector' | 'Custom'

Radar electronic scan mode, specified as one of the following:

- 'Sector' -- The radar scans back and forth within a sector in the azimuth plane, first in counter-clockwise direction, then in clockwise direction, and so on.
- 'Custom'
- 'None'

**ElectronicScanLimits — Electronic azimuth coverage for section scanning**

[-60 60; 0 0] (default) | 2-by-2 matrix

Coverage measured in degrees for electronic sector scanning, specified as a 2-by-2 matrix. The first row specifies the scan coverage in the azimuth direction, and the second row specifies the scan coverage in the elevation direction.

**Dependencies**

To enable this property, set the `ElectronicScanMode` property to 'Sector'.

Data Types: double

**ElectronicScanRate — Electronic scanning rate**

[10;0] (default) | two-element column vector

Scanning rate measured in degrees per second for the electronic scan, specified as a two-element column vector. The first row specifies the scan rate in the azimuth direction, and the second row specifies the scan rate in the elevation direction.

**Dependencies**

To enable this property, set the `ElectronicScanMode` property to 'Sector'.

Data Types: double

**MountingLocation — Radar location on mounting platform (m)**

[0 0 0] (default) | 1-by-3 vector



Offset of the radar's origin from the origin of its mounting platform, specified as a 1-by-3 vector in the form  $[x, y, z]$  and measured in meters.

Data Types: `double`

### **MountingAngles — Radar mounting angles (deg)**

`[0 0 0]` (default) | 1-by-3 vector

Angles at which the radar is mounted relative to the platform's orientation, specified as a 1-by-3 vector in Euler angles around  $[z, y, x]$  axes. These angles are also referred to as [yaw, pitch, roll] angles.

Assume the platform's orientation is defined by the axes  $X_p$ ,  $Y_p$ , and  $Z_p$ . The roll angle specifies the counterclockwise rotation around  $X_p$ , the pitch angle specifies the counterclockwise rotation around  $Y_p$ , and the yaw angle specifies the counterclockwise rotation around  $Z_p$ . To obtain the radar's orientation axes  $X_r$ ,  $Y_r$ , and  $Z_r$  from the platform's orientation axes, perform the intrinsic rotation of the platform's orientation axes  $[X_p, Y_p, Z_p]$  in the order of roll, pitch, and yaw.

Data Types: `double`

### **NumRepetitionsSource — Source of number of pulses or sweeps in the signal**

'Property' (default) | 'Input port'

Source of number of pulses or sweeps in the signal, specified as one of the following:

- 'Property' -- The number of pulses or sweeps in the signal is specified by the `NumRepetitions` property.
- 'Input port' -- The number of pulses or sweeps in the signal is specified through an input.

### **NumRepetitions — Number of pulses or sweeps in signal**

1 (default) | positive integer

Number of pulses or sweeps in the signal, specified as a positive integer.

#### **Dependencies**

To enable this property, set the `NumRepetitionsSource` property to 'Property'.

Data Types: `double`

## **Usage**

### **Syntax**

```

y = radarTrans(tgt,t)
y = radarTrans(proppaths,t)
y = radarTrans( ____,N)
y = radarTrans( ____,PRFIDX)
y = radarTrans( ____,wt)
y = radarTrans( ____,steert)
y = radarTrans( ____,wst)
y = radarTrans( ____,wr)
y = radarTrans( ____,steerr)
y = radarTrans( ____,wsr)

```

```
[y,info] = radarTrans( ___ )
```

**Description**

`y = radarTrans(tgt,t)` returns the target echo received at the radar `y`, at time `t` seconds due to targets in `tgt`.

To use this syntax, set the `NumRepetitionSource` to 'Property'.

`y = radarTrans(proppaths,t)` returns the target echo received at the radar `y` at time `t` (in seconds) due to the propagation paths specified in `proppaths`.

This syntax applies when you set the `NumRepetitionSource` to 'Property'.

`y = radarTrans( ___,N)` specifies the number of pulses/sweeps `N` in the signal as a positive integer.

This syntax applies when you set the `NumRepetitionSource` to 'Input port'.

`y = radarTrans( ___,PRFIDX)` specifies the PRF index of the radar waveform as a positive integer.

This syntax applies when you set the `PRFSelectionInputPort` property to `true` in the radar's `Waveform` property.

`y = radarTrans( ___,wt)` specifies the transmit weights of the radar system as a column vector.

This syntax applies when you set the `ElectronicScanMode` property to 'Custom' and the `WeightsInputPort` property to `true` in the radar's `TransmitAntenna` property.

`y = radarTrans( ___,steert)` specifies the transmit steering angle (in degrees) as a 2-by-1 vector in the form [azimuth; elevation].

This syntax applies when you set the `ElectronicScanMode` property to 'Custom'. Use a subarray in the transmit antenna and set its `SubarraySteering` property to 'Phase' or 'Time'.

`y = radarTrans( ___,wst)` specifies the transmit weights applied to each element as either a matrix or a cell array.

This syntax applies when you set the `ElectronicScanMode` property to 'Custom'. Use a subarray in the transmit antenna and set its `SubarraySteering` property to 'Custom'.

`y = radarTrans( ___,wr)` specifies the receive weights of the radar system as a column vector.

This syntax applies when you set the `ElectronicScanMode` property to 'Custom' and the `WeightsInputPort` property to `true` in the radar's `ReceiveAntenna` property.

`y = radarTrans( ___,steerr)` specifies the receive steering angle (in degrees) as a 2-by-1 vector in the form [azimuth; elevation].

This syntax applies when you set the `ElectronicScanMode` property to 'Custom', use a subarray in the receive antenna, and set its `SubarraySteering` property to 'Phase' or 'Time'.

`y = radarTrans( ___,wsr)` specifies the receive weights applied to each element as either a matrix or a cell array.

This syntax applies when you set the `ElectronicScanMode` to 'Custom', use a subarray in the receive antenna, and set its `SubarraySteering` property to 'Custom'.

`[y,info] = radarTrans( ___ )` also returns additional simulation metadata in the structure `info`.

You can combine optional input arguments when you set the properties to enable them. Optional inputs must be listed in the same order as the enabled properties.

Example: `[y,info] = radarTrans(TGT,T,N,PRFIDX,WT,STEERT,WR,STEERR); [y,info] = radarTrans(TGT,T,N,PRFIDX,WT,WST,WR,WSR);`

## Input Arguments

### **tgt** — Radar target

array of structures

Radar target that reflects the signal, specified as an array of structures. Each structure describes a point target and contains the following fields:

- **Position** -- Specify the position of the target as a 1-by-3 vector (in meters) in the form of `[x y z]`. The position is specified in the radar mounting platform's coordinate system.

This is a required field and there is no default value.

- **Velocity** -- Specify the velocity of the target as a 1-by-3 vector (in meters) in the form of `[x y z]`. The velocity is specified in the radar mounting platform's coordinate system. The default value is `[0 0 0]`.
- **Orientation** -- Specify the target orientation as a scalar quaternion or a 3-by-3 real-valued orthonormal frame rotation matrix, which rotates the axes of the radar mounting platform into alignment with the axes of the target's frame. The default value is `quaternion(1,0,0,0)`.
- **Signatures** -- Specify the target radar cross section (RCS) signature as a `struct` or an `rcsSignature` object.

If `Signatures` is a `struct`, it must have the following fields:

- **Azimuth** -- Specify the azimuth angles (in degrees) at which the RCS pattern is sampled as a length-*Q* vector. The default is `[-180 180]`.
- **Elevation** -- Specify the elevation angles (in degrees) at which the RCS pattern is sampled as a length-*P* vector. The default is `[-90; 90]`.
- **Frequency** -- Specify the frequencies (in Hz) at which the RCS pattern is sampled as a length-*K* vector. The default is `[0 1e20]`.
- **Pattern** -- Specify the target's RCS pattern (in dBm) as either a *P*-by-*Q* matrix or a *P*-by-*Q*-by-*K* array. If defined as a *P*-by-*Q*-by-*K* array, each entry in the array specifies the RCS at the corresponding frequency and the corresponding (azimuth, elevation) direction. If defined as a *P*-by-*Q* matrix, then the pattern applies to all frequencies. The default is `[0 0;0 0]`.

Example: `tgt1 = struct('Position',[0 5e3 0],'Velocity',[0 0 0]);tgt2 = struct('Position',[10e3 0 0],'Velocity',[0 0 0]);tgt = [tgt1 tgt2];`

Data Types: `struct`

### **proppaths** — Propagation path between transmitter and receiver

array of structures

Propagation path between transmitter and receiver, specified as an array of structures. Each structure describes a propagation path between the transmitter and the receiver, and contains the following required fields:

- **PathLength** -- Specify the length of a propagation path as a positive scalar (in meters).
- **PathLoss** -- Specify the propagation loss along the path as a scalar (in dB).
- **ReflectionCoefficient** -- Specify the cumulative reflection coefficients for all reflections along the path as a scalar. This include the effects like reflections from a scatterer or a target.
- **AngleOfDeparture** -- Specify the path's angle of departure (in degrees) as a two-column vector in the form [azimuth; elevation] angles. The angle is measured in the transmit antenna's coordinate system.
- **AngleOfArrival** -- Specify the path's angle of arrival (in degrees) as a two-column vector in the form [azimuth; elevation] angles. The angle is measured in the receive antenna's coordinate system.
- **DopplerShift** -- Specify the cumulative Doppler shift along path as a scalar (in Hz).

Data Types: `struct`

**t – Current time in seconds**

nonnegative scalar value

Current time at which the radar receives the target echo, specified as a nonnegative scalar in seconds.

Data Types: `double`

**N – Number of pulses/sweeps**

positive integer

Number of pulses/sweeps in the signal, specified as a positive integer.

You can specify this input only when the `NumRepetitionSource` property is set to `'Input port'`.

Data Types: `double`

**PRFIDX – PRF index of radar waveform**

positive integer

PRF index of the radar waveform, specified as a positive integer.

You can specify this input only when you set the `PRFSelectionInputPort` property to `true` in the radar's `Waveform` property.

Data Types: `double`

**wt – Transmit weights of radar system**

column vector

Transmit weights of the radar system, specified as a column vector.

If a regular antenna array is used to transmit, `wt` is of length `NTE` where `NTE` is the number of antenna elements in the radar's transmit antenna array.

If a subarray is used to transmit, `wt` is of length `NTS` where `NTS` is the number of subarrays in the radar's transmit antenna array.

You can specify this input only when you set the `ElectronicScanMode` property to 'Custom' and the `WeightsInputPort` property to `true` in the radar's `TransmitAntenna` property.

Data Types: `double`

### **steert** — Transmit steering angle

2-by-1 vector

Transmit steering angle (in degrees), specified as a 2-by-1 vector in the form of [azimuth; elevation].

You can specify this input only when you set the `ElectronicScanMode` property to 'Custom'. Use a subarray in the transmit antenna, and set its `SubarraySteering` property to 'Phase' or 'Time'.

Data Types: `double`

### **wst** — Transmit weights applied to each element

matrix | cell array

Transmit weights applied to each element, specified as either a matrix or a cell array.

If the transmit antenna uses a:

- `phased.ReplicatedSubarray`, `wst` must be an *NTE*-by-*NTS* matrix where *NTE* is the number of elements in each individual subarray and *NTS* is the number of subarrays. Each column in `wst` specifies the weights for the elements in the corresponding subarray.
- `phased.PartitionedArray` and its individual subarrays have the same number of elements, `wst` must be an *NTE*-by-*NTS* matrix where *NTE* is the number of elements in each individual subarray and *NTS* is the number of subarrays. Each column in `wst` specifies the weights for the elements in the corresponding subarray.
- `phased.PartitionedArray` and its subarrays can have different number of elements, `wst` can be one of the following:
  - *NTE*-by-*NTS* matrix, where *NTE* indicates the number of elements in the largest subarray and *NTS* is the number of subarrays.

If `wst` is a matrix, the first *KT* entries in each column, where *KT* is the number of elements in the corresponding subarray, specify the weights for the elements in the corresponding subarray.

- 1-by-*NTS* cell array, where *NTS* is the number of subarrays and each cell contains a column vector whose length is the same as the number of elements of the corresponding subarray.

You can specify this input only when you set the `ElectronicScanMode` property to 'Custom'. Use a subarray in the transmit antenna, and set its `SubarraySteering` property to 'Custom'.

Data Types: `double`

### **wr** — Receive weights of radar system

column vector

Receive weights of the radar system, specified as a column vector. If a regular antenna array is used to receive, `wr` is of length *NRE*, where *NRE* is the number of antenna elements in the radar's receive antenna array. If a subarray is used to receive, `wr` is of length *NRS* where *NRS* is the number of subarrays in the radar's receive antenna array.

You can specify this input only when you set the `ElectronicScanMode` property to 'Custom' and the `WeightsInputPort` property to `true` in the radar's `ReceiveAntenna` property.

Data Types: double

### **steerr** — Receive steering angle in degrees

2-by-1 vector

Receive steering angle in degrees, specified as a 2-by-1 vector in the form of [azimuth; elevation].

You can specify this input only when you set the `ElectronicScanMode` property to 'Custom', use a subarray in the receive antenna, and set its `SubarraySteering` property to 'Phase' or 'Time'.

Data Types: double

### **wsr** — Receive weights applied to each element

matrix | cell array

Receive weights applied to each element, specified as either a matrix or a cell array.

If the receive antenna uses a:

- `phased.ReplicatedSubarray` object, `wsr` must be an  $NRE$ -by- $NRS$  matrix where  $NRE$  is the number of elements in each individual subarray and  $NRS$  is the number of subarrays. Each column in `wsr` specifies the weights for the elements in the corresponding subarray.
- `phased.PartitionedArray` object, and its individual subarrays have same number of elements, `wsr` must be an  $NRE$ -by- $NRS$  matrix where  $NRE$  is the number of elements in each individual subarray and  $NRS$  is the number of subarrays. Each column in `wsr` specifies the weights for the elements in the corresponding subarray.
- `phased.PartitionedArray` object, and its subarrays can have different number of elements, `wsr` can be one of the following:
  - $NRE$ -by- $NRS$  matrix, where  $NRE$  indicates the number of elements in the largest subarray and  $NRS$  is the number of subarrays.

If `wsr` is a matrix, the first  $KR$  entries in each column, where  $KR$  is the number of elements in the corresponding subarray, specify the weights for the elements in the corresponding subarray.

- 1-by- $NRS$  cell array, where  $NRS$  is the number of subarrays and each cell contains a column vector whose length is the same as the number of elements of the corresponding subarray.

You can specify this input only when you set the `ElectronicScanMode` to 'Custom', use a subarray in the receive antenna, and set its `SubarraySteering` property to 'Custom'.

Data Types: double

## **Output Arguments**

### **y** — Signal received at radar receiver

vector | array

Signal received at the radar receiver, returned as a one of the following:

- $NS$ -by- $NRE$ -by- $N$  array -- If the radar uses a regular antenna array for receiving, the dimension of `y` is  $NS$ -by- $NRE$ -by- $N$ , where  $NRE$  is the number of antenna elements in the radar's receive antenna array,  $NS$  is the number of samples in each transmitted pulse/sweep, and  $N$  is the number of transmitted pulses/sweeps.

In this syntax,  $N$  is specified by the value of the `NumRepetition` property.

- *NS-by-NRS-by-N* array -- If the radar uses a subarray for receiving, the dimension of  $y$  is *NS-by-NRS-by-N*, where *NRS* is the number of subarrays in the radar's receive antenna array. When multiple pulses/sweeps are simulated, the targets are assumed to move according to a constant velocity trajectory.

Data Types: `double`

### info — Simulation metadata

structure

Simulation metadata, returned as a structure containing the following fields:

- `IsScanDone` -- Whether one period of mechanical scan is done.
- `MechanicalAngle` -- Current antenna pointing angle due to mechanical scan.
- `Origin` -- Radar location in the platform coordinate system.
- `Orientation` -- Radar orientation axes in the platform coordinate system.

Data Types: `struct`

## Object Functions

To use an object function, specify the `System` object as the first input argument. For example, to release system resources of a `System` object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

`step` Run `System` object algorithm

`release` Release resources and allow changes to `System` object property values and input characteristics

`reset` Reset internal states of `System` object

## Examples

### Model Target Echo Received by Monostatic Radar

Model the target echo received by a monostatic radar using the `radarTransceiver` object.

Create the radar targets as an array of two structures with a specified position and velocity.

```
tgt1 = struct( ...
    'Position', [0 5e3 0], ...
    'Velocity', [0 0 0]);
tgt2 = struct( ...
    'Position', [10e3 0 0], ...
    'Velocity', [0 0 0]);
```

Create a surveillance radar 15 meters above the ground. Specify `rpm` to determine the scan rate (in deg/s). For the specified scanrate and beamwidth, determine the update rate.

```

rpm = 12.5;
scanrate = rpm*360/60;      % deg/s
beamw = 1;                 % beamwidth
updaterate = scanrate/beamw; % update at each beam
radarht = 15.0;           % radar height

```

Create a `phased.CustomAntennaElement` object that acts as a transmit antenna element and a receive antenna element in the `radarTransceiver` object.

```

az = -180:0.5:180;
el = -90:0.5:90;
pat = zeros(numel(el),numel(az));
pat(-0.5 <= el & el <= 0.5,-0.5 <= az & az <= 0.5) = 1;
ant = phased.CustomAntennaElement('AzimuthAngles',az,...
    'ElevationAngles',el,'MagnitudePattern',mag2db(abs(pat)),...
    'PhasePattern',zeros(size(pat)));

```

Create a `radarTransceiver` object. Specify a rectangular waveform for the radar using the `phased.RectangularWaveform` object. Specify the transmit antenna and the receive antenna. The mechanical scan mode is set to 'Circular' with a defined scan rate.

```

wav = phased.RectangularWaveform('PulseWidth',1e-5);
sensor = radarTransceiver( ...
    'Waveform',wav, ...
    'TransmitAntenna',phased.Radiator('Sensor',ant), ...
    'ReceiveAntenna',phased.Collector('Sensor',ant), ...
    'MechanicalScanMode','Circular', ...
    'MechanicalScanRate',scanrate, ...
    'MountingLocation',[0,0,radarht]);

```

Generate detections from a full scan of the radar.

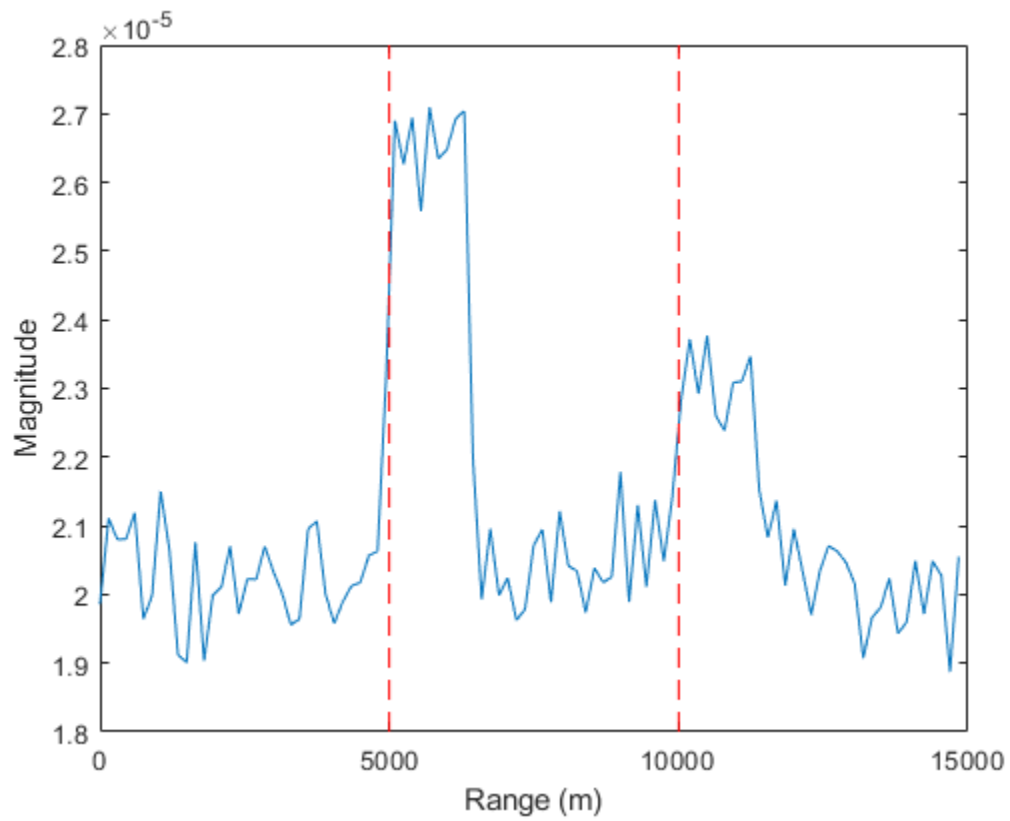
```

simTime = 0;
sigi = 0;
while true
    [sig, info] = sensor([tgt1 tgt2], simTime);
    sigi = sigi + abs(sig);

    % Is full scan complete?
    if info.IsScanDone
        break % yes
    end
    simTime = simTime + 1/updaterate;
end
r = (0:size(sigi,1)-1)/sensor.Waveform.SampleRate* ...
    sensor.TransmitAntenna.PropagationSpeed/2;
plot(r,sigi)
hold on
plot([5e3 5e3],ylim,'r--',[10e3 10e3],ylim,'r--')
xlabel('Range (m)')
ylabel('Magnitude')

```





## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Objects

Introduced in R2021a

## radarScenario

Create radar scenario

### Description

`radarScenario` creates a radar scenario object. A radar scenario simulates a 3-D environment containing multiple platforms. Platforms represent objects that you want to simulate, such as aircraft, ground vehicles, or ships. Some platforms carry sensors, such as radar, sonar, or infrared. Other platforms act as the source of signals or reflector of signals.

Populate a radar scenario by calling the `platform` function for each platform you want to add. You can model platforms as points or cuboids by specifying the `'Dimension'` property when calling the `platform` function. Platforms have signatures with properties that are specific to the type of sensor, such as radar cross-section for radar sensors. You can create trajectories for any platform using the `kinematicTrajectory`, `waypointTrajectory`, or `geoTrajectory` System object.

After you add all desired platforms, you can simulate the scenario in incremental time steps by using the `advance` function in a loop. You can run the simulation all at once using the `record` function.

### Creation

#### Syntax

```
scene = radarScenario
scene = radarScenario('IsEarthCentered',true)
scene = radarScenario(Name,Value)
```

#### Description

`scene = radarScenario` creates an empty radar scenario `scene` with default property values. You can specify platform trajectories in the scenario as Cartesian states using the `kinematicTrajectory` or `waypointTrajectory` System object.

`scene = radarScenario('IsEarthCentered',true)` creates an empty Earth-centered radar scenario and sets the `IsEarthCentered` on page 4-0 property as `true`. You can specify platform trajectories in the scenario as geodetic states using the `geoTrajectory` System object.

`scene = radarScenario(Name,Value)` configures the properties on page 4-34 of a `radarScenario` object using one or more name-value arguments. `Name` is a property name and `Value` is the corresponding value. You can specify several name-value arguments in any order. Any unspecified properties take default values.

### Properties

#### **IsEarthCentered — Enable Earth-centered reference frame and trajectories**

`false` or `0` (default) | `true` or `1`

Enable Earth-centered reference frame and trajectories, specified as a logical `0` (`false`) or `1` (`true`).

- If specified as `0` (`false`), then you must define the trajectories of platforms as Cartesian states using the `kinematicTrajectory` or `waypointTrajectory` System object.
- If specified as `1` (`true`), then you must define the trajectories of platforms as geodetic states using the `geoTrajectory` System object.

You can specify the `IsEarthCentered` property only when creating the radar scenario.

Data Types: `logical`

### **UpdateRate — Frequency of simulation updates**

`10` (default) | `nonnegative scalar`

Frequency of simulation updates, specified as a nonnegative scalar in hertz.

- When specified as a positive scalar, the scenario advances with the time step of  $1/F$ , where  $F$  is the value of the `UpdateRate` property.
- When specified as `0`, the simulation advances to the next scheduled sampling time of any mounted sensors or emitters. For example, if a scenario has two sensors with update rates of 2 Hz and 5 Hz, then the first seven simulation updates are at 0, 0.2, 0.4, 0.5, 0.6, 0.8 and 1.0 seconds, respectively.

Example: `2.0`

Data Types: `double`

### **SimulationTime — Current time of simulation**

`0` (default) | `positive scalar`

This property is read-only.

Current time of the simulation, specified as a positive scalar in seconds. To reset the simulation time to zero and restart the simulation, call the `restart` function.

Data Types: `double`

### **StopTime — Stop time of simulation**

`Inf` (default) | `positive scalar`

Stop time of the simulation, specified as a positive scalar in seconds. The simulation stops when either of these conditions is met:

- The stop time is reached
- Any platform reaches the end of its trajectory and you have specified the platform `Motion` property with waypoints using the `waypointTrajectory` System object

Example: `60.0`

Data Types: `double`

### **SimulationStatus — Simulation status**

`NotStarted` | `InProgress` | `Completed`

This property is read-only.

Simulation status, specified as one of these values.

- `NotStarted` — When the advance function has not been used on the radar scenario.
- `InProgress` — When the advance function has been used on the radar scenario at least once and the scenario has not reached the `Completed` status.
- `Completed` — When the scenario reaches the stop time specified by the `StopTime` property or any `Platform` object in the scenario reaches the end of its trajectory.

You can restart a scenario simulation by using the `restart` object function.

Data Types: enumeration

### Platforms — Platforms in scenario

cell array

This property is read-only.

Platforms in the scenario, returned as a cell array of `Platform` objects. The number of elements in the cell array is equal to the number of platforms in the scenario. To add a platform to the scenario, use the `platform` function.

### InitialAdvance — Initial advance when calling advance function

Zero (default) | `UpdateInterval`

Initial advance when calling the advance function, specified as one of these values.

- `Zero` — The scenario simulation starts at time  $0$  in the first call to the advance function.
- `UpdateInterval` — The scenario simulation starts at time  $1/F$ , where  $F$  is the value of a nonzero `UpdateRate` property. If the `UpdateRate` property is specified as  $0$ , then the scenario simulation ignores the `InitialAdvance` property and starts at time  $0$ .

Data Types: enumeration

### SurfaceManager — Surface manager

`surfaceManager` object

This property contains the `SurfaceManager` object associated with the scenario.

## Object Functions

<code>platform</code>	Add platform to radar scenario
<code>advance</code>	Advance radar scenario simulation by one time step
<code>restart</code>	Restart simulation of radar scenario
<code>record</code>	Record simulation of radar scenario
<code>emit</code>	Collect emissions from all emitters in radar scenario
<code>propagate</code>	Propagate emissions in radar scenario
<code>detect</code>	Collect detections from all sensors in radar scenario
<code>receive</code>	Receive IQ signal from radars in the scenario
<code>clutterGenerator</code>	Add clutter generator for radar
<code>platformProfiles</code>	Profiles of radar scenario platforms
<code>platformPoses</code>	Position information for each platform in radar scenario
<code>coverageConfig</code>	Sensor and emitter coverage configuration
<code>perturb</code>	Apply perturbations to radar scenario
<code>receive</code>	Receive IQ signal from radars in the scenario
<code>clone</code>	Create copy of radar scenario

## Examples

### Create Radar Scenario with Two Platforms

Create a radar scenario with two platforms that follow different trajectories.

```
sc = radarScenario('UpdateRate',100,'StopTime',1.2);
```

Create two platforms.

```
platfm1 = platform(sc);
platfm2 = platform(sc);
```

Platform 1 follows a circular path of radius 10 m for one second. This is accomplished by placing waypoints in a circular shape, ensuring that the first and last waypoint are the same.

```
wpts1 = [0 10 0; 10 0 0; 0 -10 0; -10 0 0; 0 10 0];
time1 = [0; 0.25; .5; .75; 1.0];
platfm1.Trajectory = waypointTrajectory(wpts1,time1);
```

Platform 2 follows a straight path for one second.

```
wpts2 = [-8 -8 0; 10 10 0];
time2 = [0; 1.0];
platfm2.Trajectory = waypointTrajectory(wpts2,time2);
```

Verify the number of platforms in the scenario.

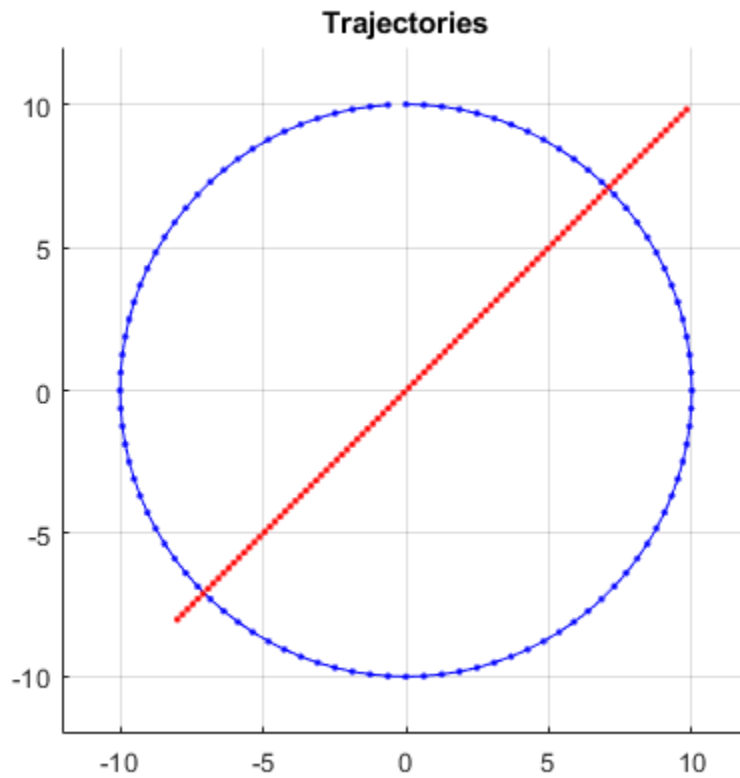
```
disp(sc.Platforms)

    {1x1 radar.scenario.Platform}    {1x1 radar.scenario.Platform}
```

Run the simulation and plot the current position of each platform using an animated line.

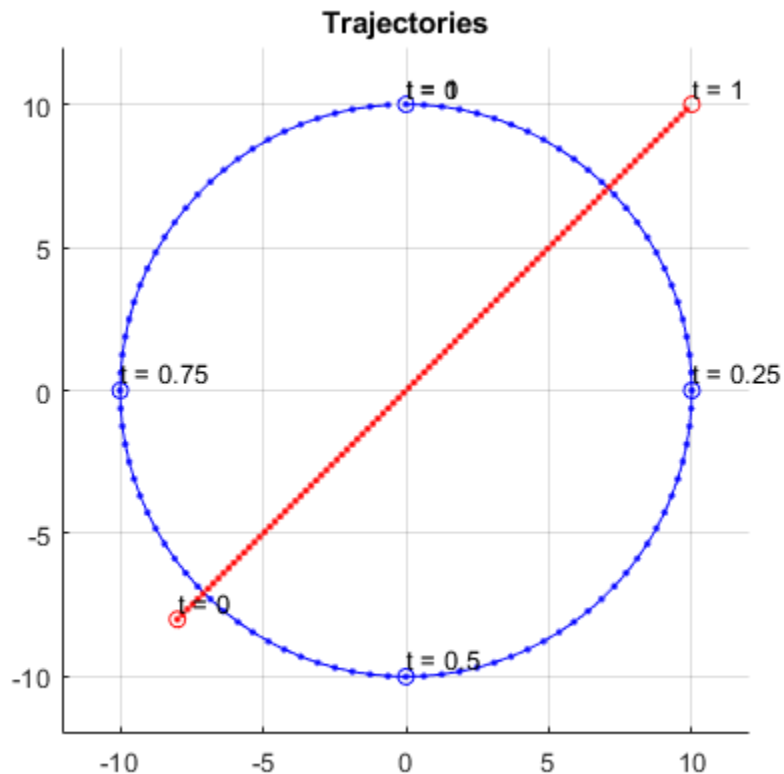
```
figure
grid
axis equal
axis([-12 12 -12 12])
line1 = animatedline('DisplayName','Trajectory 1','Color','b','Marker','.');
line2 = animatedline('DisplayName','Trajectory 2','Color','r','Marker','.');
title('Trajectories')
p1 = pose(platfm1);
p2 = pose(platfm2);
addpoints(line1,p1.Position(1),p1.Position(2));
addpoints(line2,p2.Position(2),p2.Position(2));

while advance(sc)
    p1 = pose(platfm1);
    p2 = pose(platfm2);
    addpoints(line1,p1.Position(1),p1.Position(2));
    addpoints(line2,p2.Position(2),p2.Position(2));
    pause(0.1)
end
```



Plot the waypoints for both platforms.

```
hold on
plot(wpts1(:,1),wpts1(:,2),'ob')
text(wpts1(:,1),wpts1(:,2),"t = " + string(time1),'HorizontalAlignment','left','VerticalAlignment','bottom')
plot(wpts2(:,1),wpts2(:,2),'or')
text(wpts2(:,1),wpts2(:,2),"t = " + string(time2),'HorizontalAlignment','left','VerticalAlignment','bottom')
hold off
```



### Create Earth-Centered Radar Scenario

Create an Earth-centered radar scenario and specify the update rate.

```
scene = radarScenario('IsEarthCentered',true,'UpdateRate',0.01);
```

Add a platform to the scenario that represents an airplane. The trajectory of the airplane changes in longitude and altitude. Specify the trajectory using geodetic coordinates.

```
geoTraj = geoTrajectory([42.300,-71.351,10600;42.300,-124.411,0],[0 21600]);
plane = platform(scene,'Trajectory',geoTraj);
```

Advance the radar scenario and record the geodetic and Cartesian positions of the plane target.

```
positions = [];
while advance(scene)
    poseLLA = pose(plane,'CoordinateSystem','Geodetic');
    poseXYZ = pose(plane,'CoordinateSystem','Cartesian');
    positions = [positions;poseXYZ.Position];%#ok<AGROW> Allow the buffer to grow.
end
```

Convert the distance units from meters to kilometers.

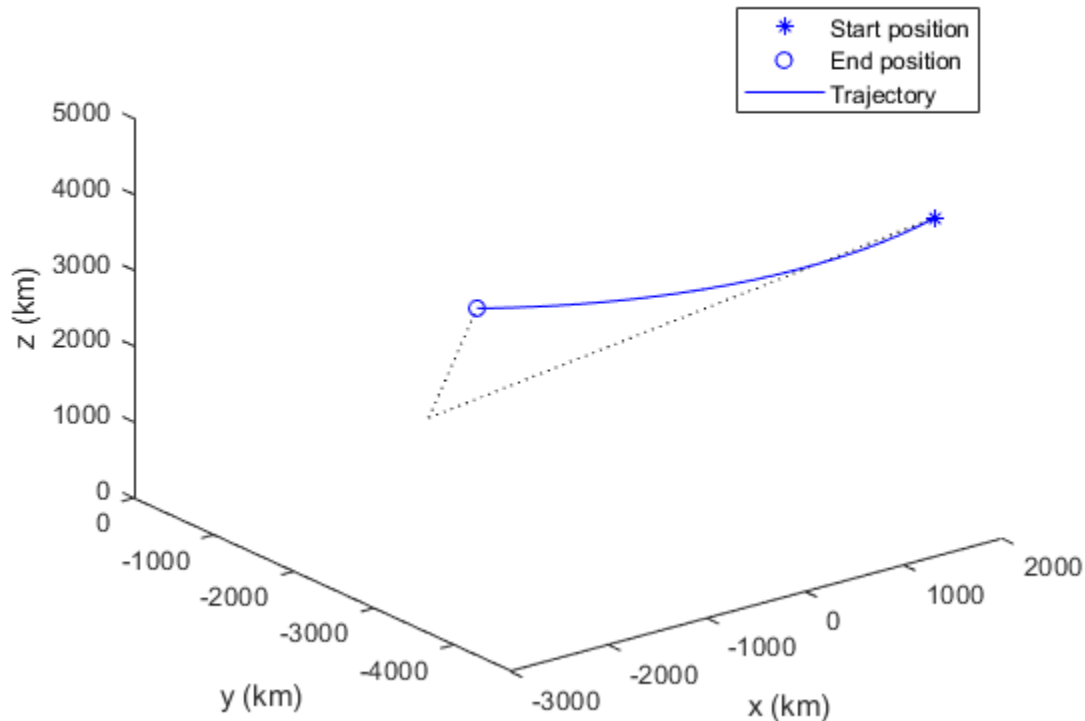
```
km = 1000;
positions = positions/km;
```

Visualize the start position, end position, and trajectory in the ECEF frame.

```
hold on
plot3(positions(1,1),positions(1,2),positions(1,3),'b*')
plot3(positions(end,1),positions(end,2),positions(end,3),'bo')
plot3(positions(:,1),positions(:,2),positions(:,3),'b')
```

Plot the Earth radial lines of the start position and end position.

```
plot3([0 positions(1,1)],[0 positions(1,2)],[0 positions(1,3)],'k:')
plot3([0 positions(end,1)],[0 positions(end,2)],[0 positions(end,3)],'k:')
xlabel('x (km)')
ylabel('y (km)')
zlabel('z (km)')
legend('Start position','End position','Trajectory')
view(3)
```



## See Also

[kinematicTrajectory](#) | [geoTrajectory](#) | [waypointTrajectory](#) | [Platform](#) | [radarScenarioRecording](#) | [SurfaceManager](#)

## Topics

“Radar Scenario Tutorial”

**Introduced in R2021a**



## advance

Advance radar scenario simulation by one time step

### Syntax

```
isRunning = advance(scenario)
```

### Description

`isRunning = advance(scenario)` advances the simulation of the radar scenario `scenario` by one time step, and returns the running status of the scenario. Set up the advance behavior using the `UpdateRate` and `InitialAdvance` properties of the `radarScenario` object.

- When the `UpdateRate` property is a positive scalar  $F$ , the simulation advances in the time step of  $1/F$ . Moreover, if the `InitialAdvance` property is 'Zero', then the simulation starts at time 0. If the `InitialAdvance` property is specified as 'UpdateInterval', then the simulation starts at time  $1/F$ .
- When the `UpdateRate` property is 0, the simulation advances to the next scheduled sampling time of any mounted sensors or emitters. For example, if a scenario has two sensors with update rates of 2 Hz and 5 Hz, then the first seven simulation updates are at 0, 0.2, 0.4, 0.5, 0.6, 0.8 and 1.0 seconds, respectively.

In this case, the initial time is always time 0. Also, you must trigger the running of the sensors or emitters by using at least one of the these options between calls to the `advance` function:

- Directly running the sensors or emitters
- Using the `emit` or `detect` function of the radar scenario to run sensors or emitters in the scenario
- Using the `emit` or `detect` function of the platform with corresponding sensors or emitters

### Examples

#### Advance Radar Scenario

Create a new radar scenario.

```
rs = radarScenario;
```

Create a platform that follows a circular path of radius 10 m for one second. This is accomplished by placing waypoints in a circular shape, ensuring that the first and last waypoint are the same.

```
plat = platform(rs);
wpts = [0 10 0; 10 0 0; 0 -10 0; -10 0 0; 0 10 0];
time = [0; 0.25; .5; .75; 1.0];
plat.Trajectory = waypointTrajectory(wpts,time);
```

Perform the simulation, advancing one time step at a time. Display the simulation time and the position and velocity of the platform at each time step.

```
while advance(rs)
    p = pose(plat);
    disp(strcat("Time = ",num2str(rs.SimulationTime)))
    disp(strcat("  Position = [",num2str(p.Position),"]"))
    disp(strcat("  Velocity = [",num2str(p.Velocity),"]"))
end
```

```
Time = 0
    Position = [0  10  0]
    Velocity = [62.8318 -1.88403e-05 0]
Time = 0.1
    Position = [5.8779 8.0902 0]
    Velocity = [50.832 -36.9316 0]
Time = 0.2
    Position = [9.5106 3.0902 0]
    Velocity = [19.4161 -59.7566 0]
Time = 0.3
    Position = [9.5106 -3.0902 0]
    Velocity = [-19.4161 -59.7567 0]
Time = 0.4
    Position = [5.8779 -8.0902 0]
    Velocity = [-50.832 -36.9316 0]
Time = 0.5
    Position = [0 -10 0]
    Velocity = [-62.8319 1.88181e-05 0]
Time = 0.6
    Position = [-5.8779 -8.0902 0]
    Velocity = [-50.832 36.9316 0]
Time = 0.7
    Position = [-9.5106 -3.0902 0]
    Velocity = [-19.4161 59.7566 0]
Time = 0.8
    Position = [-9.5106 3.0902 0]
    Velocity = [19.4161 59.7566 0]
Time = 0.9
```

```
Position = [-5.8779      8.0902      0]
Velocity = [50.832     36.9316     0]
Time = 1
Position = [-7.10543e-15      10      0]
Velocity = [62.8319 -1.88404e-05      0]
```

## Input Arguments

### **scenario** — Radar scenario

radarScenario object

Radar scenario, specified as a radarScenario object.

## Output Arguments

### **isRunning** — Run-state of simulation

0 | 1

Run-state of the simulation, returned as a logical 0 or 1. If `isRunning` is 1, then the simulation is running. If `isRunning` is 0, then the simulation has stopped. A simulation stops when either of these conditions is met:

- The stop time is reached.
- Any platform reaches the end of its trajectory and you have specified the platform `Motion` property with waypoints using the `waypointTrajectory` System object.

Data Types: `logical`

## See Also

`radarScenario` | `restart` | `record` | `detect` | `waypointTrajectory` | `kinematicTrajectory`

**Introduced in R2021a**

## clone

Create copy of radar scenario

### Syntax

```
newScenario = clone(scenario)
```

### Description

`newScenario = clone(scenario)` creates a copy of the radar scenario, `scenario`.

### Examples

#### Copy Radar Scenario

Create a radar scenario.

```
scene = radarScenario;
```

Add a platform with a specified position to the scene.

```
platform(scene, 'Position', [10 10 0]);
```

Create a copy of the scenario. The copy of the scenario, `newScene`, includes the platform.

```
newScene = clone(scene)
```

```
newScene =
```

```
    radarScenario with properties:
```

```
    IsEarthCentered: 0
    UpdateRate: 10
    SimulationTime: 0
    StopTime: Inf
    SimulationStatus: NotStarted
    Platforms: {[1x1 radar.scenario.Platform]}
    SurfaceManager: [1x1 radar.scenario.SurfaceManager]
```

### Input Arguments

#### **scenario** — Radar scenario

radarScenario object

Radar scenario, specified as a `radarScenario` object.

## Output Arguments

### **newScenario** — Copy of radar scenario

radarScenario object

Copy of radar scenario, returned as a radarScenario object.

## See Also

radarScenario

**Introduced in R2021a**

## detect

Collect detections from all sensors in radar scenario

### Syntax

```
detections = detect(scenario)
detections = detect(scenario,signals)
detections = detect(scenario,signals,emitterConfigs)
[detections,sensorConfigs] = detect(____)
[____,sensorConfigPIDs] = detect(____)
```

### Description

`detections = detect(scenario)` reports the detections from all sensors mounted on every platform in the radar scenario, `scenario`. Use this syntax only when none of the sensors require information on the signals present in the scenario.

`detections = detect(scenario,signals)` reports the detections from all sensors when at least one sensor requires information on the signals present in the scenario.

`detections = detect(scenario,signals,emitterConfigs)` reports the detections from all sensors when at least one sensor also requires information on the emitter configurations in the scenario.

`[detections,sensorConfigs] = detect(____)` also returns the configurations of each sensor at the detection time. This output argument can be used with any of the previous syntaxes.

`[____,sensorConfigPIDs] = detect(____)` also returns all platform IDs corresponding to the sensor configurations, `sensorConfigs`. This output argument can be used with any of the previous syntaxes.

### Examples

#### Obtain Detections from Two Platforms in Radar Scenario

Set the seed of the random number generator for reproducible results.

```
s = rng('default');
```

Create a radar scenario.

```
rs = radarScenario('UpdateRate',1);
```

Create the first platform and mount one emitter and one sensor on it.

```
plat1 = platform(rs);
plat1.Trajectory.Position = [0,0,0];
emitter1 = radarEmitter(1,'UpdateRate',1);
sensor1 = radarSensor(1,'DetectionMode','Monostatic','EmitterIndex',1,'RangeResolution',1);
```

```
plat1.Emitters = emitter1;
plat1.Sensors = sensor1;
```

Create the second platform and mount one emitter and one sensor on it.

```
plat2 = platform(rs);
plat2.Trajectory.Position = [100,0,0];
emitter2 = radarEmitter(2, 'UpdateRate',1);
sensor2 = radarSensor(2, 'DetectionMode', 'Monostatic', 'EmitterIndex',2, 'RangeResolution',1);
plat2.Emitters = emitter2;
plat2.Sensors = sensor2;
```

Advance the radar scenario by one time step.

```
advance(rs);
```

Transmit and propagate the emissions.

```
[emtx,emitterConfs,emitterConfPIDs] = emit(rs);
emprop = propagate(rs,emtx, 'HasOcclusion',true);
```

Collect the signals.

```
[dets,sensorConfs,sensorConfPIDs] = detect(rs,emprop,emitterConfs);
```

Display the detection results. The sensor on platform 1 detects the second platform.

```
detection = dets{1}
detection =
  objectDetection with properties:
      Time: 0
      Measurement: [3x1 double]
      MeasurementNoise: [3x3 double]
      SensorIndex: 1
      ObjectClassID: 0
      MeasurementParameters: [1x1 struct]
      ObjectAttributes: {[1x1 struct]}

detectedPlatform = detection.ObjectAttributes{1}
detectedPlatform = struct with fields:
    TargetIndex: 2
    EmitterIndex: 1
    SNR: 82.0123
```

Return the random number generator to its previous state.

```
rng(s)
```

## Input Arguments

### scenario — Radar scenario

radarScenario object

Radar scenario, specified as a `radarScenario` object.

### **signals — Signal emissions**

cell array of signal emission object

Signal emissions, specified as a cell array of signal emission objects, such as `radarEmission` objects.

### **emitterConfigs — Emitter configurations**

array of emitter configuration structures

Emitter configurations, specified as an array of emitter configuration structures. Each structure contains these fields.

<b>Field</b>	<b>Description</b>
<code>EmitterIndex</code>	Unique emitter index, returned as a positive integer.
<code>IsValidTime</code>	Valid emission time, returned as 0 or 1. <code>IsValidTime</code> is 0 when emitter updates are requested at times that are between update intervals specified by the <code>UpdateInterval</code> property.
<code>IsScanDone</code>	Whether the emitter has completed a scan, returned as <code>true</code> or <code>false</code> .
<code>FieldOfView</code>	Field of view of the emitter, returned as a two-element vector [azimuth; elevation] in degrees.
<code>MeasurementParameters</code>	Emitter measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.

## **Output Arguments**

### **detections — Detections**

cell array of `objectDetection` objects

Detections, returned as a cell array of `objectDetection` objects.

### **sensorConfigs — Sensor configurations**

array of sensor configuration structures

Sensor configurations, returned as an array of sensor configuration structures. Each structure contains these fields.

<b>Field</b>	<b>Description</b>
<code>SensorIndex</code>	Unique sensor index, returned as a positive integer.



<code>IsValidTime</code>	Valid detection time, returned as <code>true</code> or <code>false</code> . <code>IsValidTime</code> is <code>false</code> when detection updates are requested between update intervals specified by the update rate.
<code>IsScanDone</code>	<code>IsScanDone</code> is <code>true</code> when the sensor has completed a scan.
<code>FieldOfView</code>	Field of view of the sensor, returned as a 2-by-1 vector of positive real values, <code>[azfov;elfov]</code> . <code>azfov</code> and <code>elfov</code> represent the field of view in azimuth and elevation, respectively.
<code>RangeLimits</code>	Minimum and maximum range of sensor, in meters, specified as a 1-by-2 nonnegative real-valued vector of the form <code>[rmin, rmax]</code> .
<code>RangeRateLimits</code>	Minimum and maximum range rate of sensor, in meters per second, specified as a 1-by-2 real-valued vector of the form <code>[rrmin, rrmx]</code> .
<code>MeasurementParameters</code>	Sensor measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current sensor frame.

### **sensorConfigIDs — Platform IDs for sensor configurations**

array of positive integers

Platform IDs for sensor configurations in the `sensorConfigs` output argument, returned as an array of positive integers.

### **See Also**

`radarScenario` | `detect` | `emit` | `propagate` | `radarEmission`

**Introduced in R2021a**

## emit

Collect emissions from all emitters in radar scenario

### Syntax

```
emissions = emit(scenario)
[emissions,emitterConfigs] = emit(scenario)
[emissions,emitterConfigs,emitterConfigPIDs] = emit(scenario)
```

### Description

`emissions = emit(scenario)` reports signals emitted from all emitters mounted on platforms in the radar scenario, `scenario`.

`[emissions,emitterConfigs] = emit(scenario)` also returns the configurations of all emitters at the emission time.

`[emissions,emitterConfigs,emitterConfigPIDs] = emit(scenario)` also returns the IDs of platforms on which the emitters are mounted.

### Examples

#### Collect Emissions in Radar Scenario

Create a radar scenario and add two platforms. Set the position of each platform and add an emitter to each platform.

```
rs = radarScenario('UpdateRate',1);
plat1 = platform(rs);
plat1.Trajectory.Position = [0,0,0];
emitter1 = radarEmitter(1,'UpdateRate',1);
plat1.Emitters = emitter1;
plat2 = platform(rs);
plat2.Trajectory.Position = [100,0,0];
emitter2 = radarEmitter(2,'UpdateRate',1);
plat2.Emitters = emitter2;
```

Advance the radar scenario by one time step. Collect the emissions of all emitters in the scenario.

```
advance(rs);
[emissions,configs,sensorConfigPIDs] = emit(rs);
```

Confirm that there are two emissions, one from each emitter.

```
disp("There are " + numel(emissions) + " emissions.");
```

```
There are 2 emissions.
```

Display the properties of both emitters after the first time step.

```
disp("The first emission is:"); ...
disp(emissions{1});
```

The first emission is:

radarEmission with properties:

```
PlatformID: 1
EmitterIndex: 1
OriginPosition: [0 0 0]
OriginVelocity: [0 0 0]
Orientation: [1x1 quaternion]
FieldOfView: [1 5]
CenterFrequency: 300000000
Bandwidth: 3000000
WaveformType: 0
ProcessingGain: 0
PropagationRange: 0
PropagationRangeRate: 0
EIRP: 100
RCS: 0
```

```
disp("The second emission is:"); ...
disp(emissions{2});
```

The second emission is:

radarEmission with properties:

```
PlatformID: 2
EmitterIndex: 2
OriginPosition: [100 0 0]
OriginVelocity: [0 0 0]
Orientation: [1x1 quaternion]
FieldOfView: [1 5]
CenterFrequency: 300000000
Bandwidth: 3000000
WaveformType: 0
ProcessingGain: 0
PropagationRange: 0
PropagationRangeRate: 0
EIRP: 100
RCS: 0
```

Display the configuration of both emitters after the first time step.

```
disp("The emitter configuration associated with the first emission is:"); ...
disp(configs(1));
```

The emitter configuration associated with the first emission is:

```
EmitterIndex: 1
IsValidTime: 1
IsScanDone: 0
FieldOfView: [1 5]
RangeLimits: [0 Inf]
RangeRateLimits: [0 Inf]
MeasurementParameters: [1x1 struct]
```

```
disp("The emitter configuration associated with the second emission is:"); ...
disp(configs(2));
```

The emitter configuration associated with the second emission is:

```
EmitterIndex: 2
IsValidTime: 1
IsScanDone: 0
FieldOfView: [1 5]
RangeLimits: [0 Inf]
RangeRateLimits: [0 Inf]
MeasurementParameters: [1x1 struct]
```

Display the platform IDs for the emitter configurations.

```
disp("The emitter configurations are connected with platform IDs: "); ...
disp(sensorConfigPIDs');
```

The emitter configurations are connected with platform IDs:

```
1    2
```

## Input Arguments

### scenario — Radar scenario

radarScenario object

Radar scenario, specified as a radarScenario object.

## Output Arguments

### emissions — Emissions of all emitters

cell array of emission objects

Emissions of all emitters in the radar scenario, returned as a cell array of emission objects such as radarEmission objects.

### emitterConfigs — Emitter configurations

array of sensor configuration structures

Emitter configurations, returned as an array of emitter configuration structures. Each structure contains these fields.

Field	Description
EmitterIndex	Unique emitter index, returned as a positive integer.
IsValidTime	Valid emission time, returned as 0 or 1. IsValidTime is 0 when emitter updates are requested at times that are between update intervals specified by the UpdateInterval property.

IsScanDone	Whether the emitter has completed a scan, returned as <code>true</code> or <code>false</code> .
FieldOfView	Field of view of the emitter, returned as a two-element vector [azimuth; elevation] in degrees.
MeasurementParameters	Emitter measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.

### **emitterConfigPIDs — Platform IDs for emitter configurations**

array of positive integers

Platform IDs for emitter configurations in the `emitterConfigs` output argument, returned as an array of positive integers.

### **See Also**

`radarScenario` | `emit` | `propagate` | `detect`

**Introduced in R2021a**

## perturb

Apply perturbations to radar scenario

### Syntax

```
offsets = perturb(scenario)
```

### Description

`offsets = perturb(scenario)` perturbs the baseline radar scenario, `scenario`, and returns offset values. Use the `perturbations` function to define the perturbations on objects, such as trajectories, sensors, and platforms, in the scenario.

### Examples

#### Radar Scenario Perturbation

Create a radar scenario and add a platform.

```
scenario = radarScenario;
p = platform(scenario);
```

Add a trajectory to the platform.

```
p.Trajectory = waypointTrajectory('Waypoints',...
    [30 -40 -3; 30 -20 -3; 20 -10 -3; 0 -10 -3; -10 -10 -3]*1e3, ...
    'TimeOfArrival', [0; 100; 150; 350; 450], ...
    'Course', [90;90;180;180;180]);
```

Plot the trajectory.

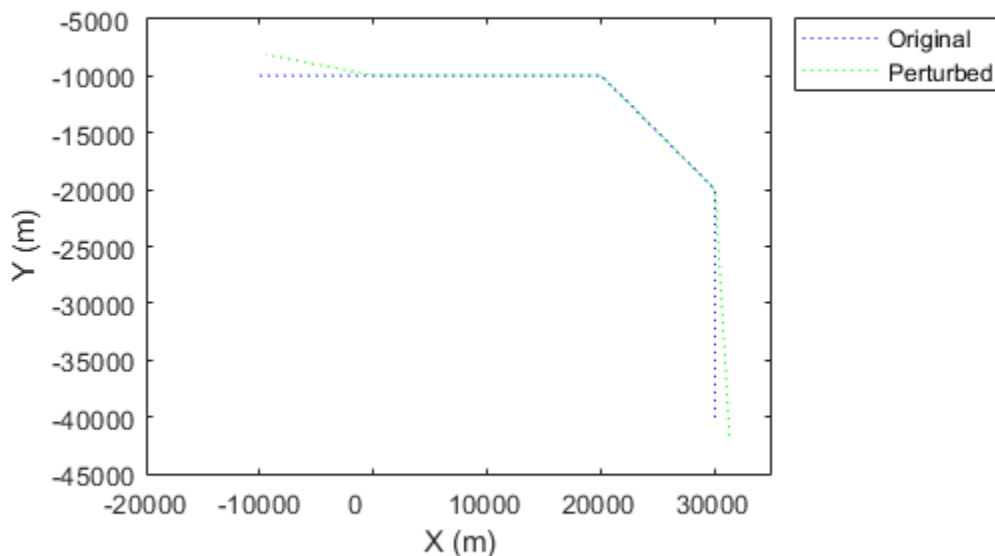
```
tp = theaterPlot("XLimits",[-20 35]*1e3,"YLimits",[-45 -5]*1e3);
trajPlotter1 = trajectoryPlotter(tp,'DisplayName','Original','Color','b');
plotTrajectory(trajPlotter1,{p.Trajectory.Waypoints});
```

Define perturbations for the waypoints. The following defines perturbations on the first and last waypoints as uniform distributions.

```
perturbations(p.Trajectory, "Waypoints", "Uniform",...
    [-2000 -2000 0; 0 0 0; 0 0 0; 0 0 0; -2000 -2000 0],...
    [+2000 +2000 0; 0 0 0; 0 0 0; 0 0 0; +2000 +2000 0]);
```

Perturb the scenario and observe the changed waypoints of the platform.

```
perturb(scenario);
trajPlotter2 = trajectoryPlotter(tp,'DisplayName','Perturbed','Color','g');
plotTrajectory(trajPlotter2,{p.Trajectory.Waypoints})
```



## Input Arguments

### scenario — Radar scenario

radarScenario object

Radar scenario, specified as a radarScenario object.

## Output Arguments

### offsets — Property offsets

array of structures

Property offsets, returned as an array of structures. Each structure contains these fields.

Field Name	Description
PlatformID	ID of the platform
PerturbedObject	Perturbed object mounted on the platform
Property	Name of the perturbed property
Offset	Offset values applied in the perturbation
PerturbedValue	Property values after the perturbation

**See Also**  
perturbations

**Introduced in R2021a**



# platform

Add platform to radar scenario

## Syntax

```
plat = platform(scenario)
plat = platform(scenario,Name,Value)
```

## Description

`plat = platform(scenario)` creates a new Platform object, `plat`, and adds the platform to the radar scenario, `scenario`.

`plat = platform(scenario,Name,Value)` creates a new Platform object with additional properties specified by one or more name-value arguments.

## Examples

### Create Platform with Circular Trajectory

Create a radar scenario.

```
rs = radarScenario;
```

Create a platform with default property values and add it to the scenario.

```
plat = platform(rs);
```

Specify the trajectory of the platform as a circular path of radius 10 m for one second. This is accomplished by placing waypoints in a circular shape, ensuring that the first and last waypoint are the same.

```
wpts = [0 10 0; 10 0 0; 0 -10 0; -10 0 0; 0 10 0];
times = [0; 0.25; .5; .75; 1.0];
plat.Trajectory = waypointTrajectory(wpts,times);
```

Display the properties of the platform object.

```
plat
```

```
plat =
  Platform with properties:
    PlatformID: 1
    ClassID: 0
    Position: [0 10 0]
    Orientation: [-1.7180e-05 0 0]
    Dimensions: [1x1 struct]
    Trajectory: [1x1 waypointTrajectory]
    PoseEstimator: [1x1 insSensor]
    Emitters: {}
```

```
Sensors: {}  
Signatures: {[1x1 rcsSignature]}
```

Perform the simulation, advancing one time step at a time. Display the simulation time and the position and velocity of the platform at each time step.

```
while advance(rs)  
    p = pose(plat);  
    disp(strcat("Time = ",num2str(rs.SimulationTime)))  
    disp(strcat("  Position = [",num2str(p.Position),"]"))  
    disp(strcat("  Velocity = [",num2str(p.Velocity),"]"))  
end
```

```
Time = 0  
  Position = [0  10  0]  
  Velocity = [62.8318 -1.88403e-05  0]  
Time = 0.1  
  Position = [5.8779  8.0902  0]  
  Velocity = [50.832  -36.9316  0]  
Time = 0.2  
  Position = [9.5106  3.0902  0]  
  Velocity = [19.4161  -59.7566  0]  
Time = 0.3  
  Position = [9.5106  -3.0902  0]  
  Velocity = [-19.4161  -59.7567  0]  
Time = 0.4  
  Position = [5.8779  -8.0902  0]  
  Velocity = [-50.832  -36.9316  0]  
Time = 0.5  
  Position = [0 -10  0]  
  Velocity = [-62.8319  1.88181e-05  0]  
Time = 0.6  
  Position = [-5.8779  -8.0902  0]  
  Velocity = [-50.832  36.9316  0]  
Time = 0.7  
  Position = [-9.5106  -3.0902  0]  
  Velocity = [-19.4161  59.7566  0]  
Time = 0.8
```

```

    Position = [-9.5106      3.0902      0]
    Velocity = [19.4161     59.7566     0]
Time = 0.9
    Position = [-5.8779      8.0902      0]
    Velocity = [50.832      36.9316     0]
Time = 1
    Position = [-7.10543e-15      10      0]
    Velocity = [62.8319 -1.88404e-05      0]

```

### Create Cuboid Platforms with Circular Trajectory

Create a radar scenario.

```
rs = radarScenario;
```

Create a cuboid platform for a truck with dimensions 5 m by 2.5 m by 3.5 m.

```
dim1 = struct('Length',5,'Width',2.5,'Height',3.5,'OriginOffset',[0 0 0]);
truck = platform(rs,'Dimension',dim1);
```

Specify the trajectory of the truck as a circle with radius 20 m.

```
truck.Trajectory = waypointTrajectory('Waypoints', ...
    [20*cos(2*pi*(0:10)'/10) 20*sin(2*pi*(0:10)'/10) -1.75*ones(11,1)], ...
    'TimeOfArrival',linspace(0,50,11)');
```

Create the platform for a small quadcopter with dimensions 0.3 m by 0.3 m by 0.1 m.

```
dim2 = struct('Length',.3,'Width',.3,'Height',.1,'OriginOffset',[0 0 0]);
quad = platform(rs,'Dimension',dim2);
```

Specify the trajectory of the quadcopter as a circle 10 m above the truck with a small angular delay. Note that the negative z coordinates correspond to positive elevation.

```
quad.Trajectory = waypointTrajectory('Waypoints', ...
    [20*cos(2*pi*((0:10)'-.6)/10) 20*sin(2*pi*((0:10)'-.6)/10) -11.80*ones(11,1)], ...
    'TimeOfArrival',linspace(0,50,11)');
```

Visualize the results using theaterPlot.

```
tp = theaterPlot('XLim',[-30 30],'YLim',[-30 30],'Zlim',[-12 5]);
pp1 = platformPlotter(tp,'DisplayName','truck','Marker','s');
pp2 = platformPlotter(tp,'DisplayName','quadcopter','Marker','o');
```

Specify a view direction and run the simulation.

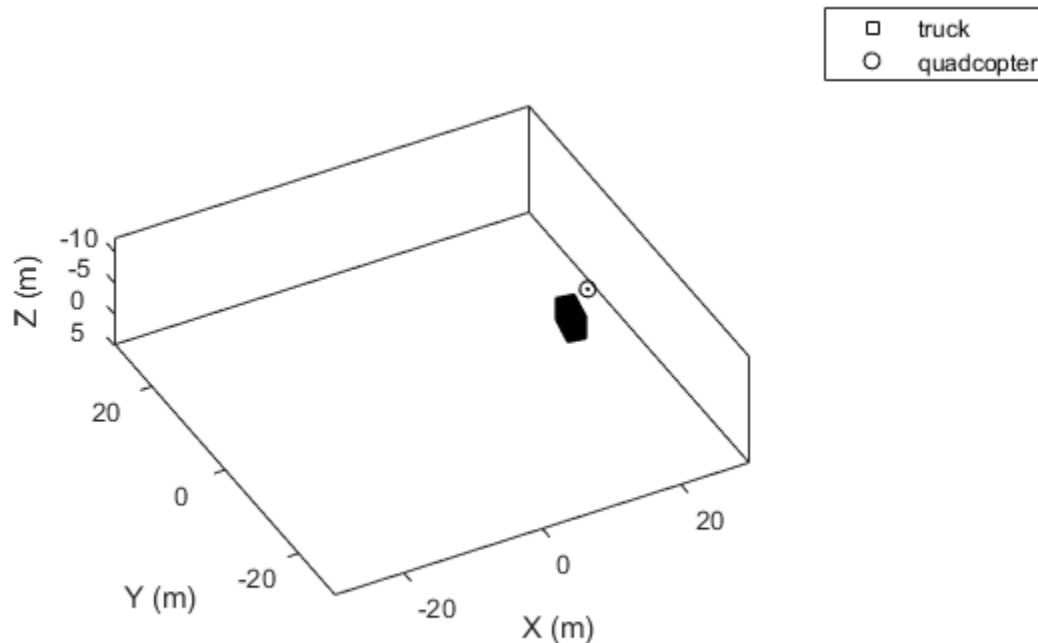
```
view(-28,37);
set(gca,'Zdir','reverse');
```

```
while advance(rs)
```

```

poses = platformPoses(rs);
plotPlatform(pp1,poses(1).Position,truck.Dimensions,poses(1).Orientation);
plotPlatform(pp2,poses(2).Position,quad.Dimensions,poses(2).Orientation);
end

```



## Input Arguments

### scenario — Radar scenario

radarScenario object

Radar scenario, specified as a radarScenario object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1, ..., NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'ClassID',2

### ClassID — Platform classification identifier

0 (default) | nonnegative integer

Platform classification identifier, specified as a nonnegative integer. You can define your own platform classification scheme and assign `ClassID` values to platforms according to the scheme. The value of `0` is reserved for an object of unknown or unassigned class.

Example: 5

Data Types: `double`

### Trajectory – Platform motion

`kinematicTrajectory` object | `waypointTrajectory` object | `geoTrajectory` object

Platform motion, specified as a `kinematicTrajectory` object, a `waypointTrajectory` object, or a `geoTrajectory` object. The trajectory object defines the time evolution of the position and velocity of the platform frame origin, as well as the orientation of the platform frame relative to the scenario frame.

- When the `IsEarthCentered` property of the scenario is set to `false`, use the `kinematicTrajectory` or the `waypointTrajectory` object. By default, the platform uses a stationary `kinematicTrajectory` object.
- When the `IsEarthCentered` property of the scenario is set to `true`, use the `geoTrajectory` object. By default, the platform uses a stationary `geoTrajectory` object.

### Position – Position of platform

three-element vector of scalars

This property is read-only.

Current position of the platform, specified as a three-element vector of scalars.

- When the `IsEarthCentered` property of the scenario is set to `false`, the position is specified as a three-element Cartesian state `[x, y, z]` in meters.
- When the `IsEarthCentered` property of the scenario is set to `true`, the position is specified as a three-element geodetic state: `latitude` in degrees, `longitude` in degrees, and `altitude` in meters.

Specify this argument only when creating a stationary platform. If you choose to specify the trajectory of the platform, use the `Trajectory` argument.

Data Types: `double`

### Orientation – Orientation of platform

three-element numeric vector

This property is read-only.

Orientation of the platform, specified as a three-element numeric vector in degrees. The three elements are the `[yaw, pitch, roll]` rotation angles from the local reference frame to the body frame of the platform.

Specify this argument only when creating a stationary platform. If you choose to specify the orientation over time, use the `Trajectory` argument.

Data Types: `double`

### Signatures – Platform signatures

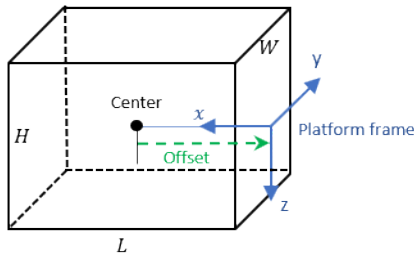
cell array of signature objects | `{}`

Platform signatures, specified as a cell array of signature objects or an empty cell array (`{}`). The default value is a cell array containing an `rcsSignature` object with default property values. If you have Sensor Fusion and Tracking Toolbox, then the cell array can also include `irSignature` and `tsSignature` objects. The cell array contains at most one instance for each type of signature object. A signature represents the reflection or emission pattern of a platform such as its radar cross-section, target strength, or IR intensity.

### Dimensions — Platform dimensions and origin offset

structure

Platform dimensions and origin offset, specified as a structure. The structure contains the `Length`, `Width`, `Height`, and `OriginOffset` of a cuboid that approximates the dimensions of the platform. The `OriginOffset` is the position vector from the center of the cuboid to the origin of the platform coordinate frame. The `OriginOffset` is expressed in the platform coordinate system. For example, if the platform origin is at the center of the cuboid rear face as shown in the figure, then set `OriginOffset` as `[-L/2, 0, 0]`. The default value for `Dimensions` is a structure with all fields set to zero, which corresponds to a point model.



### Fields of Dimensions

Fields	Description	Default
Length	Dimension of a cuboid along the $x$ direction	0
Width	Dimension of a cuboid along the $y$ direction	0
Height	Dimension of a cuboid along the $z$ direction	0
OriginOffset	Position of the platform coordinate frame origin with respect to the cuboid center	[0 0 0]

Example: `struct('Length',5,'Width',2.5,'Height',3.5,'OriginOffset',[-2.5 0 0])`

Data Types: `struct`

### PoseEstimator — Platform pose estimator

`insSensor` object (default) | pose estimator object

Platform pose estimator, specified as a pose-estimator object such as an `insSensor` object. The pose estimator determines platform pose with respect to the local NED scenario coordinates. The interface of any pose estimator must match the interface of the `insSensor` object. By default, the platform sets the pose estimator accuracy properties to zero.

**Emitters — Emitters mounted on platform**

cell array of emitter objects

Emitters mounted on the platform, specified as a cell array of emitter objects such as `radarEmitter` objects. If you have Sensor Fusion and Tracking Toolbox, then the cell array can also include `sonarEmitter` objects.

**Sensors — Sensors mounted on platform**

cell array of sensor objects

Sensors mounted on the platform, specified as a cell array of sensor objects such as `radarDataGenerator` objects.

**Output Arguments****plat — Scenario platform**

Platform object

Scenario platform, returned as a `Platform` object.

**See Also**

`Platform` | `radarScenario` | `waypointTrajectory` | `rscSignature` | `insSensor` | `radarEmitter` | `radarDataGenerator`

**Introduced in R2021a**

## platformProfiles

Profiles of radar scenario platforms

### Syntax

```
profiles = platformProfiles(scenario)
```

### Description

`profiles = platformProfiles(scenario)` returns the profiles of all platforms in the radar scenario, `scenario`.

### Examples

#### Generate Platform Profiles from Radar Scenario

Create a radar scenario.

```
rs = radarScenario;
```

Add two platforms to the scenario. Specify the `ClassID` of the second platform as 3.

```
p1 = platform(rs);  
p2 = platform(rs);  
p2.ClassID = 3;
```

Extract the profiles for all platforms in the scenario.

```
profiles = platformProfiles(rs)  
  
profiles=1x2 struct array with fields:  
    PlatformID  
    ClassID  
    Dimensions  
    Signatures
```

### Input Arguments

#### **scenario** — Radar scenario

radarScenario object

Radar scenario, specified as a `radarScenario` object.

### Output Arguments

#### **profiles** — Platform profiles

array of structures



Profiles of all platforms in the radar scenario, returned as an array of structures. The number of structures in the array is equal to the number of platforms. Each profile contains the signatures of a platform and identifying information. Each structure contains these fields.

Field	Description
PlatformID	Scenario-defined platform identifier, defined as a positive integer
ClassID	User-defined platform classification identifier, defined as a nonnegative integer
Dimensions	Platform dimensions, defined as a structure with these fields: <ul style="list-style-type: none"> <li>• Length</li> <li>• Width</li> <li>• Height</li> <li>• OriginOffset</li> </ul>
Signatures	Platform signatures, defined as a cell array of signature objects such as <code>rcsSignature</code> objects

See `Platform` for more information about the fields.

### See Also

`radarScenario` | `Platform` | `platform` | `platformPoses`

**Introduced in R2021a**

## platformPoses

Position information for each platform in radar scenario

### Syntax

```
poses = platformPoses(scenario)
poses = platformPoses(scenario, format)
poses = platformPoses( ____, 'CoordinateSystem', coordinateSystem)
```

### Description

`poses = platformPoses(scenario)` returns the current poses for all platforms in the radar scenario, `scenario`. Pose is the position, velocity, and orientation of a platform relative to scenario coordinates.

`poses = platformPoses(scenario, format)` also specifies the format of the returned platform orientation as 'quaternion' or 'rotmat'.

`poses = platformPoses( ____, 'CoordinateSystem', coordinateSystem)` specifies the coordinate system of the poses output argument. You can use this syntax only when the `IsEarthCentered` property of the radar scenario is set to `true`.

### Examples

#### Get Pose of Platforms in Radar Scenario

Create a radar scenario.

```
rs = radarScenario;
```

Add a platform to the scenario.

```
plat = platform(rs);
plat.Trajectory.Position = [1 1 0];
plat.Trajectory.Orientation = quaternion([90 0 0], 'eulerd', 'ZYX', 'frame');
```

Extract the pose of the platform from the radar scenario.

```
poses = platformPoses(rs)

poses = struct with fields:
    PlatformID: 1
    ClassID: 0
    Position: [1 1 0]
    Velocity: [0 0 0]
    Acceleration: [0 0 0]
    Orientation: [1x1 quaternion]
    AngularVelocity: [0 0 0]
```

## Get Platform Orientation in Matrix Format

Create a radar scenario.

```
rs = radarScenario;
```

Add a platform to the scenario.

```
plat = platform(rs);
plat.Trajectory.Position = [1 1 0];
plat.Trajectory.Orientation = quaternion([90 0 0], 'eulerd', 'ZYX', 'frame');
```

Extract the pose orientation in matrix format.

```
poses = platformPoses(rs, 'rotmat');
poses.Orientation
```

```
ans = 3×3
```

```
    0.0000    1.0000         0
   -1.0000    0.0000         0
         0         0    1.0000
```

## Input Arguments

### scenario — Radar scenario

radarScenario object

Radar scenario, specified as a radarScenario object.

### format — Pose orientation format

'quaternion' (default) | 'rotmat'

Pose orientation format, specified as 'quaternion' or 'rotmat'. When specified as 'quaternion', the Orientation field of the platform pose structure is a quaternion. When specified as 'rotmat', the Orientation field is a rotation matrix.

Data Types: char | string

### coordinateSystem — Coordinate system

'Cartesian' (default) | 'Geodetic'

Coordinate system in which the function reports poses, specified as one of these values:

- 'Cartesian' — Report poses using Cartesian coordinates in the Earth-Centered-Earth-Fixed coordinate frame.
- 'Geodetic' — Report positions using geodetic coordinates (latitude, longitude, and altitude). Report orientation, velocity, and acceleration in the local reference frame of each platform (North-East-Down by default) corresponding to the current waypoint.

Specify this argument only when the IsEarthCentered property of the radar scenario, scenario, is set to true.

Data Types: char | string

## Output Arguments

### poses — Platform poses in scenario coordinates

structure | array of structures

Poses of all platforms in the radar scenario, returned as a structure or an array of structures. Each structure contains these fields.

Field	Description
PlatformID	Unique identifier for the platform, specified as a positive integer. This is a required field with no default value.
ClassID	User-defined integer used to classify the type of target, specified as a nonnegative integer. Zero is reserved for unclassified platform types and is the default value.
Position	Position of target in scenario coordinates, specified as a real-valued 1-by-3 row vector. <ul style="list-style-type: none"> <li>If the <code>coordinateSystem</code> argument is specified as 'Cartesian', then <code>Position</code> is a three-element vector of Cartesian position coordinates in meters.</li> <li>If the <code>coordinateSystem</code> argument is specified as 'Geodetic', then <code>Position</code> is a three-element vector of geodetic coordinates: latitude in degrees, longitude in degrees, and altitude in meters.</li> </ul>
Velocity	Velocity of platform in scenario coordinates, specified as a real-valued 1-by-3 row vector. Units are meters per second. The default value is <code>[0 0 0]</code> .
Acceleration	Acceleration of the platform in scenario coordinates, specified as a 1-by-3 row vector in meters per second squared. The default value is <code>[0 0 0]</code> .
Orientation	Orientation of the platform with respect to the local scenario navigation frame, specified as a scalar quaternion or a 3-by-3 rotation matrix. Orientation defines the frame rotation from the local navigation coordinate system to the current platform body coordinate system. Units are dimensionless. The default value is <code>quaternion(1,0,0,0)</code> .

Field	Description
AngularVelocity	Angular velocity of the platform in scenario coordinates, specified as a real-valued 1-by-3 vector. The magnitude of the vector defines the angular speed. The direction defines the axis of clockwise rotation. Units are degrees per second. The default value is [0 0 0].

Data Types: struct

### See Also

radarScenario | platform | Platform | platformProfiles

**Introduced in R2021a**

## propagate

Propagate emissions in radar scenario

### Syntax

```
propEmissions = propagate(scenario,emissions)
propEmissions = propagate(scenario,emissions,'HasOcclusion',tf)
```

### Description

`propEmissions = propagate(scenario,emissions)` returns propagated emissions that are a combination of the input emissions and the reflections of these input emissions from the platforms in the radar scenario, `scenario`.

`propEmissions = propagate(scenario,emissions,'HasOcclusion',tf)` specifies whether the radar channel models occlusion or not. By default, the radar channel models occlusion.

### Examples

#### Propagate Emissions from Two Platforms in Radar Scenario

Create a radar scenario and add two platforms. Set the position and add an emitter to each platform.

```
rs = radarScenario('UpdateRate',1);
plat1 = platform(rs);
plat1.Trajectory.Position = [0,0,0];
emitter1 = radarEmitter(1,'UpdateRate',1);
plat1.Emitters = emitter1;
plat2 = platform(rs);
plat2.Trajectory.Position = [100,0,0];
emitter2 = radarEmitter(2,'UpdateRate',1);
plat2.Emitters = emitter2;
```

Advance the radar scenario, generate emissions, and obtain propagated emissions.

```
advance(rs);
emtx = emit(rs);
emprop = propagate(rs,emtx,'HasOcclusion',true)
```

```
emprop=3x1 cell array
    {1x1 radarEmission}
    {1x1 radarEmission}
    {1x1 radarEmission}
```

Display the last propagated emission in the radar scenario. The last emission is emitted by emitter 1 and reflected from platform 2.

```
disp(emprop{end})
    radarEmission with properties:
```

```

PlatformID: 2
EmitterIndex: 1
OriginPosition: [100 0 0]
OriginVelocity: [0 0 0]
Orientation: [1x1 quaternion]
FieldOfView: [180 180]
CenterFrequency: 300000000
Bandwidth: 3000000
WaveformType: 0
ProcessingGain: 0
PropagationRange: 100.0313
PropagationRangeRate: 0
EIRP: 38.0131
RCS: 10

```

## Input Arguments

### **scenario** — Radar scenario

radarScenario object

Radar scenario, specified as a radarScenario object.

### **emissions** — Emissions in radar scenario

cell array of emission objects

Emissions in the radar scenario, specified as a cell array of emission objects, such as radarEmission objects. You can obtain emissions from a radar scenario using the `emit` function.

### **tf** — Radar channel models occlusion

true or 1 (default) | false or 0

Radar channel models occlusion, specified as a numeric or logical 1 (true) or 0 (false).

## Output Arguments

### **propEmissions** — Propagated emissions

cell array of emission objects

Propagated emissions in the radar scenario, specified as a cell array of emission objects, such as radarEmission objects. The propagated emissions contain the source emissions and the emissions reflected from the platforms.

## See Also

radarScenario | emit | detect | radarEmission | radarChannel

### Introduced in R2021a

## receive

Receive IQ signal from radars in the scenario

### Syntax

```
sig = receive(scenario)
[sig,info] = receive(scenario)
[sig,info,pids] = receive(scenario)
```

### Description

`sig = receive(scenario)` returns the target echos, `sig`, received at radars in the scenario.

`[sig,info] = receive(scenario)` also returns a structure array of configurations at each radar, `info`, of each radar when the signal is received.

`[sig,info,pids] = receive(scenario)` also returns a column array of platform IDs `pids`.

### Examples

#### Received Platform Echo in Radar Scenario

Obtain the signal from two platforms in the a radar scenario using a radar detection generator. Set the random number seed to insure the repeatability of the data.

```
s = rng(0);
scenario = radarScenario('UpdateRate',1);
plat1 = platform(scenario);
plat1.Trajectory.Position = [0,0,0];
plat1.Sensors = radarDataGenerator(1);
plat2 = platform(scenario);
plat2.Trajectory.Position = [1000,0,0];
[sig,sensorConfs] = receive(scenario)

sig = 1x1 cell array
    {1001x1 double}

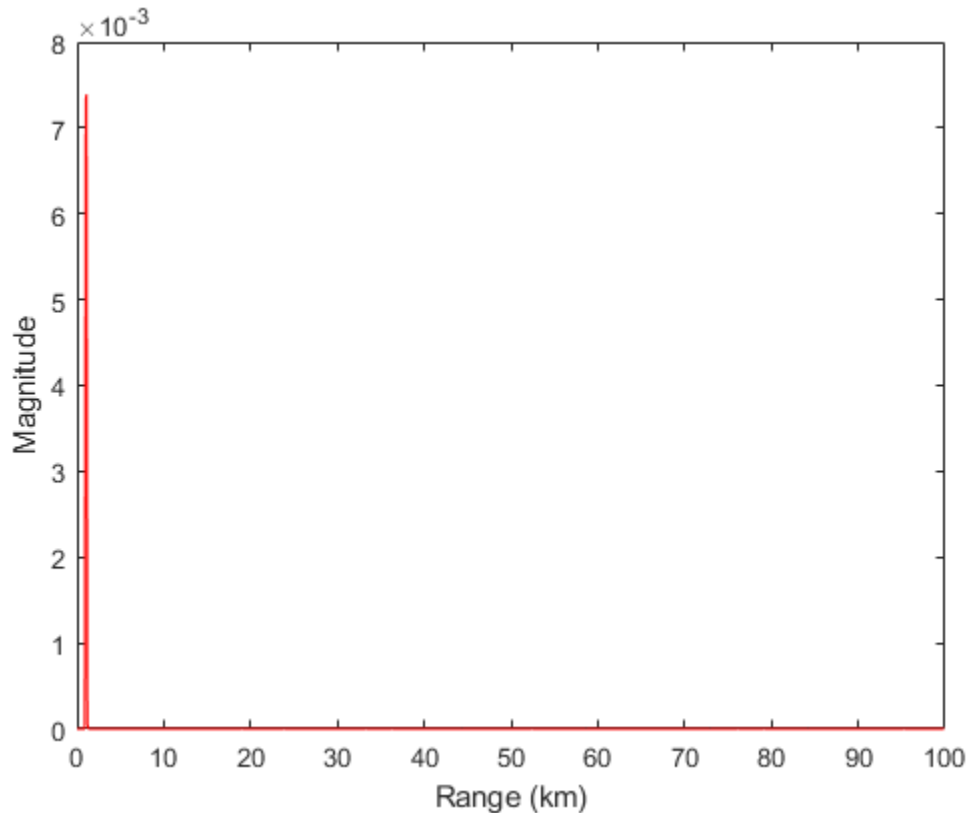
sensorConfs = struct with fields:
    IsScanDone: 0
    MechanicalAngle: 0
    ElectronicAngle: [2x1 double]
    OriginPosition: [3x1 double]
    Orientation: [3x3 double]
    ReferenceSensorIndex: 1
    Sensor: [1x1 radarTransceiver]
```

Platform 1 receives echo from platform 2:

```
rgrid = physconst('lightspeed')*(0:size(sig{1},1)-1)/ ...
    (2*sensorConfs.Sensor.Waveform.SampleRate)/1e3;
```



```
plot(rgrid,abs(sig{1}),'r')
xlabel('Range (km)')
ylabel('Magnitude')
```



### Signal from Two Platforms in Radar Scenario

Obtain the signal from two platforms in a radar scenario using a `radarTransceiver`. Set the seed in the random number generator to obtain repeatable data.

```
s = rng(0);
scenario = radarScenario('UpdateRate', 1);
plat1 = platform(scenario);
plat1.Trajectory.Position = [0,0,0];
plat1.Sensors = radarTransceiver;
plat2 = platform(scenario);
plat2.Trajectory.Position = [1000,0,0];
```

Platform 1 receives echo from platform 2. Obtain the received signal.

```
[sig,sensorConfs,sensorConfPIDs] = receive(scenario)
```

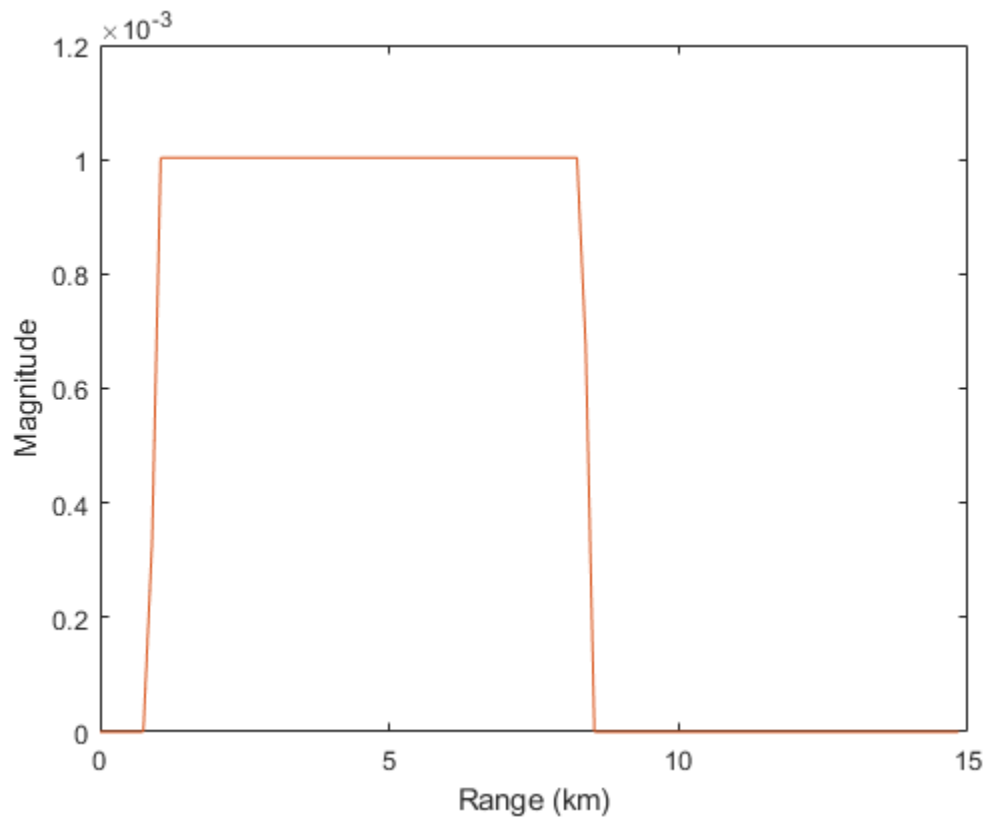
```
sig = 1x1 cell array
      {100x2 double}
```

```
sensorConfs = struct with fields:
    IsScanDone: 0
    MechanicalAngle: 0
    ElectronicAngle: [2x1 double]
    OriginPosition: [3x1 double]
    Orientation: [3x3 double]
    ReferenceSensorIndex: 0
    Sensor: [1x1 radarTransceiver]
```

```
sensorConfPIDs = 1
```

Plot the received signal.

```
rgrid = physconst('lightspeed')*(0:size(sig{1},1)-1)/ ...
    (2*sensorConfs.Sensor.Waveform.SampleRate)/1e3;
plot(rgrid,abs(sig{1}))
xlabel('Range (km)')
ylabel('Magnitude')
```



Return the random number generator to its previous state

```
rng(s)
```

## Input Arguments

### **scenario** — Radar scenario

radarScenario object

Radar scenario, specified as a radarScenario object.

## Output Arguments

### **sig** — Signal received at radar receivers

cell array

Signal received at the radar receiver, returned as a cell array. Each element of the cell array is the received echoes at each radar.

### **info** — Radar configurations array

structure array

Radar configurations, returned as a structure array. Each structure contains the following fields:

- **IsScanDone** -- Whether one period of mechanical scan is done
- **MechanicalAngle** -- Current antenna pointing angle due to mechanical scan
- **Origin** -- Radar location in the platform coordinate system
- **Orientation** -- Radar orientation axes in the platform coordinate system

Data Types: struct

### **pids** — Platform IDs

column vector of real values

Platform IDs on which radars are mounted, returned as a column vector of real values.

Data Types: double

## See Also

radarTransceiver | radarScenario | detect | emit | propagate | radarEmission

**Introduced in R2021a**

## record

Record simulation of radar scenario

### Syntax

```
rec = record(scenario)
rec = record(scenario,format)
rec = record( ____,Name,Value)
```

### Description

`rec = record(scenario)` returns a record, `rec`, of the evolution of the radar scenario simulation, `scenario`. The function starts from the beginning of the simulation and stores the record until the end of the simulation. A scenario simulation ends when either the `StopTime` of the scenario is reached or any platform in the scenario has finished its trajectory as specified by the `Trajectory` property.

---

**Note** The `record` function only records detections generated from sensors contained in the scenario and does not record tracks generated from a `radarDataGenerator` object contained in the scenario. `radarDataGenerator` generates detections when you set its `TargetReportFormat` property to 'Detections' or 'Clustered Detections' and generates tracks when you set its `TargetReportFormat` property to 'Tracks'.

---

`rec = record(scenario,format)` also specifies the format of the returned platform orientation.

`rec = record( ____,Name,Value)` specifies additional recording quantities using name-value arguments.

### Examples

#### Record Radar Scenario

Create a new radar scenario.

```
scenario = radarScenario;
```

Add a platform that follows a 25 m trajectory along the x-axis at 20 m/s.

```
plat = platform(scenario);
plat.Trajectory = waypointTrajectory('Waypoints',[0 0 0; 25 0 0], ...
    'TimeOfArrival',[0 25/20]);
```

Run the simulation and record the results.

```
r = record(scenario);
```

Show the platform states at the initial time.

```
r(1)
```

```
ans = struct with fields:
  SimulationTime: 0
  Poses: [1x1 struct]
```

`r(1).Poses`

```
ans = struct with fields:
  PlatformID: 1
  ClassID: 0
  Position: [0 0 0]
  Velocity: [20 0 0]
  Acceleration: [0 0 0]
  Orientation: [1x1 quaternion]
  AngularVelocity: [0 0 0]
```

Show the platform states at the final time.

`r(end)`

```
ans = struct with fields:
  SimulationTime: 1.2000
  Poses: [1x1 struct]
```

`r(end).Poses`

```
ans = struct with fields:
  PlatformID: 1
  ClassID: 0
  Position: [24 0 0]
  Velocity: [20 0 0]
  Acceleration: [0 0 0]
  Orientation: [1x1 quaternion]
  AngularVelocity: [0 0 0]
```

## Input Arguments

### scenario — Radar scenario

radarScenario object

Radar scenario, specified as a radarScenario object.

### format — Pose orientation format

'quaternion' (default) | 'rotmat'

Pose orientation format, specified as 'quaternion' or 'rotmat'. When specified as 'quaternion', the Orientation field of the platform pose structure is a quaternion. When specified as 'rotmat', the Orientation field is a rotation matrix.

Data Types: char | string

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `'CoordinateSystem', 'Geodetic'` reports recorded poses using geodetic coordinates

### IncludeEmitters — Enable recording emission information

`false` (default) | `true`

Enable recording emission information, specified as `true` or `false`. When specified as `true`, the rec output contains `Emissions`, `EmitterConfigurations`, `EmitterPlatformIDs`, and `CoverageConfig` fields.

### IncludeSensors — Enable recording sensor information

`false` (default) | `true`

Enable recording sensor information, specified as `true` or `false`. When specified as `true`, the rec output contains `Detections`, `SensorConfiguration`, `SensorPlatformIDs`, and `CoverageConfig` fields.

### InitialSeed — Initial random seed for recording

current random seed (default) | positive integer

Initial random seed for recording, specified as a positive integer. If specified as a positive integer, the function assigns this number to the random number generator "Twister" before the recording and resets the random number generator at the end of the recording.

### HasOcclusion — Enable occlusion in signal transmission

`true` (default) | `false`

Enable occlusion in signal transmission, specified as `true` or `false`. When specified as `true`, the function accounts for the effect of occlusion in radar emission propagation.

### RecordingFormat — Format of recording

`'Struct'` (default) | `'Recording'`

Format of recording, specified as `'Struct'` or `'Recording'`. When specified as `'Struct'`, the rec output is an array of structures. When specified as `'Recording'`, the rec output is a `radarScenarioRecording` object.

### CoordinateSystem — Coordinate system to report recorded poses

`'Cartesian'` (default) | `'Geodetic'`

Coordinate system to report recorded positions, specified as one of these values.

- `'Cartesian'` — Report recorded poses using Cartesian coordinates in the Earth-Centered-Earth-Fixed coordinate frame.
- `'Geodetic'` — Report recorded positions using geodetic coordinates (latitude, longitude, and altitude). Report recorded orientation, velocity, and acceleration in the local reference frame of each platform (North-East-Down by default) corresponding to the current waypoint.

Specify this argument only when the `IsEarthCentered` property of the radar scenario, `scenario`, is set to `true`.

## Output Arguments

### **rec** — Records of platform states during simulation

*M*-by-1 array of structures | `radarScenarioRecording` object

Records of platform states during the simulation, returned as an *M*-by-1 array of structures if the `RecordingFormat` is specified as `'struct'` (default) or a `radarScenarioRecording` object if the `RecordingFormat` is specified as `'Recording'`. *M* is the number of time steps in the simulation.

Each record contains the simulation time step and the recorded information at that time. The record structure has at least two fields: `SimulationTime` and `Poses`. It can also have other optional fields depending on the values of the `'IncludeEmitters'` and `'IncludeSensors'` name-value arguments.

The `SimulationTime` field contains the simulation time of the record. `Poses` is an *N*-by-1 array of structures, where *N* is the number of platforms. Each structure in `Poses` contains these fields.

Field	Description
<code>PlatformID</code>	Unique identifier for the platform, specified as a positive integer. This is a required field with no default value.
<code>ClassID</code>	User-defined integer used to classify the type of target, specified as a nonnegative integer. Zero is reserved for unclassified platform types and is the default value.
<code>Position</code>	Position of target in scenario coordinates, specified as a real-valued 1-by-3 row vector. <ul style="list-style-type: none"> <li>If the <code>coordinateSystem</code> argument is specified as <code>'Cartesian'</code>, then <code>Position</code> is a three-element vector of Cartesian position coordinates in meters.</li> <li>If the <code>coordinateSystem</code> argument is specified as <code>'Geodetic'</code>, then <code>Position</code> is a three-element vector of geodetic coordinates: latitude in degrees, longitude in degrees, and altitude in meters.</li> </ul>
<code>Velocity</code>	Velocity of platform in scenario coordinates, specified as a real-valued 1-by-3 row vector. Units are meters per second. The default value is <code>[0 0 0]</code> .
<code>Acceleration</code>	Acceleration of the platform in scenario coordinates, specified as a 1-by-3 row vector in meters per second squared. The default value is <code>[0 0 0]</code> .

Field	Description
Orientation	Orientation of the platform with respect to the local scenario navigation frame, specified as a scalar quaternion or a 3-by-3 rotation matrix. Orientation defines the frame rotation from the local navigation coordinate system to the current platform body coordinate system. Units are dimensionless. The default value is <code>quaternion(1,0,0,0)</code> .
AngularVelocity	Angular velocity of the platform in scenario coordinates, specified as a real-valued 1-by-3 vector. The magnitude of the vector defines the angular speed. The direction defines the axis of clockwise rotation. Units are degrees per second. The default value is <code>[0 0 0]</code> .

The rec output contains these optional fields.

Field	Description
Emissions	Cell array of emissions (such as <code>radarEmission</code> objects) in the scenario
EmitterConfigurations	Structure array of emitter configurations for each emitter
EmitterPlatformIDs	Numeric array of platform IDs for each emitter
Detections	Cell array of <code>objectDetection</code> objects generated by the sensors in the scenario
SensorConfigurations	Structure array of sensor configurations for each sensor
SensorPlatformIDs	Numeric array of platform IDs for each sensor
CoverageConfig	Structure array of coverage configurations for each sensor or emitter

Each emitter configuration structure contains these fields.

Field	Description
EmitterIndex	Unique emitter index, returned as a positive integer.
IsValidTime	Valid emission time, returned as 0 or 1. <code>IsValidTime</code> is 0 when emitter updates are requested at times that are between update intervals specified by the <code>UpdateInterval</code> property.
IsScanDone	Whether the emitter has completed a scan, returned as <code>true</code> or <code>false</code> .
FieldOfView	Field of view of the emitter, returned as a two-element vector [azimuth; elevation] in degrees.



MeasurementParameters	Emitter measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.
-----------------------	---

Each sensor configuration structure contains these fields.

Field	Description
SensorIndex	Unique sensor index, returned as a positive integer.
IsValidTime	Valid detection time, returned as <code>true</code> or <code>false</code> . <code>IsValidTime</code> is <code>false</code> when detection updates are requested between update intervals specified by the update rate.
IsScanDone	<code>IsScanDone</code> is <code>true</code> when the sensor has completed a scan.
FieldOfView	Field of view of the sensor, returned as a 2-by-1 vector of positive real values, <code>[azfov;elfov]</code> . <code>azfov</code> and <code>elfov</code> represent the field of view in azimuth and elevation, respectively.
RangeLimits	Minimum and maximum range of sensor, in meters, specified as a 1-by-2 nonnegative real-valued vector of the form <code>[rmin,rmax]</code> .
RangeRateLimits	Minimum and maximum range rate of sensor, in meters per second, specified as a 1-by-2 real-valued vector of the form <code>[rrmin,rrmax]</code> .
MeasurementParameters	Sensor measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current sensor frame.

Each coverage configuration structure contains these fields.

Field	Description
Index	A unique integer to distinguish sensors or emitters. In practice, you can use <code>SensorIndex</code> or <code>EmitterIndex</code> property of the sensor or emitter objects, respectively.
LookAngle	The current boresight angles of the sensor or emitter, specified as one of these values: <ul style="list-style-type: none"> <li>A scalar in degrees if scanning only in the azimuth direction.</li> <li>A two-element vector <code>[azimuth;elevation]</code> in degrees if scanning both in the azimuth and elevation directions.</li> </ul>

Field	Description
FieldOfView	The field of view of the sensor or emitter, specified as a two-element vector [azimuth; elevation] in degrees.
ScanLimits	<p>The minimum and maximum angles the sensor or emitter can scan from its Orientation.</p> <ul style="list-style-type: none"> <li>• If the sensor or emitter can scan only in the azimuth direction, then specify the limits as a 1-by-2 row vector [minAz, maxAz] in degrees.</li> <li>• If the sensor or emitter can also scan in the elevation direction, then specify the limits as a 2-by-2 matrix [minAz, maxAz; minEl, maxEl] in degrees.</li> </ul>
Range	The range of the beam and coverage area of the sensor or emitter in meters.
Position	The origin position of the sensor or emitter, specified as a three-element vector [X, Y, Z] on the axes of the theater plot.
Orientation	The rotation transformation from the scenario or global frame to the sensor or emitter mounting frame, specified as a rotation matrix, a quaternion, or three Euler angles in ZYX sequence.

### See Also

radarScenario | restart | advance | platformPoses | coverageConfig | radarScenarioRecording

**Introduced in R2021a**

# restart

Restart simulation of radar scenario

## Syntax

```
restart(scenario)
```

## Description

`restart(scenario)` restarts the simulation of the radar scenario, `scenario`, from the beginning and sets the `SimulationTime` property of `scenario` to zero.

## Examples

### Restart Radar Scenario

Create a new radar scenario.

```
scenario = radarScenario;
```

Add a platform that follows a 25 m trajectory along the x-axis at 20 m/s.

```
plat = platform(scenario);
plat.Trajectory = waypointTrajectory('Waypoints',[0 0 0; 25 0 0], ...
    'TimeOfArrival',[0 25/20]);
```

Run the simulation to completion.

```
rec = record(scenario)
```

```
rec=13x1 struct array with fields:
    SimulationTime
    Poses
```

Display the scenario simulation time after the simulation is complete.

```
scenario.SimulationTime
```

```
ans = 1.3000
```

Restart the simulation and confirm that the scenario simulation time is reset to 0.

```
restart(scenario);
scenario.SimulationTime
```

```
ans = 0
```

## **Input Arguments**

### **scenario — Radar scenario**

radarScenario object

Radar scenario, specified as a radarScenario object.

## **See Also**

radarScenario | advance | record

**Introduced in R2021a**

# radarScenarioRecording

Return recording of radar scenario

## Description

Use the `radarScenarioRecording` object to record a radar scenario.

## Creation

You can create a `radarScenarioRecording` object in these ways:

- Create a recording of a `radarScenario` object by using the `record` function and specifying the 'RecordingFormat' name-value argument as 'Recording'.
- Create a recording from prerecorded radar scenario data by using the `radarScenarioRecording` function described here.

## Syntax

```
recording = radarScenarioRecording(recordedData)
recording = radarScenarioRecording(recordedData,Name,Value)
```

### Description

`recording = radarScenarioRecording(recordedData)` creates a `radarScenarioRecording` object using recorded data. The `recordedData` argument sets the value of the `RecordedData` property.

`recording = radarScenarioRecording(recordedData,Name,Value)` sets one or both of the `CurrentTime` and `CurrentStep` properties using name-value arguments. Enclose each property name in quotes.

## Properties

### RecordedData — Recorded data stored in recording object

structure

Recorded data stored in the recording object, specified as a structure. You can set this property only when creating the object. The fields of the structure are the same as the fields of the output of the `record` function of the `radarScenario` object.

### CurrentTime — Timestamp of latest read data

0 | nonnegative scalar

Timestamp of the latest read data, specified as a nonnegative scalar. When you call the `read` function on the object, the function reads the recorded data set that has `SimulationTime` larger than the `CurrentTime`.

**CurrentStep — Step index of latest read data**

0 | nonnegative integer

Step index of the latest read data, specified as a nonnegative integer. When you call the `read` function on the object, the function reads the data set with the next step index.

**Object Functions**

`isDone` Indicates end of radar scenario recording  
`read` Read next step from radar scenario recording  
`reset` Reset to beginning of radar scenario recording

**Examples****Run Recorded Radar Scenario**

Load prerecorded data from a radar scenario. The data is saved as a struct with the variable name `recordedData`. Create a `radarScenarioRecording` object using the recorded data.

```
load recordedRadarScenarioData.mat
recording = radarScenarioRecording(recordedData);
```

Construct a theater plot to display the recorded data using multiple plotters.

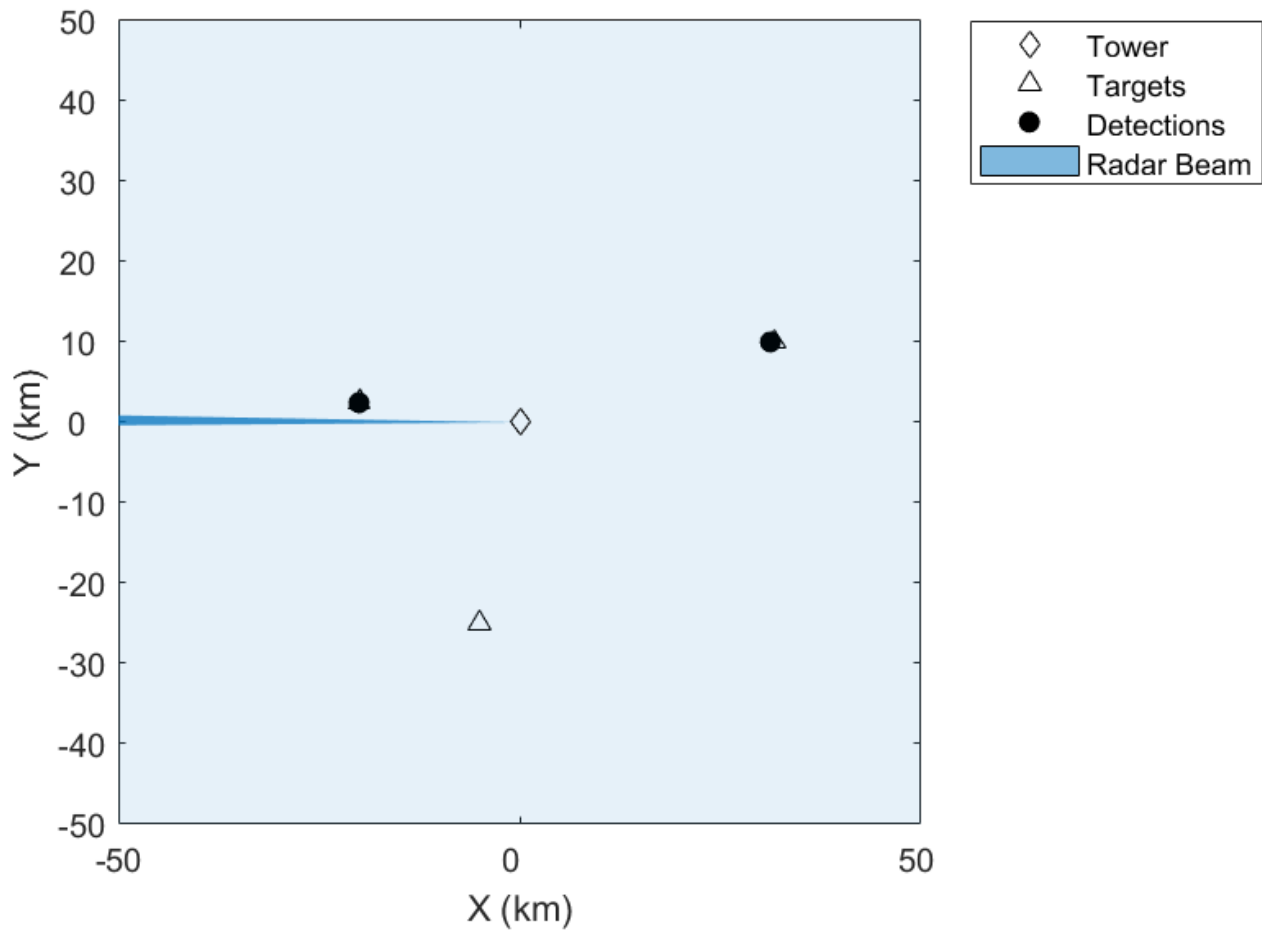
```
tp = theaterPlot('AxesUnits',["km" "km" "km"], ...
    'XLimits',[-50 50]*1e3,'YLimits',[-50 50]*1e3,'ZLimits',[-20 20]*1e3);
to = platformPlotter(tp,'DisplayName','Tower','Marker','d');
pp = platformPlotter(tp,'DisplayName','Targets');
dp = detectionPlotter(tp,'DisplayName','Detections','MarkerFaceColor','black');
cp = coveragePlotter(tp,'DisplayName','Radar Beam');
```

```
coverage = struct('Index',1,'LookAngle',[0;-7],'FieldOfView',[1;10], ...
    'ScanLimits',[0 365;-12 -2],'Range',100e3,'Position',[0;0;-15], ...
    'Orientation',eye(3));
```

Run the recorded scenario and animate the results.

```
scanBuffer = {};
while ~isDone(recording)
    % Step the reader to read the next frame of data
    [simTime,poses,covcon,dets,senconfig] = read(recording);
    scanBuffer = [scanBuffer;dets]; %#ok<AGROW>
    plotPlatform(to,poses(1).Position);
    plotPlatform(pp,reshape([poses(2:4).Position]',3,[]));
    plotCoverage(cp,covcon);
    if ~isempty(dets)
        plotDetection(dp,cell2mat(cellfun(@(c) c.Measurement(:)', scanBuffer, 'UniformOutput', false),
            scanBuffer, 'UniformOutput', false));
    end

    % Clear the buffer when a 360 degree scan is complete
    if senconfig.IsScanDone
        scanBuffer = {};
        dp.clearData;
    end
end
```

**See Also**

radarScenario | record

**Introduced in R2021a**

## isDone

Indicates end of radar scenario recording

### Syntax

```
tf = isDone(recording)
```

### Description

`tf = isDone(recording)` returns `true` if you have reached the end of data in the radar scenario recording and `false` otherwise. Use `isDone` to check if you have reached the end of the recording before reading the next step in the recording.

### Examples

#### Run Recorded Radar Scenario

Load prerecorded data from a radar scenario. The data is saved as a struct with the variable name `recordedData`. Create a `radarScenarioRecording` object using the recorded data.

```
load recordedRadarScenarioData.mat
recording = radarScenarioRecording(recordedData);
```

Construct a theater plot to display the recorded data using multiple plotters.

```
tp = theaterPlot('AxesUnits',["km" "km" "km"], ...
    'XLimits',[-50 50]*1e3,'YLimits',[-50 50]*1e3,'ZLimits',[-20 20]*1e3);
to = platformPlotter(tp,'DisplayName','Tower','Marker','d');
pp = platformPlotter(tp,'DisplayName','Targets');
dp = detectionPlotter(tp,'DisplayName','Detections','MarkerFaceColor','black');
cp = coveragePlotter(tp,'DisplayName','Radar Beam');
```

```
coverage = struct('Index',1,'LookAngle',[0;-7],'FieldOfView',[1;10], ...
    'ScanLimits',[0 365;-12 -2],'Range',100e3,'Position',[0;0;-15], ...
    'Orientation',eye(3));
```

Run the recorded scenario and animate the results.

```
scanBuffer = {};
while ~isDone(recording)
    % Step the reader to read the next frame of data
    [simTime,poses,covcon,dets,senconfig] = read(recording);
    scanBuffer = [scanBuffer;dets]; %#ok<AGROW>
    plotPlatform(to,poses(1).Position);
    plotPlatform(pp,reshape([poses(2:4).Position]',3,[]));
    plotCoverage(cp,covcon);
    if ~isempty(dets)
        plotDetection(dp,cell2mat(cellfun(@(c) c.Measurement(:)', scanBuffer, 'UniformOutput', false)),
            'Position',dets,'Measurement',scanBuffer);
    end

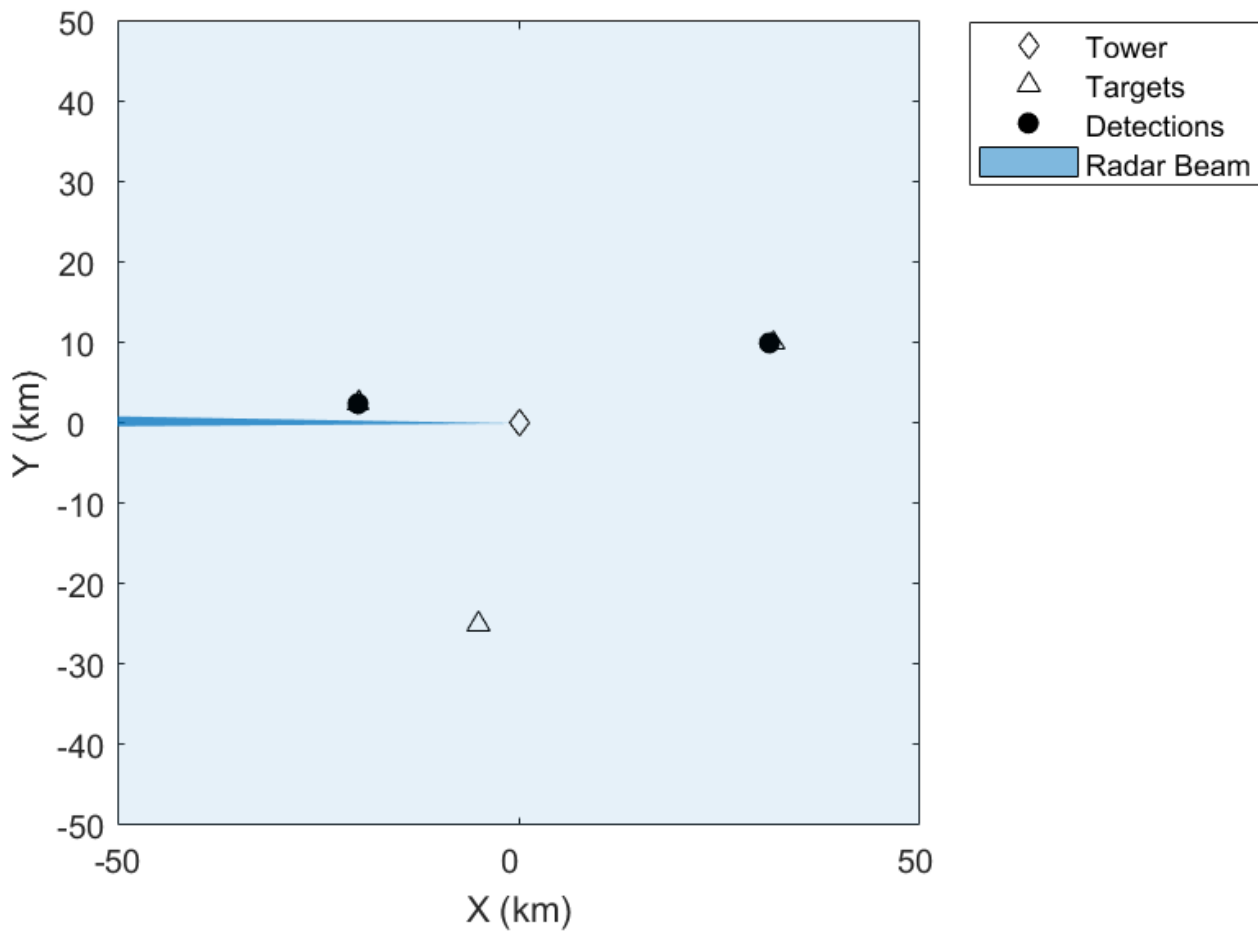
    % Clear the buffer when a 360 degree scan is complete
end
```



```

    if senconfig.IsScanDone
        scanBuffer = {};
        dp.clearData;
    end
end
end

```



## Input Arguments

### **recording** — Radar scenario recording

radarScenarioRecording object

Radar scenario recording, specified as a radarScenarioRecording object.

## Output Arguments

### **tf** — Recording has reached the end

true | false

Recording has reached the end, returned as true or false.

**See Also**

radarScenarioRecording

**Introduced in R2021a**

# read

Read next step from radar scenario recording

## Syntax

```
[simTime,poses,detections,sensorConfigs,sensorPlatformIDs,emissions,
emitterConfigs,emitterPlatformIDs] = read(recording)
```

## Description

[simTime,poses,detections,sensorConfigs,sensorPlatformIDs,emissions,emitterConfigs,emitterPlatformIDs] = read(recording) returns one recorded data set at the simulation time, simTime, from a radar scenario recording.

## Examples

### Run Recorded Radar Scenario

Load prerecorded data from a radar scenario. The data is saved as a struct with the variable name recordedData. Create a radarScenarioRecording object using the recorded data.

```
load recordedRadarScenarioData.mat
recording = radarScenarioRecording(recordedData);
```

Construct a theater plot to display the recorded data using multiple plotters.

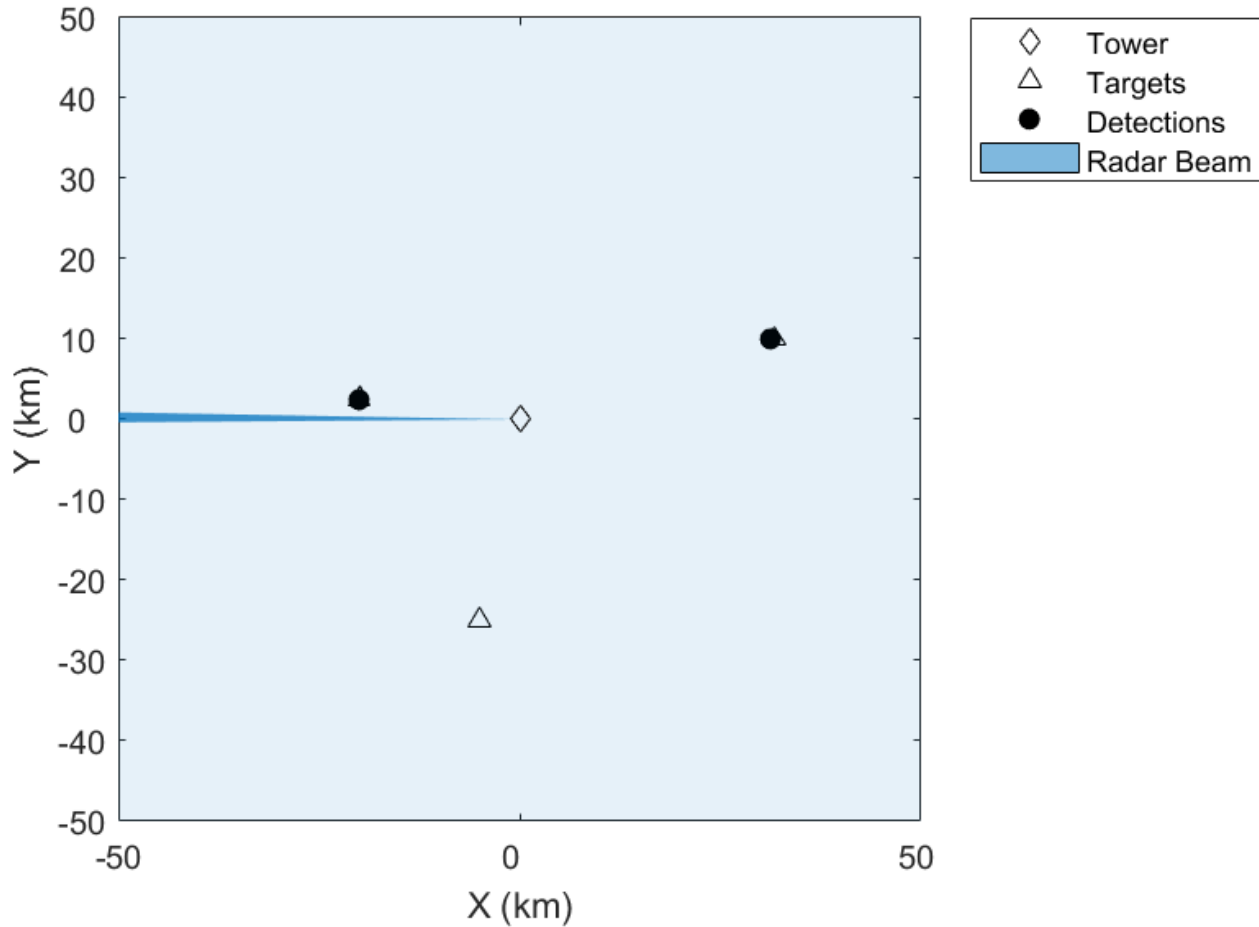
```
tp = theaterPlot('AxesUnits',["km" "km" "km"], ...
'XLimits',[-50 50]*1e3,'YLimits',[-50 50]*1e3,'ZLimits',[-20 20]*1e3);
to = platformPlotter(tp,'DisplayName','Tower','Marker','d');
pp = platformPlotter(tp,'DisplayName','Targets');
dp = detectionPlotter(tp,'DisplayName','Detections','MarkerFaceColor','black');
cp = coveragePlotter(tp,'DisplayName','Radar Beam');
```

```
coverage = struct('Index',1,'LookAngle',[0;-7],'FieldOfView',[1;10], ...
'ScanLimits',[0 365;-12 -2],'Range',100e3,'Position',[0;0;-15], ...
'Orientation',eye(3));
```

Run the recorded scenario and animate the results.

```
scanBuffer = {};
while ~isDone(recording)
    % Step the reader to read the next frame of data
    [simTime,poses,covcon,dets,senconfig] = read(recording);
    scanBuffer = [scanBuffer;dets]; %#ok<AGROW>
    plotPlatform(to,poses(1).Position);
    plotPlatform(pp,reshape([poses(2:4).Position]',3,[]));
    plotCoverage(cp,covcon);
    if ~isempty(dets)
        plotDetection(dp,cell2mat(cellfun(@(c) c.Measurement(:)', scanBuffer, 'UniformOutput', false),
end
```

```
% Clear the buffer when a 360 degree scan is complete
if senconfig.IsScanDone
    scanBuffer = {};
    dp.clearData;
end
end
```



## Input Arguments

### **recording** – Radar scenario recording

radarScenarioRecording object

Radar scenario recording, specified as a radarScenarioRecording object.

## Output Arguments

### **simTime** – Simulation time

nonnegative scalar

Simulation time, returned as a nonnegative scalar.

### poses — Poses of platforms

array of structures

Poses of platforms, returned as an array of structures. Each structure has these fields.

Field	Description
PlatformID	Unique identifier for the platform, specified as a positive integer. This is a required field with no default value.
ClassID	User-defined integer used to classify the type of target, specified as a nonnegative integer. Zero is reserved for unclassified platform types and is the default value.
Position	Position of target in scenario coordinates, specified as a real-valued 1-by-3 row vector. <ul style="list-style-type: none"> <li>If the <code>coordinateSystem</code> argument is specified as 'Cartesian', then <code>Position</code> is a three-element vector of Cartesian position coordinates in meters.</li> <li>If the <code>coordinateSystem</code> argument is specified as 'Geodetic', then <code>Position</code> is a three-element vector of geodetic coordinates: latitude in degrees, longitude in degrees, and altitude in meters.</li> </ul>
Velocity	Velocity of platform in scenario coordinates, specified as a real-valued 1-by-3 row vector. Units are meters per second. The default value is $[0 \ 0 \ 0]$ .
Acceleration	Acceleration of the platform in scenario coordinates, specified as a 1-by-3 row vector in meters per second squared. The default value is $[0 \ 0 \ 0]$ .
Orientation	Orientation of the platform with respect to the local scenario navigation frame, specified as a scalar quaternion or a 3-by-3 rotation matrix. Orientation defines the frame rotation from the local navigation coordinate system to the current platform body coordinate system. Units are dimensionless. The default value is <code>quaternion(1,0,0,0)</code> .
AngularVelocity	Angular velocity of the platform in scenario coordinates, specified as a real-valued 1-by-3 vector. The magnitude of the vector defines the angular speed. The direction defines the axis of clockwise rotation. Units are degrees per second. The default value is $[0 \ 0 \ 0]$ .

**detections – Detections**cell array of `objectDetection` objectsDetections, returned as a cell array of `objectDetection` objects.**sensorConfigs – Sensor configurations**

array of structures

Sensor configurations, returned as an array of structures. Each structure has these fields.

Field	Description
<code>SensorIndex</code>	Unique sensor index, returned as a positive integer.
<code>IsValidTime</code>	Valid detection time, returned as <code>true</code> or <code>false</code> . <code>IsValidTime</code> is <code>false</code> when detection updates are requested between update intervals specified by the update rate.
<code>IsScanDone</code>	<code>IsScanDone</code> is <code>true</code> when the sensor has completed a scan.
<code>FieldOfView</code>	Field of view of the sensor, returned as a 2-by-1 vector of positive real values, <code>[azfov;elfov]</code> . <code>azfov</code> and <code>elfov</code> represent the field of view in azimuth and elevation, respectively.
<code>RangeLimits</code>	Minimum and maximum range of sensor, in meters, specified as a 1-by-2 nonnegative real-valued vector of the form <code>[rmin,rmax]</code> .
<code>RangeRateLimits</code>	Minimum and maximum range rate of sensor, in meters per second, specified as a 1-by-2 real-valued vector of the form <code>[rrmin,rrmax]</code> .
<code>MeasurementParameters</code>	Sensor measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current sensor frame.

**sensorPlatformIDs – Platform IDs of sensors**

array of nonnegative integers

Platform IDs of sensors, returned as an array of nonnegative integers.

**emissions – Emissions**cell array of `emission` objectsEmissions, returned as a cell array of `emission` objects such as `radarEmission` objects.**emitterConfigs – Emitter configurations**

array of structures

Emitter configurations, returned as an array of structures. Each structure has these fields.

<b>Field</b>	<b>Description</b>
EmitterIndex	Unique emitter index, returned as a positive integer.
IsValidTime	Valid emission time, returned as 0 or 1. IsValidTime is 0 when emitter updates are requested at times that are between update intervals specified by the UpdateInterval property.
IsScanDone	Whether the emitter has completed a scan, returned as true or false.
FieldOfView	Field of view of the emitter, returned as a two-element vector [azimuth; elevation] in degrees.
MeasurementParameters	Emitter measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.

#### **emitterPlatformIDs — Platform IDs of emitters**

array of nonnegative integers

Platform IDs of emitters, returned as an array of nonnegative integers.

#### **See Also**

radarScenarioRecording

**Introduced in R2021a**

## **reset**

Reset to beginning of radar scenario recording

### **Syntax**

```
reset(recording)
```

### **Description**

`reset(recording)` resets the radar scenario recording to the beginning of the recording.

### **Input Arguments**

#### **recording – Radar scenario recording**

`radarScenarioRecording` object

Radar scenario recording, specified as a `radarScenarioRecording` object.

### **See Also**

`radarScenarioRecording`

**Introduced in R2021a**



# quaternion

Create a quaternion array

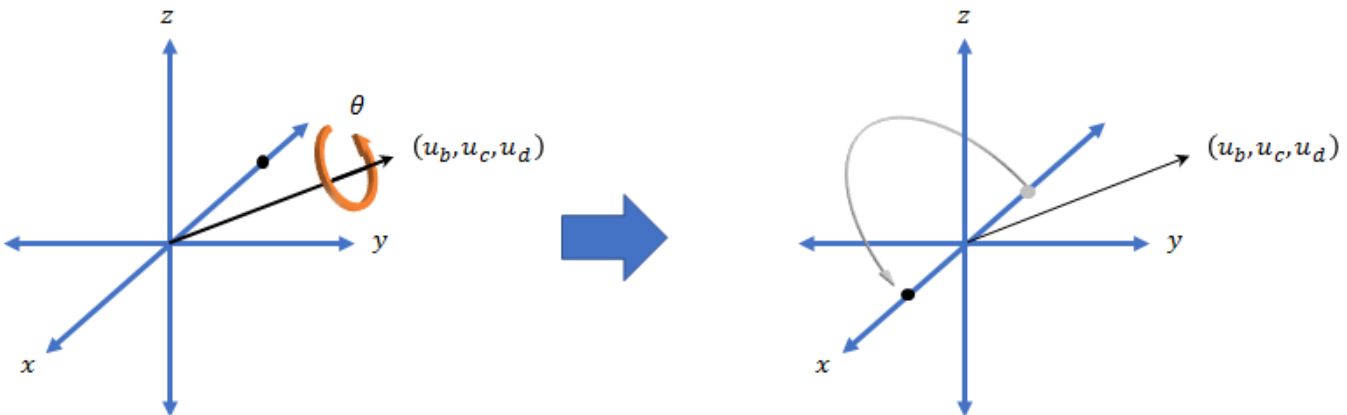
## Description

A quaternion is a four-part hyper-complex number used in three-dimensional rotations and orientations.

A quaternion number is represented in the form  $a + bi + cj + dk$ , where  $a$ ,  $b$ ,  $c$ , and  $d$  parts are real numbers, and  $i$ ,  $j$ , and  $k$  are the basis elements, satisfying the equation:  $i^2 = j^2 = k^2 = ijk = -1$ .

The set of quaternions, denoted by  $\mathbf{H}$ , is defined within a four-dimensional vector space over the real numbers,  $\mathbf{R}^4$ . Every element of  $\mathbf{H}$  has a unique representation based on a linear combination of the basis elements,  $i$ ,  $j$ , and  $k$ .

All rotations in 3-D can be described by an axis of rotation and angle about that axis. An advantage of quaternions over rotation matrices is that the axis and angle of rotation is easy to interpret. For example, consider a point in  $\mathbf{R}^3$ . To rotate the point, you define an axis of rotation and an angle of rotation.



The quaternion representation of the rotation may be expressed as  $q = \cos(\theta/2) + \sin(\theta/2)(u_b i + u_c j + u_d k)$ , where  $\theta$  is the angle of rotation and  $[u_b, u_c, \text{ and } u_d]$  is the axis of rotation.

## Creation

### Syntax

```
quat = quaternion()
quat = quaternion(A,B,C,D)
quat = quaternion(matrix)
quat = quaternion(RV, 'rotvec')
```

```
quat = quaternion(RV, 'rotvecd')
quat = quaternion(RM, 'rotmat', PF)
quat = quaternion(E, 'euler', RS, PF)
quat = quaternion(E, 'eulerd', RS, PF)
```

### Description

`quat = quaternion()` creates an empty quaternion.

`quat = quaternion(A,B,C,D)` creates a quaternion array where the four quaternion parts are taken from the arrays A, B, C, and D. All the inputs must have the same size and be of the same data type.

`quat = quaternion(matrix)` creates an  $N$ -by-1 quaternion array from an  $N$ -by-4 matrix, where each column becomes one part of the quaternion.

`quat = quaternion(RV, 'rotvec')` creates an  $N$ -by-1 quaternion array from an  $N$ -by-3 matrix of rotation vectors, RV. Each row of RV represents a rotation vector in radians.

`quat = quaternion(RV, 'rotvecd')` creates an  $N$ -by-1 quaternion array from an  $N$ -by-3 matrix of rotation vectors, RV. Each row of RV represents a rotation vector in degrees.

`quat = quaternion(RM, 'rotmat', PF)` creates an  $N$ -by-1 quaternion array from the 3-by-3-by- $N$  array of rotation matrices, RM. PF can be either 'point' if the Euler angles represent point rotations or 'frame' for frame rotations.

`quat = quaternion(E, 'euler', RS, PF)` creates an  $N$ -by-1 quaternion array from the  $N$ -by-3 matrix, E. Each row of E represents a set of Euler angles in radians. The angles in E are rotations about the axes in sequence RS.

`quat = quaternion(E, 'eulerd', RS, PF)` creates an  $N$ -by-1 quaternion array from the  $N$ -by-3 matrix, E. Each row of E represents a set of Euler angles in degrees. The angles in E are rotations about the axes in sequence RS.

### Input Arguments

#### A, B, C, D — Quaternion parts

comma-separated arrays of the same size

Parts of a quaternion, specified as four comma-separated scalars, matrices, or multi-dimensional arrays of the same size.

Example: `quat = quaternion(1,2,3,4)` creates a quaternion of the form  $1 + 2i + 3j + 4k$ .

Example: `quat = quaternion([1,5],[2,6],[3,7],[4,8])` creates a 1-by-2 quaternion array where `quat(1,1) = 1 + 2i + 3j + 4k` and `quat(1,2) = 5 + 6i + 7j + 8k`

Data Types: `single` | `double`

#### matrix — Matrix of quaternion parts

$N$ -by-4 matrix

Matrix of quaternion parts, specified as an  $N$ -by-4 matrix. Each row represents a separate quaternion. Each column represents a separate quaternion part.

Example: `quat = quaternion(rand(10,4))` creates a 10-by-1 quaternion array.

Data Types: `single` | `double`

### **RV — Matrix of rotation vectors**

*N*-by-3 matrix

Matrix of rotation vectors, specified as an *N*-by-3 matrix. Each row of **RV** represents the [X Y Z] elements of a rotation vector. A rotation vector is a unit vector representing the axis of rotation scaled by the angle of rotation in radians or degrees.

To use this syntax, specify the first argument as a matrix of rotation vectors and the second argument as the `'rotvec'` or `'rotvecd'`.

Example: `quat = quaternion(rand(10,3), 'rotvec')` creates a 10-by-1 quaternion array.

Data Types: `single` | `double`

### **RM — Rotation matrices**

3-by-3 matrix | 3-by-3-by-*N* array

Array of rotation matrices, specified by a 3-by-3 matrix or 3-by-3-by-*N* array. Each page of the array represents a separate rotation matrix.

Example: `quat = quaternion(rand(3), 'rotmat', 'point')`

Example: `quat = quaternion(rand(3), 'rotmat', 'frame')`

Data Types: `single` | `double`

### **PF — Type of rotation matrix**

`'point'` | `'frame'`

Type of rotation matrix, specified by `'point'` or `'frame'`.

Example: `quat = quaternion(rand(3), 'rotmat', 'point')`

Example: `quat = quaternion(rand(3), 'rotmat', 'frame')`

Data Types: `char` | `string`

### **E — Matrix of Euler angles**

*N*-by-3 matrix

Matrix of Euler angles, specified by an *N*-by-3 matrix. If using the `'euler'` syntax, specify *E* in radians. If using the `'eulerd'` syntax, specify *E* in degrees.

Example: `quat = quaternion(E, 'euler', 'YZY', 'point')`

Example: `quat = quaternion(E, 'euler', 'XYZ', 'frame')`

Data Types: `single` | `double`

### **RS — Rotation sequence**

character vector | scalar string

Rotation sequence, specified as a three-element character vector:

- `'YZY'`
- `'YXY'`
- `'ZYZ'`

- 'ZXZ'
- 'YXY'
- 'XZX'
- 'XYZ'
- 'YZX'
- 'ZXY'
- 'XZY'
- 'ZYX'
- 'YXZ'

Assume you want to determine the new coordinates of a point when its coordinate system is rotated using frame rotation. The point is defined in the original coordinate system as:

```
point = [sqrt(2)/2,sqrt(2)/2,0];
```

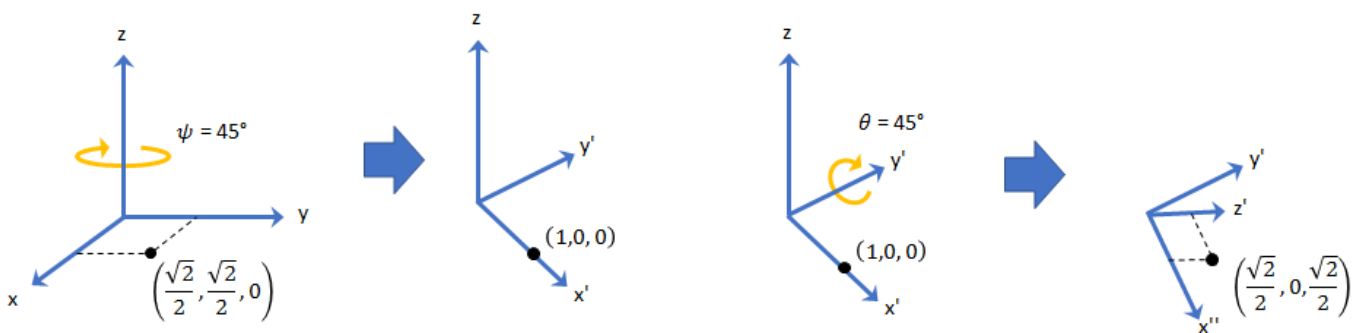
In this representation, the first column represents the x-axis, the second column represents the y-axis, and the third column represents the z-axis.

You want to rotate the point using the Euler angle representation [45,45,0]. Rotate the point using two different rotation sequences:

- If you create a quaternion rotator and specify the 'ZYX' sequence, the frame is first rotated 45° around the z-axis, then 45° around the new y-axis.

```
quatRotator = quaternion([45,45,0], 'eulerd', 'ZYX', 'frame');
newPointCoordinate = rotateframe(quatRotator,point)
```

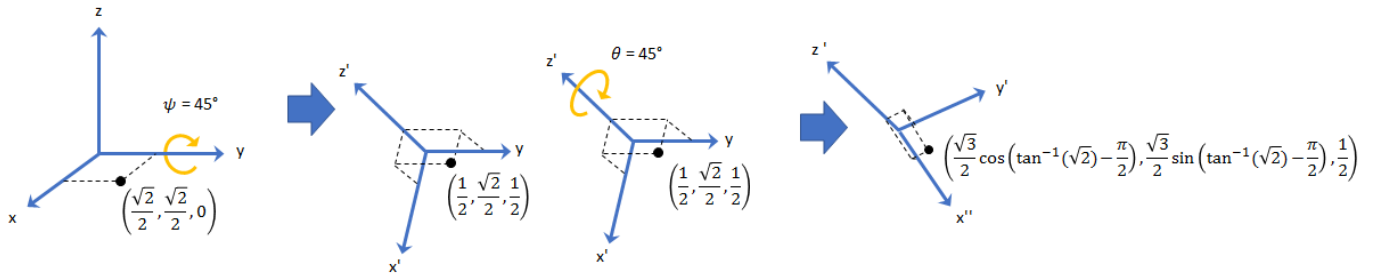
```
newPointCoordinate =
    0.7071    -0.0000    0.7071
```



- If you create a quaternion rotator and specify the 'YZX' sequence, the frame is first rotated 45° around the y-axis, then 45° around the new z-axis.

```
quatRotator = quaternion([45,45,0], 'eulerd', 'YZX', 'frame');
newPointCoordinate = rotateframe(quatRotator,point)
```

```
newPointCoordinate =
    0.8536    0.1464    0.5000
```



Data Types: char | string

### Object Functions

angvel	Angular velocity from quaternion array
classUnderlying	Class of parts within quaternion
compact	Convert quaternion array to N-by-4 matrix
conj	Complex conjugate of quaternion
'	Complex conjugate transpose of quaternion array
dist	Angular distance in radians
euler	Convert quaternion to Euler angles (radians)
eulerd	Convert quaternion to Euler angles (degrees)
exp	Exponential of quaternion array
.\,ldivide	Element-wise quaternion left division
log	Natural logarithm of quaternion array
meanrot	Quaternion mean rotation
-	Quaternion subtraction
*	Quaternion multiplication
norm	Quaternion norm
normalize	Quaternion normalization
ones	Create quaternion array with real parts set to one and imaginary parts set to zero
parts	Extract quaternion parts
.^,power	Element-wise quaternion power
prod	Product of a quaternion array
randrot	Uniformly distributed random rotations
./,rdivide	Element-wise quaternion right division
rotateframe	Quaternion frame rotation
rotatepoint	Quaternion point rotation
rotmat	Convert quaternion to rotation matrix
rotvec	Convert quaternion to rotation vector (radians)
rotvecd	Convert quaternion to rotation vector (degrees)
slerp	Spherical linear interpolation
.*,times	Element-wise quaternion multiplication
'	Transpose a quaternion array
-	Quaternion unary minus
zeros	Create quaternion array with all parts set to zero

### Examples

#### Create Empty Quaternion

```
quat = quaternion()
```

```
quat =  
    0x0 empty quaternion array
```

By default, the underlying class of the quaternion is a double.

```
classUnderlying(quat)
```

```
ans =  
'double'
```

### Create Quaternion by Specifying Individual Quaternion Parts

You can create a quaternion array by specifying the four parts as comma-separated scalars, matrices, or multidimensional arrays of the same size.

#### Define quaternion parts as scalars.

```
A = 1.1;  
B = 2.1;  
C = 3.1;  
D = 4.1;  
quatScalar = quaternion(A,B,C,D)  
  
quatScalar = quaternion  
    1.1 + 2.1i + 3.1j + 4.1k
```

#### Define quaternion parts as column vectors.

```
A = [1.1;1.2];  
B = [2.1;2.2];  
C = [3.1;3.2];  
D = [4.1;4.2];  
quatVector = quaternion(A,B,C,D)  
  
quatVector = 2x1 quaternion array  
    1.1 + 2.1i + 3.1j + 4.1k  
    1.2 + 2.2i + 3.2j + 4.2k
```

#### Define quaternion parts as matrices.

```
A = [1.1,1.3; ...  
    1.2,1.4];  
B = [2.1,2.3; ...  
    2.2,2.4];  
C = [3.1,3.3; ...  
    3.2,3.4];  
D = [4.1,4.3; ...  
    4.2,4.4];  
quatMatrix = quaternion(A,B,C,D)  
  
quatMatrix = 2x2 quaternion array  
    1.1 + 2.1i + 3.1j + 4.1k    1.3 + 2.3i + 3.3j + 4.3k  
    1.2 + 2.2i + 3.2j + 4.2k    1.4 + 2.4i + 3.4j + 4.4k
```

**Define quaternion parts as three dimensional arrays.**

```

A = randn(2,2,2);
B = zeros(2,2,2);
C = zeros(2,2,2);
D = zeros(2,2,2);
quatMultiDimArray = quaternion(A,B,C,D)

quatMultiDimArray = 2x2x2 quaternion array
quatMultiDimArray(:,:,1) =

    0.53767 +      0i +      0j +      0k   -2.2588 +      0i +      0j +      0k
    1.8339 +      0i +      0j +      0k   0.86217 +      0i +      0j +      0k

quatMultiDimArray(:,:,2) =

    0.31877 +      0i +      0j +      0k   -0.43359 +      0i +      0j +      0k
   -1.3077 +      0i +      0j +      0k   0.34262 +      0i +      0j +      0k

```

**Create Quaternion by Specifying Quaternion Parts Matrix**

You can create a scalar or column vector of quaternions by specify an  $N$ -by-4 matrix of quaternion parts, where columns correspond to the quaternion parts A, B, C, and D.

Create a column vector of random quaternions.

```

quatParts = rand(3,4)

quatParts = 3x4

    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706

quat = quaternion(quatParts)

quat = 3x1 quaternion array
    0.81472 + 0.91338i + 0.2785j + 0.96489k
    0.90579 + 0.63236i + 0.54688j + 0.15761k
    0.12699 + 0.09754i + 0.95751j + 0.97059k

```

To retrieve the `quatParts` matrix from quaternion representation, use `compact`.

```

retrievedquatParts = compact(quat)

retrievedquatParts = 3x4

    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706

```

### Create Quaternion by Specifying Rotation Vectors

You can create an  $N$ -by-1 quaternion array by specifying an  $N$ -by-3 matrix of rotation vectors in radians or degrees. Rotation vectors are compact spatial representations that have a one-to-one relationship with normalized quaternions.

#### Rotation Vectors in Radians

Create a scalar quaternion using a rotation vector and verify the resulting quaternion is normalized.

```
rotationVector = [0.3491,0.6283,0.3491];  
quat = quaternion(rotationVector,'rotvec')  
  
quat = quaternion  
      0.92124 + 0.16994i + 0.30586j + 0.16994k  
  
norm(quat)  
  
ans = 1.0000
```

You can convert from quaternions to rotation vectors in radians using the `rotvec` function. Recover the `rotationVector` from the quaternion, `quat`.

```
rotvec(quat)  
  
ans = 1×3  
      0.3491    0.6283    0.3491
```

#### Rotation Vectors in Degrees

Create a scalar quaternion using a rotation vector and verify the resulting quaternion is normalized.

```
rotationVector = [20,36,20];  
quat = quaternion(rotationVector,'rotvecd')  
  
quat = quaternion  
      0.92125 + 0.16993i + 0.30587j + 0.16993k  
  
norm(quat)  
  
ans = 1
```

You can convert from quaternions to rotation vectors in degrees using the `rotvecd` function. Recover the `rotationVector` from the quaternion, `quat`.

```
rotvecd(quat)  
  
ans = 1×3  
      20.0000    36.0000    20.0000
```



## Create Quaternion by Specifying Rotation Matrices

You can create an  $N$ -by-1 quaternion array by specifying a 3-by-3-by- $N$  array of rotation matrices. Each page of the rotation matrix array corresponds to one element of the quaternion array.

Create a scalar quaternion using a 3-by-3 rotation matrix. Specify whether the rotation matrix should be interpreted as a frame or point rotation.

```
rotationMatrix = [1 0      0; ...
                  0 sqrt(3)/2 0.5; ...
                  0 -0.5    sqrt(3)/2];
quat = quaternion(rotationMatrix, 'rotmat', 'frame')

quat = quaternion
    0.96593 + 0.25882i +      0j +      0k
```

You can convert from quaternions to rotation matrices using the `rotmat` function. Recover the `rotationMatrix` from the quaternion, `quat`.

```
rotmat(quat, 'frame')

ans = 3×3

    1.0000      0      0
         0    0.8660    0.5000
         0   -0.5000    0.8660
```

## Create Quaternion by Specifying Euler Angles

You can create an  $N$ -by-1 quaternion array by specifying an  $N$ -by-3 array of Euler angles in radians or degrees.

### Euler Angles in Radians

Use the `euler` syntax to create a scalar quaternion using a 1-by-3 vector of Euler angles in radians. Specify the rotation sequence of the Euler angles and whether the angles represent a frame or point rotation.

```
E = [pi/2,0,pi/4];
quat = quaternion(E, 'euler', 'ZYX', 'frame')

quat = quaternion
    0.65328 + 0.2706i + 0.2706j + 0.65328k
```

You can convert from quaternions to Euler angles using the `euler` function. Recover the Euler angles, `E`, from the quaternion, `quat`.

```
euler(quat, 'ZYX', 'frame')

ans = 1×3

    1.5708      0    0.7854
```

### Euler Angles in Degrees

Use the `eulerd` syntax to create a scalar quaternion using a 1-by-3 vector of Euler angles in degrees. Specify the rotation sequence of the Euler angles and whether the angles represent a frame or point rotation.

```
E = [90,0,45];
quat = quaternion(E, 'eulerd', 'ZYX', 'frame')

quat = quaternion
    0.65328 + 0.2706i + 0.2706j + 0.65328k
```

You can convert from quaternions to Euler angles in degrees using the `eulerd` function. Recover the Euler angles, `E`, from the quaternion, `quat`.

```
eulerd(quat, 'ZYX', 'frame')

ans = 1×3

    90.0000         0    45.0000
```

### Quaternion Algebra

Quaternions form a noncommutative associative algebra over the real numbers. This example illustrates the rules of quaternion algebra.

#### Addition and Subtraction

Quaternion addition and subtraction occur part-by-part, and are commutative:

```
Q1 = quaternion(1,2,3,4)

Q1 = quaternion
    1 + 2i + 3j + 4k

Q2 = quaternion(9,8,7,6)

Q2 = quaternion
    9 + 8i + 7j + 6k

Q1plusQ2 = Q1 + Q2

Q1plusQ2 = quaternion
    10 + 10i + 10j + 10k

Q2plusQ1 = Q2 + Q1

Q2plusQ1 = quaternion
    10 + 10i + 10j + 10k

Q1minusQ2 = Q1 - Q2
```

```
Q1minusQ2 = quaternion
-8 - 6i - 4j - 2k
```

```
Q2minusQ1 = Q2 - Q1
```

```
Q2minusQ1 = quaternion
8 + 6i + 4j + 2k
```

You can also perform addition and subtraction of real numbers and quaternions. The first part of a quaternion is referred to as the *real* part, while the second, third, and fourth parts are referred to as the *vector*. Addition and subtraction with real numbers affect only the real part of the quaternion.

```
Q1plusRealNumber = Q1 + 5
```

```
Q1plusRealNumber = quaternion
6 + 2i + 3j + 4k
```

```
Q1minusRealNumber = Q1 - 5
```

```
Q1minusRealNumber = quaternion
-4 + 2i + 3j + 4k
```

## Multiplication

Quaternion multiplication is determined by the products of the basis elements and the distributive law. Recall that multiplication of the basis elements,  $i$ ,  $j$ , and  $k$ , are not commutative, and therefore quaternion multiplication is not commutative.

```
Q1timesQ2 = Q1 * Q2
```

```
Q1timesQ2 = quaternion
-52 + 16i + 54j + 32k
```

```
Q2timesQ1 = Q2 * Q1
```

```
Q2timesQ1 = quaternion
-52 + 36i + 14j + 52k
```

```
isequal(Q1timesQ2,Q2timesQ1)
```

```
ans = logical
0
```

You can also multiply a quaternion by a real number. If you multiply a quaternion by a real number, each part of the quaternion is multiplied by the real number individually:

```
Q1times5 = Q1*5
```

```
Q1times5 = quaternion
5 + 10i + 15j + 20k
```

Multiplying a quaternion by a real number is commutative.

```
isequal(Q1*5,5*Q1)
```

```
ans = logical  
     1
```

### Conjugation

The complex conjugate of a quaternion is defined such that each element of the vector portion of the quaternion is negated.

Q1

```
Q1 = quaternion  
     1 + 2i + 3j + 4k
```

```
conj(Q1)
```

```
ans = quaternion  
     1 - 2i - 3j - 4k
```

Multiplication between a quaternion and its conjugate is commutative:

```
isequal(Q1*conj(Q1),conj(Q1)*Q1)
```

```
ans = logical  
     1
```

### Quaternion Array Manipulation

You can organize quaternions into vectors, matrices, and multidimensional arrays. Built-in MATLAB® functions have been enhanced to work with quaternions.

#### Concatenate

Quaternions are treated as individual objects during concatenation and follow MATLAB rules for array manipulation.

```
Q1 = quaternion(1,2,3,4);  
Q2 = quaternion(9,8,7,6);
```

```
qVector = [Q1,Q2]
```

```
qVector = 1x2 quaternion array  
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
```

```
Q3 = quaternion(-1,-2,-3,-4);  
Q4 = quaternion(-9,-8,-7,-6);
```

```
qMatrix = [qVector;Q3,Q4]
```

```
qMatrix = 2x2 quaternion array  
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
```

$$\begin{array}{cc} -1 - 2i - 3j - 4k & -9 - 8i - 7j - 6k \end{array}$$

```
qMultiDimensionalArray(:,:,1) = qMatrix;
qMultiDimensionalArray(:,:,2) = qMatrix
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =
```

$$\begin{array}{cc} 1 + 2i + 3j + 4k & 9 + 8i + 7j + 6k \\ -1 - 2i - 3j - 4k & -9 - 8i - 7j - 6k \end{array}$$

```
qMultiDimensionalArray(:,:,2) =
```

$$\begin{array}{cc} 1 + 2i + 3j + 4k & 9 + 8i + 7j + 6k \\ -1 - 2i - 3j - 4k & -9 - 8i - 7j - 6k \end{array}$$

## Indexing

To access or assign elements in a quaternion array, use indexing.

```
qLoc2 = qMultiDimensionalArray(2)
```

```
qLoc2 = quaternion
      -1 - 2i - 3j - 4k
```

Replace the quaternion at index two with a quaternion one.

```
qMultiDimensionalArray(2) = ones('quaternion')
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =
```

$$\begin{array}{cc} 1 + 2i + 3j + 4k & 9 + 8i + 7j + 6k \\ 1 + 0i + 0j + 0k & -9 - 8i - 7j - 6k \end{array}$$

```
qMultiDimensionalArray(:,:,2) =
```

$$\begin{array}{cc} 1 + 2i + 3j + 4k & 9 + 8i + 7j + 6k \\ -1 - 2i - 3j - 4k & -9 - 8i - 7j - 6k \end{array}$$

## Reshape

To reshape quaternion arrays, use the reshape function.

```
qMatReshaped = reshape(qMatrix,4,1)
```

```
qMatReshaped = 4x1 quaternion array
      1 + 2i + 3j + 4k
      -1 - 2i - 3j - 4k
      9 + 8i + 7j + 6k
      -9 - 8i - 7j - 6k
```

## Transpose

To transpose quaternion vectors and matrices, use the `transpose` function.

```
qMatTransposed = transpose(qMatrix)
```

```
qMatTransposed = 2x2 quaternion array
    1 + 2i + 3j + 4k   -1 - 2i - 3j - 4k
    9 + 8i + 7j + 6k   -9 - 8i - 7j - 6k
```

## Permute

To permute quaternion vectors, matrices, and multidimensional arrays, use the `permute` function.

```
qMultiDimensionalArray
```

```
qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =
```

```
    1 + 2i + 3j + 4k    9 + 8i + 7j + 6k
    1 + 0i + 0j + 0k   -9 - 8i - 7j - 6k
```

```
qMultiDimensionalArray(:,:,2) =
```

```
    1 + 2i + 3j + 4k    9 + 8i + 7j + 6k
   -1 - 2i - 3j - 4k   -9 - 8i - 7j - 6k
```

```
qMatPermute = permute(qMultiDimensionalArray,[3,1,2])
```

```
qMatPermute = 2x2x2 quaternion array
qMatPermute(:,:,1) =
```

```
    1 + 2i + 3j + 4k    1 + 0i + 0j + 0k
    1 + 2i + 3j + 4k   -1 - 2i - 3j - 4k
```

```
qMatPermute(:,:,2) =
```

```
    9 + 8i + 7j + 6k   -9 - 8i - 7j - 6k
    9 + 8i + 7j + 6k   -9 - 8i - 7j - 6k
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

**Introduced in R2021a**

# radarTracker

Multi-target tracker using GNN assignment

## Description

The radarTracker System object initializes, confirms, predicts, corrects, and deletes the tracks of moving objects. Inputs to the radar tracker are detection reports generated as an `objectDetection` object by radar sensors. The radar tracker accepts detections from multiple sensors and assigns them to tracks using a global nearest neighbor (GNN) criterion. Each detection is assigned to a separate track. If the detection cannot be assigned to any track, based on the `AssignmentThreshold` property, the tracker creates a new track. The tracks are returned in a structure array.

A new track starts in a *tentative* state. If enough detections are assigned to a tentative track, its status changes to *confirmed*. If the detection is a known classification (the `ObjectClassID` field of the returned track is nonzero), that track can be confirmed immediately. For details on the radar tracker properties used to confirm tracks, see “Algorithms” on page 4-121.

When a track is confirmed, the radar tracker considers that track to represent a physical object. If detections are not added to the track within a specifiable number of updates, the track is deleted.

The tracker also estimates the state vector and state vector covariance matrix for each track using a Kalman filter. These state vectors are used to predict a track's location in each frame and determine the likelihood of each detection being assigned to each track.

To track objects using a radar tracker:

- 1 Create the `radarTracker` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
tracker = radarTracker
tracker = radarTracker(Name,Value)
```

### Description

`tracker = radarTracker` creates a `radarTracker` System object with default property values.

`tracker = radarTracker(Name,Value)` sets properties for the radar tracker using one or more name-value pairs. For example, `radarTracker('FilterInitializationFcn',@initcvukf,'MaxNumTracks',100)` creates a radar tracker that uses a constant-velocity, unscented Kalman filter and maintains a maximum of 100 tracks. Enclose each property name in quotes.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### TrackerIndex — Unique tracker identifier

0 (default) | nonnegative integer

Unique tracker identifier, specified as a nonnegative integer. This property is used as the `SourceIndex` in the tracker outputs, and distinguishes tracks that come from different trackers in a multiple-tracker system.

Example: 1

### FilterInitializationFcn — Kalman filter initialization function

@initcvkf (default) | function handle | character vector | string scalar

Kalman filter initialization function, specified as a function handle or as a character vector or string scalar of the name of a valid Kalman filter initialization function.

The toolbox supplies several initialization functions that you can use to specify `FilterInitializationFcn`.

Initialization Function	Function Definition
<code>initcvekf</code>	Initialize constant-velocity extended Kalman filter.
<code>initcvkf</code>	Initialize constant-velocity linear Kalman filter.
<code>initcvukf</code>	Initialize constant-velocity unscented Kalman filter.
<code>initcaekf</code>	Initialize constant-acceleration extended Kalman filter.
<code>initcakf</code>	Initialize constant-acceleration linear Kalman filter.
<code>initcaukf</code>	Initialize constant-acceleration unscented Kalman filter.
<code>initctekf</code>	Initialize constant-turnrate extended Kalman filter.
<code>initctukf</code>	Initialize constant-turnrate unscented Kalman filter.

You can also write your own initialization function. The input to this function must be a detection report created by `objectDetection`. The output of this function must be a Kalman filter object: `trackingKF`, `trackingEKF`, or `trackingUKF`. To guide you in writing this function, you can examine the details of the supplied functions from within MATLAB. For example:

```
type initcvkf
```

Data Types: `function_handle` | `char` | `string`



**AssignmentThreshold — Detection assignment threshold**

30\*[1 Inf] (default) | positive scalar | 1-by-2 vector of positive values

Detection assignment threshold (or gating threshold), specified as a positive scalar or an 1-by-2 vector of  $[C_1, C_2]$ , where  $C_1 \leq C_2$ . If specified as a scalar, the specified value, *val*, will be expanded to  $[val, Inf]$ .

Initially, the tracker executes a *coarse* estimation for the normalized distance between all the tracks and detections. The tracker only calculates the accurate normalized distance for the combinations whose coarse normalized distance is less than  $C_2$ . Also, the tracker can only assign a detection to a track if their *accurate* normalized distance is less than  $C_1$ . See the `distance` function used with tracking filters (for example, `trackingEKF`) for an explanation of the distance calculation.

Tips:

- Increase the value of  $C_2$  if there are combinations of track and detection that should be calculated for assignment but are not. Decrease it if cost calculation takes too much time.
- Increase the value of  $C_1$  if there are detections that should be assigned to tracks but are not. Decrease it if there are detections that are assigned to tracks they should not be assigned to (too far away).

**MaxNumTracks — Maximum number of tracks**

200 (default) | positive integer

Maximum number of tracks that the tracker can maintain, specified as a positive integer.

Data Types: double

**MaxNumSensors — Maximum number of sensors**

20 (default) | positive integer

Maximum number of sensors that can be connected to the tracker, specified as a positive integer. When you specify detections as input to the radar tracker, `MaxNumSensors` must be greater than or equal to the highest `SensorIndex` value in the `detections` cell array of `objectDetection` objects used to update the radar tracker. This property determines how many sets of `ObjectAttributes` fields each output track can have.

Data Types: double

**MaxNumDetections — Maximum number of detections**

Inf (default) | positive integer

Maximum number of detections that the tracker can take as inputs, specified as a positive integer.

Data Types: single | double

**Out-of-sequence measurements handling — Out-of-sequence measurements handling**

Terminate (default) | neglect

Out-of-sequence measurements handling, specified as `Terminate` or `neglect`. Each detection has a timestamp associated with it,  $t_d$ , and the tracker block has its own timestamp,  $t_t$ , which is updated in each invocation. The tracker block considers a measurement as an OOSM if  $t_d < t_t$ .

When the parameter is specified as:

- **Terminate** — The block stops running when it encounters any out-of-sequence measurements.
- **Neglect** — The block neglects any out-of-sequence measurements and continue to run.

#### **ConfirmationThreshold — Threshold for track confirmation**

[2 3] (default) | two-element vector of non-decreasing positive integers

Threshold for track confirmation, specified as a two-element vector of non-decreasing positive integers, [M N], where M is less than or equal to N. A track is confirmed if it receives at least M detections in the last N updates.

- When setting M, take into account the probability of object detection for the sensors. The probability of detection depends on factors such as occlusion or clutter. You can reduce M when tracks fail to be confirmed or increase M when too many false detections are assigned to tracks.
- When setting N, consider the number of times you want the tracker to update before it makes a confirmation decision. For example, if a tracker updates every 0.05 seconds, and you allow 0.5 seconds to make a confirmation decision, set N = 10.

Example: [3 5]

Data Types: double

#### **DeletionThreshold — Threshold for track deletion**

[5 5] (default) | two-element vector of positive non-decreasing integers

Threshold for track deletion, specified as a two-element vector of positive non-decreasing integers [P Q], where P is less than or equal to Q. If a confirmed track is not assigned to any detection P times in the last Q tracker updates, then the track is deleted.

- Decrease Q (or increase P) if tracks should be deleted earlier.
- Increase Q (or decrease P) if tracks should be kept for a longer time before deletion.

Example: [3 5]

Data Types: single | double

#### **HasCostMatrixInput — Enable cost matrix input**

false (default) | true

Enable a cost matrix as input to the radarTracker System object, specified as false or true.

Data Types: logical

#### **HasDetectableTrackIDsInput — Enable input of detectable track IDs**

false (default) | true

Enable the input of detectable track IDs at each object update, specified as false or true. Set this property to true if you want to provide a list of detectable track IDs. This list tells the tracker of all tracks that the sensors are expected to detect and, optionally, the probability of detection for each track.

Data Types: logical

#### **StateParameters — Parameters of the track state reference frame**

struct([]) (default) | struct | struct array

Parameters of the track state reference frame, specified as a struct or a struct array. Use this property to define the track state reference frame and how to transform the track from the tracker (called source) coordinate system to the fuser coordinate system.

This property is tunable.

Data Types: `struct`

### **NumTracks — Number of tracks maintained by radar tracker**

nonnegative integer

This property is read-only.

Number of tracks maintained by the radar tracker, specified as a nonnegative integer.

Data Types: `double`

### **NumConfirmedTracks — Number of confirmed tracks**

nonnegative integer

This property is read-only.

Number of confirmed tracks, specified as a nonnegative integer. The `IsConfirmed` fields of the output track structures indicate which tracks are confirmed.

Data Types: `double`

## **Usage**

### **Syntax**

```
confirmedTracks = tracker(detections,time)
[confirmedTracks,tentativeTracks] = tracker(detections,time)
[confirmedTracks,tentativeTracks,allTracks] = tracker(detections,time)
[___] = tracker(detections,time,costMatrix)
[___] = tracker( ___,detectableTrackIDs)
```

### **Description**

`confirmedTracks = tracker(detections,time)` creates, updates, and deletes tracks in the radar tracker and returns details about the confirmed tracks. Updates are based on the specified list of `detections`, and all tracks are updated to the specified `time`. Each element in the returned `confirmedTracks` corresponds to a single track.

`[confirmedTracks,tentativeTracks] = tracker(detections,time)` also returns `tentativeTracks` containing details about the tentative tracks.

`[confirmedTracks,tentativeTracks,allTracks] = tracker(detections,time)` also returns `allTracks` containing details about all the confirmed and tentative tracks. The tracks are returned in the order by which the tracker internally maintains them. You can use this output to help you calculate the cost matrix, an optional input argument.

`[___] = tracker(detections,time,costMatrix)` specifies a cost matrix, returning any of the outputs from preceding syntaxes.

To specify a cost matrix, set the `HasCostMatrixInput` property of the tracker to `true`.

`[ ___ ] = tracker( ___, detectableTrackIDs)` also specifies a list of expected detectable tracks given by `detectableTrackIDs`. This argument can be used with any of the previous input syntaxes.

To enable this syntax, set the `HasDetectableTrackIDsInput` property to `true`.

### Input Arguments

#### **detections** – Detection list

cell array of `objectDetection` objects

Detection list, specified as a cell array of `objectDetection` objects. The `Time` property value of each `objectDetection` object must be less than or equal to the current time of update, `time`, and greater than the previous time value used to update the multi-object tracker.

#### **time** – Time of update

real scalar

Time of update, specified as a real scalar. The tracker updates all tracks to this time. Units are in seconds.

`time` must be greater than or equal to the largest `Time` property value of the `objectDetection` objects in the input `detections` list. `time` must increase in value with each update to the multi-object tracker.

Data Types: `double`

#### **costMatrix** – Cost matrix

$N_T$ -by- $N_D$  matrix

Cost matrix, specified as a real-valued  $N_T$ -by- $N_D$  matrix, where  $N_T$  is the number of existing tracks, and  $N_D$  is the number of current detections. The rows of the cost matrix correspond to the existing tracks. The columns correspond to the detections. Tracks are ordered as they appear in the list of tracks in the `allTracks` output argument of the previous update to the multi-object tracker.

In the first update to the multi-object tracker, or when the tracker has no previous tracks, assign the cost matrix a size of  $[0, N_D]$ . The cost must be calculated so that lower costs indicate a higher likelihood that the tracker assigns a detection to a track. To prevent certain detections from being assigned to certain tracks, use `Inf`.

### Dependencies

To enable specification of the cost matrix when updating tracks, set the `HasCostMatrixInput` property of the tracker to `true`

Data Types: `double`

#### **detectableTrackIDs** – Detectable track IDs

real-valued  $M$ -by-1 vector | real-valued  $M$ -by-2 matrix

Detectable track IDs, specified as a real-valued  $M$ -by-1 vector or  $M$ -by-2 matrix. Detectable tracks are tracks that the sensors expect to detect. The first column of the matrix contains a list of track IDs that the sensors report as detectable. The optional second column contains the detection probability for the track. The detection probability is either reported by a sensor or, if not reported, obtained from the `DetectionProbability` property.

Tracks whose identifiers are not included in `detectableTrackIDs` are considered as undetectable. The track deletion logic does not count the lack of detection as a 'missed detection' for track deletion purposes.

### Dependencies

To enable this input argument, set the `detectableTrackIDs` property to `true`.

Data Types: `single` | `double`

### Output Arguments

#### **confirmedTracks** — Confirmed tracks

array of `objectTrack` objects | array of structures

Confirmed tracks, returned as an array of `objectTrack` objects in MATLAB, and returned as an array of structures in code generation. In code generation, the field names of the returned structure are same with the property names of `objectTrack`.

A track is confirmed if it satisfies the confirmation threshold specified in the `ConfirmationThreshold` property. In that case, the `IsConfirmed` property of the object or field of the structure is `true`.

Data Types: `struct` | `object`

#### **tentativeTracks** — Tentative tracks

array of `objectTrack` objects | array of structures

Tentative tracks, returned as an array of `objectTrack` objects in MATLAB, and returned as an array of structures in code generation. In code generation, the field names of the returned structure are same with the property names of `objectTrack`.

A track is tentative if it does not satisfy the confirmation threshold specified in the `ConfirmationThreshold` property. In that case, the `IsConfirmed` property of the object or field of the structure is `false`.

Data Types: `struct` | `object`

#### **allTracks** — All tracks

array of `objectTrack` objects | array of structures

All tracks, returned as an array of `objectTrack` objects in MATLAB, and returned as an array of structures in code generation. In code generation, the field names of the returned structure are same with the property names of `objectTrack`. All tracks consists of confirmed and tentative tracks.

Data Types: `struct` | `object`

## Object Functions

To use an object function, specify the `System` object as the first input argument. For example, to release system resources of a `System` object named `obj`, use this syntax:

```
release(obj)
```

### Specific to radarTracker

deleteTrack	Delete existing track
getTrackFilterProperties	Obtain values of filter properties from radarTracker
initializeTrack	Initialize new track in tracker
predictTracksToTime	Predict tracks to a time stamp
setTrackFilterProperties	Sets values of track filter properties

### Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
clone	Create duplicate System object
isLocked	Determine if System object is in use
reset	Reset internal states of System object

## Examples

### Track Single Object Using Radar Tracker

Create a radarTracker System object™ using the default filter initialization function for a 3-D constant-velocity model. For this motion model, the state vector is  $[x;vx;y;vy;z;vz]$ .

```
tracker = radarTracker('ConfirmationThreshold',[4 5], ...  
    'DeletionThreshold',10);
```

Create a detection by specifying an objectDetection object. To use this detection with the radar tracker, enclose the detection in a cell array.

```
dettime = 1.0;  
det = { ...  
    objectDetection(dettime,[10; -1; 1], ...  
    'SensorIndex',1, ...  
    'ObjectAttributes',{'ExampleObject',1}) ...  
};
```

Update the radar tracker with this detection. The time at which you update the tracker must be greater than or equal to the time at which the object was detected.

```
updatetime = 1.25;  
[confirmedTracks,tentativeTracks,allTracks] = tracker(det,updatetime);
```

Create another detection of the same object and update the tracker. The tracker maintains only one track.

```
dettime = 1.5;  
det = { ...  
    objectDetection(dettime,[10.1; -1.1; 1.2], ...  
    'SensorIndex',1, ...  
    'ObjectAttributes',{'ExampleObject',1}) ...  
};  
updatetime = 1.75;  
[confirmedTracks,tentativeTracks,allTracks] = tracker(det,updatetime);
```

Determine whether the track has been verified by checking the number of confirmed tracks.

```
numConfirmed = tracker.NumConfirmedTracks
```

```
numConfirmed = 0
```

Examine the position and velocity of the tracked object. Because the track has not been confirmed, get the position and velocity from the `tentativeTracks` structure.

```
positionSelector = [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 1 0];
velocitySelector = [0 1 0 0 0 0; 0 0 0 1 0 0; 0 0 0 0 0 1];
position = getTrackPositions(tentativeTracks,positionSelector)
```

```
position = 1×3
```

```
    10.1426    -1.1426     1.2852
```

```
velocity = getTrackVelocities(tentativeTracks,velocitySelector)
```

```
velocity = 1×3
```

```
     0.1852    -0.1852     0.3705
```

### Confirm and Delete Track in Radar Tracker

Create a sequence of detections of a moving object. Track the detections using a `radarTracker System` object™. Observe how the tracks switch from tentative to confirmed and then to deleted.

Create a radar tracker using the `initcakf` filter initialization function. The tracker models 2-D constant-acceleration motion. For this motion model, the state vector is  $[x;vx;ax;y;vy;ay]$ .

```
tracker = radarTracker('FilterInitializationFcn',@initcakf, ...
    'ConfirmationThreshold',[3 4], 'DeletionThreshold',[6 6]);
```

Create a sequence of detections of a moving target using `objectDetection`. To use these detections with the `radarTracker`, enclose the detections in a cell array.

```
dt = 0.1;
pos = [10; -1];
vel = [10; 5];
for detno = 1:2
    time = (detno-1)*dt;
    det = { ...
        objectDetection(time,pos, ...
            'SensorIndex',1, ...
            'ObjectAttributes',{'ExampleObject',1}) ...
    };
    [confirmedTracks,tentativeTracks,allTracks] = tracker(det,time);
    pos = pos + vel*dt;
    meas = pos;
end
```

Verify that the track has not been confirmed yet by checking the number of confirmed tracks.

```
numConfirmed = tracker.NumConfirmedTracks
```

```
numConfirmed = 0
```

Because the track is not confirmed, get the position and velocity from the `tentativeTracks` structure.

```
positionSelector = [1 0 0 0 0 0; 0 0 0 1 0 0];
velocitySelector = [0 1 0 0 0 0; 0 0 0 0 1 0];
position = getTrackPositions(tentativeTracks,positionSelector)
```

```
position = 1×2
```

```
    10.6669    -0.6665
```

```
velocity = getTrackVelocities(tentativeTracks,velocitySelector)
```

```
velocity = 1×2
```

```
     3.3473     1.6737
```

Add more detections to confirm the track.

```
for detno = 3:5
    time = (detno-1)*dt;
    det = { ...
        objectDetection(time,pos, ...
            'SensorIndex',1, ...
            'ObjectAttributes',{'ExampleObject',1}) ...
        };
    [confirmedTracks,tentativeTracks,allTracks] = tracker(det,time);
    pos = pos + vel*dt;
    meas = pos;
end
```

Verify that the track has been confirmed, and display the position and velocity vectors for that track.

```
numConfirmed = tracker.NumConfirmedTracks
```

```
numConfirmed = 1
```

```
position = getTrackPositions(confirmedTracks,positionSelector)
```

```
position = 1×2
```

```
    13.8417     0.9208
```

```
velocity = getTrackVelocities(confirmedTracks,velocitySelector)
```

```
velocity = 1×2
```

```
     9.4670     4.7335
```

Let the tracker run but do not add new detections. The existing track is deleted.

```
for detno = 6:20
    time = (detno-1)*dt;
    det = {};
    [confirmedTracks,tentativeTracks,allTracks] = tracker(det,time);
    pos = pos + vel*dt;
```



```

    meas = pos;
end

```

Verify that the tracker has no tentative or confirmed tracks.

```
isempty(allTracks)
```

```
ans = logical
     1
```

## Algorithms

When you pass detections into a radar tracker, the System object:

- Attempts to assign the input detections to existing tracks, based on the `AssignmentThreshold` property of the multi-object tracker.
- Creates new tracks from unassigned detections.
- Updates already assigned tracks and possibly confirms them, based on the `ConfirmationThreshold` property of the tracker.
- Deletes tracks that have no assigned detections, based on the `DeletionThreshold` property of the tracker.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).
- All the detections used with the tracker must have fields with the same sizes and types.
- The `objectDetection` structure must have an `ObjectAttributes` field. The value of this field can be an empty structure, a structure, or a cell containing a structure. The structure for all detections must have the same fields and the values in these fields must always have the same size and type. The form of the structure cannot change during simulation.
- The first update to the tracker must contain at least one detection.
- When the filter initialization function specified in the tracker returns a `trackingEKF` or `trackingUKF` object and when the `MaxNumDetections` property is specified as a finite integer, the tracker supports non-dynamic memory allocation code generation.

## See Also

### Functions

`getTrackPositions` | `getTrackVelocities`

### Objects

`objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

**Introduced in R2021a**

# deleteTrack

Delete existing track

## Syntax

```
deleted = deleteTrack(tracker,trackID)
```

## Description

`deleted = deleteTrack(tracker,trackID)` deletes the track specified by `trackID` in the `tracker`.

## Examples

### Delete track in radarTracker

Create a track using detections in a `radarTracker`.

```
tracker = radarTracker
tracker =
  radarTracker with properties:
      TrackerIndex: 0
      FilterInitializationFcn: 'initcvekf'
      AssignmentThreshold: [30 Inf]
      MaxNumTracks: 100
      MaxNumDetections: Inf
      MaxNumSensors: 20
      OOSMHandling: 'Terminate'
      ConfirmationThreshold: [2 3]
      DeletionThreshold: [5 5]
      HasCostMatrixInput: false
      HasDetectableTrackIDsInput: false
      StateParameters: [1x1 struct]
      NumTracks: 0
      NumConfirmedTracks: 0

detection1 = objectDetection(0,[1;1;1]);
detection2 = objectDetection(1,[1.1;1.2;1.1]);
tracker(detection1,0);
tracker(detection2,1)

ans =
  objectTrack with properties:
```

```
        TrackID: 1
        BranchID: 0
        SourceIndex: 0
        UpdateTime: 1
        Age: 2
        State: [6x1 double]
        StateCovariance: [6x6 double]
        StateParameters: [1x1 struct]
        ObjectClassID: 0
        TrackLogic: 'History'
        TrackLogicState: [1 1 0 0 0]
        IsConfirmed: 1
        IsCoasted: 0
        IsSelfReported: 1
        ObjectAttributes: [1x1 struct]
```

Delete the first track.

```
deleted1 = deleteTrack(tracker,1)

deleted1 = logical
         1
```

Uncomment the following to delete a nonexistent track. A warning will be issued.

```
% deleted2 = deleteTrack(tracker,2)
```

## Input Arguments

### **tracker** — radar tracker

radarTracker object

Radar tracker, specified as a radarTracker object.

### **trackID** — Track identifier

positive integer

Track identifier, specified as a positive integer.

Example: 21

## Output Arguments

### **deleted** — Indicate if track was successfully deleted

1 | 0

Indicate if the track was successfully deleted or not, returned as 1 or 0. If the track specified by the trackID input existed and was successfully deleted, it returns as 1. If the track did not exist, a warning is issued and it returns as 0.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

radarTracker | initializeTrack

### Introduced in R2021a

## getTrackFilterProperties

Obtain values of filter properties from radarTracker

### Syntax

```
values = getTrackFilterProperties(tracker, trackID, property)
values = getTrackFilterProperties(tracker, trackID, property1, ..., propertyN)
```

### Description

`values = getTrackFilterProperties(tracker, trackID, property)` returns the tracking filter property values for a specific track within a multi-object tracker. `trackID` is the ID of that specific track.

`values = getTrackFilterProperties(tracker, trackID, property1, ..., propertyN)` returns multiple property values. You can specify the properties in any order.

### Examples

#### Display and Set Tracking Filter Properties in Radar Tracker

Create a `radarTracker` System object™ using a constant-acceleration, linear Kalman filter for all tracks.

```
tracker = radarTracker('FilterInitializationFcn', @initcakf, ...
    'ConfirmationThreshold', [4 5], 'DeletionThreshold', [9 9]);
```

Create two detections and generate tracks for these detections.

```
detection1 = objectDetection(1.0, [10; 10]);
detection2 = objectDetection(1.0, [1000; 1000]);
[~, tracks] = tracker([detection1 detection2], 1.1)
```

`tracks=2x1 object`  
2x1 `objectTrack` array with properties:

```
TrackID
BranchID
SourceIndex
UpdateTime
Age
State
StateCovariance
StateParameters
ObjectClassID
TrackLogic
TrackLogicState
IsConfirmed
IsCoasted
IsSelfReported
```

## ObjectAttributes

Get filter property values for the first track. Display the process noise values.

```
values = getTrackFilterProperties(tracker,1, 'MeasurementNoise', 'ProcessNoise', 'MotionModel');
values{2}

ans = 2×2

     1     0
     0     1
```

Set new values for this property by doubling the process noise for the first track. Display the updated process noise values.

```
setTrackFilterProperties(tracker,1, 'ProcessNoise', 2*values{2});
values = getTrackFilterProperties(tracker,1, 'ProcessNoise');
values{1}

ans = 2×2

     2     0
     0     2
```

## Input Arguments

### **tracker** — radar tracker

radarTracker object

Radar tracker, specified as a radarTracker object.

### **trackID** — Track ID

positive integer

Track ID, specified as a positive integer. `trackID` must be a valid track in `tracker`.

### **property** — Tracking filter property

character vector | string scalar

Tracking filter property to return values for, specified as a character vector or string scalar. `property` must be a valid property of the tracking filter used by `tracker`. Valid tracking filters are `trackingKF`, `trackingEKF`, and `trackingUKF`.

You can specify additional properties in any order.

Example: 'MeasurementNoise', 'ProcessNoise'

Data Types: char | string

## Output Arguments

### **values** — Tracking filter property values

cell array

Tracking filter property values, returned as a cell array. Each element in the cell array corresponds to the values of a specified property. `getTrackFilterProperties` returns the values in the same order in which you specified the corresponding properties.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Objects**

`radarTracker` | `trackingKF` | `trackingEKF` | `trackingUKF`

### **Functions**

`setTrackFilterProperties`

**Introduced in R2021a**



# initializeTrack

Initialize new track in tracker

## Syntax

```
trackID = initializeTrack(tracker, track)
trackID = initializeTrack(tracker, track, filter)
```

## Description

`trackID = initializeTrack(tracker, track)` initializes a new track in the tracker. The tracker must be updated at least once before initializing a track. If the track is initialized successfully, the tracker assigns the output `trackID` to the track, sets the `UpdateTime` of the track equal to the last step time in the tracker, and synchronizes the data in the input `track` to the initialized track.

A warning is issued if the tracker already maintains the maximum number of tracks specified by its `MaxNumTracks` property. In this case, the `trackID` is returned as `0`, which indicates a failure to initialize the track.

`trackID = initializeTrack(tracker, track, filter)` initializes a new track in the tracker, using a specified tracking filter, `filter`.

## Examples

### Initialize Track in Radar Tracker

Create a radar tracker and update the tracker with detections at  $t = 0$  and  $t = 1$  second.

```
tracker = radarTracker
```

```
tracker =
```

```
    radarTracker with properties:
```

```

        TrackerIndex: 0
    FilterInitializationFcn: 'initcvekf'
    AssignmentThreshold: [30 Inf]
        MaxNumTracks: 100
    MaxNumDetections: Inf
    MaxNumSensors: 20

        OOSMHandling: 'Terminate'

    ConfirmationThreshold: [2 3]
    DeletionThreshold: [5 5]

        HasCostMatrixInput: false
    HasDetectableTrackIDsInput: false
        StateParameters: [1x1 struct]

        NumTracks: 0
```

```
NumConfirmedTracks: 0
```

```
detection1 = objectDetection(0,[1;1;1]);  
detection2 = objectDetection(1,[1.1;1.2;1.1]);  
tracker(detection1,0);  
currentTrack = tracker(detection2,1);
```

As seen from the NumTracks property, the tracker now maintains one track.

```
tracker
```

```
tracker =  
  radarTracker with properties:  
  
    TrackerIndex: 0  
    FilterInitializationFcn: 'initcvekf'  
    AssignmentThreshold: [30 Inf]  
    MaxNumTracks: 100  
    MaxNumDetections: Inf  
    MaxNumSensors: 20  
  
    OOSMHandling: 'Terminate'  
  
    ConfirmationThreshold: [2 3]  
    DeletionThreshold: [5 5]  
  
    HasCostMatrixInput: false  
    HasDetectableTrackIDsInput: false  
    StateParameters: [1x1 struct]  
  
    NumTracks: 1  
    NumConfirmedTracks: 1
```

Create a new track using the objectTrack object.

```
newTrack = objectTrack()
```

```
newTrack =  
  objectTrack with properties:  
  
    TrackID: 1  
    BranchID: 0  
    SourceIndex: 1  
    UpdateTime: 0  
    Age: 1  
    State: [6x1 double]  
    StateCovariance: [6x6 double]  
    StateParameters: [1x1 struct]  
    ObjectClassID: 0  
    TrackLogic: 'History'  
    TrackLogicState: 1  
    IsConfirmed: 1  
    IsCoasted: 0  
    IsSelfReported: 1  
    ObjectAttributes: [1x1 struct]
```

Initialize a track in the GNN tracker object using the newly created track.

```
trackID = initializeTrack(tracker,newTrack)
```

```
trackID = uint32
         2
```

As seen from the NumTracks property, the tracker now maintains two tracks.

```
tracker
```

```
tracker =
  radarTracker with properties:

        TrackerIndex: 0
  FilterInitializationFcn: 'initcvekf'
  AssignmentThreshold: [30 Inf]
        MaxNumTracks: 100
  MaxNumDetections: Inf
        MaxNumSensors: 20

        OOSMHandling: 'Terminate'

  ConfirmationThreshold: [2 3]
  DeletionThreshold: [5 5]

        HasCostMatrixInput: false
  HasDetectableTrackIDsInput: false
        StateParameters: [1x1 struct]

        NumTracks: 2
  NumConfirmedTracks: 2
```

## Input Arguments

### **tracker** — radar tracker

radarTracker object

Radar tracker, specified as a radarTracker object.

### **track** — New track to be initialized

objectTrack object | structure

New track to be initialized, specified as an objectTrack object or a structure. If specified as a structure, the name, variable type, and data size of the fields of the structure must be the same as the name, variable type, and data size of the corresponding properties of the objectTrack object.

Data Types: struct | object

### **filter** — Filter object

trackingKF | trackingEKF | trackingUKF

Filter object, specified as a trackingKF, trackingEKF, or trackingUKF object.

## Output Arguments

### **trackID** — Track identifier

nonnegative integer

Track identifier, returned as a nonnegative integer. `trackID` is returned as 0 if the `track` is not initialized successfully.

Example: 2

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`radarTracker` | `deleteTrack`

**Introduced in R2021a**

# predictTracksToTime

Predict tracks to a time stamp

## Syntax

```
predictedtracks = predictTracksToTime(tracker, trackID, time)
predictedtracks = predictTracksToTime(tracker, category, time)
predictedtracks = predictTracksToTime(tracker, category,
time, 'WithCovariance', tf)
```

## Description

`predictedtracks = predictTracksToTime(tracker, trackID, time)` returns the predicted tracks, `predictedtracks`, of the tracker, at the specified time, `time`. The tracker or fuser must be updated at least once before calling this object function. Use `isLocked(tracker)` to test whether the tracker or fuser has been updated.

---

**Note** This function only outputs the predicted tracks and does not update the internal track states of the tracker.

---

`predictedtracks = predictTracksToTime(tracker, category, time)` returns all predicted tracks for a specified category, `category`, of tracked objects.

`predictedtracks = predictTracksToTime(tracker, category, time, 'WithCovariance', tf)` also allows you to specify whether to predict the state covariance of each track or not by setting the `tf` flag to `true` or `false`. Predicting the covariance slows down the prediction process and increases the computation cost, but it provides the predicted track state covariance in addition to the predicted state. The default is `false`.

## Examples

### Predict Track State in radarTracker

Create a track from a detection at time  $t = 0$  second.

```
tracker = radarTracker;
detection = objectDetection(0, [0;0;0]);
tracker(detection, 0);
```

Predict the track to  $t = 1$  second.

```
predictedtracks = predictTracksToTime(tracker, 'all', 1)
```

```
predictedtracks =
  objectTrack with properties:
```

```
    TrackID: 1
   BranchID: 0
```

```
SourceIndex: 0
UpdateTime: 1
Age: 1
State: [6x1 double]
StateCovariance: [6x6 double]
StateParameters: [1x1 struct]
ObjectClassID: 0
TrackLogic: 'History'
TrackLogicState: [1 0 0 0 0]
IsConfirmed: 0
IsCoasted: 0
IsSelfReported: 1
ObjectAttributes: [1x1 struct]
```

## Input Arguments

### **tracker** — radar tracker

radarTracker object

Radar tracker, specified as a radarTracker object.

### **trackID** — Track identifier

positive integer

Track identifier, specified as a positive integer. Only the track specified by the `trackID` is predicted in the tracker.

Example: 15

Data Types: single | double

### **time** — Prediction time

scalar

Prediction time, specified as a scalar. The states of tracks are predicted to this time. The time must be greater than the time input to the tracker in the previous track update. Units are in seconds.

Example: 1.0

Data Types: single | double

### **category** — Track categories

'all' | 'confirmed' | 'tentative'

Track categories, specified as 'all', 'confirmed', or 'tentative'. You can choose to predict all tracks, only confirmed tracks, or only tentative tracks.

Data Types: char

## Output Arguments

### **predictedtracks** — List of predicted track or branch states

array of objectTrack objects | array of structures

List of tracks or branches, returned as:

- An array of `objectTrack` objects in the MATLAB interpreted mode.
- An array of structures in the code generation mode. The field names of the structures are the same as the names of properties in `objectTrack`.

Data Types: `struct` | `object`

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`radarTracker`

**Introduced in R2021a**

## setTrackFilterProperties

Sets values of track filter properties

### Syntax

```
setTrackFilterProperties(tracker, trackID, property, value)
setTrackFilterProperties(tracker,
trackID, property1, value1, ..., propertyN, valueN)
```

### Description

`setTrackFilterProperties(tracker, trackID, property, value)` sets the specified tracking filter property to the indicated value for a specific track within the radar tracker. `trackID` is the ID of that specific track.

`setTrackFilterProperties(tracker, trackID, property1, value1, ..., propertyN, valueN)` sets multiple property values. You can specify the property-value pairs in any order.

### Examples

#### Display and Set Tracking Filter Properties in Radar Tracker

Create a `radarTracker` System object™ using a constant-acceleration, linear Kalman filter for all tracks.

```
tracker = radarTracker('FilterInitializationFcn', @initcakf, ...
    'ConfirmationThreshold', [4 5], 'DeletionThreshold', [9 9]);
```

Create two detections and generate tracks for these detections.

```
detection1 = objectDetection(1.0, [10; 10]);
detection2 = objectDetection(1.0, [1000; 1000]);
[~, tracks] = tracker([detection1 detection2], 1.1)
```

`tracks=2x1 object`  
2x1 `objectTrack` array with properties:

```
TrackID
BranchID
SourceIndex
UpdateTime
Age
State
StateCovariance
StateParameters
ObjectClassID
TrackLogic
TrackLogicState
IsConfirmed
IsCoasted
```



```
IsSelfReported
ObjectAttributes
```

Get filter property values for the first track. Display the process noise values.

```
values = getTrackFilterProperties(tracker,1,'MeasurementNoise','ProcessNoise','MotionModel');
values{2}
```

```
ans = 2×2
```

```
    1    0
    0    1
```

Set new values for this property by doubling the process noise for the first track. Display the updated process noise values.

```
setTrackFilterProperties(tracker,1,'ProcessNoise',2*values{2});
values = getTrackFilterProperties(tracker,1,'ProcessNoise');
values{1}
```

```
ans = 2×2
```

```
    2    0
    0    2
```

## Input Arguments

### **tracker** — radar tracker

radarTracker object

Radar tracker, specified as a radarTracker object.

### **trackID** — Track ID

positive integer

Track ID, specified as a positive integer. trackID must be a valid track in tracker.

### **property** — Tracking filter property

character vector | string scalar

Tracking filter property to set values for, specified as a character vector or string scalar. property must be a valid property of the tracking filter used by tracker. Valid tracking filters are trackingKF, trackingEKF, and trackingUKF.

You can specify additional property-value pairs in any order.

Example: 'MeasurementNoise',eye(2,2),'MotionModel','2D Constant Acceleration'

Data Types: char | string

### **value** — Value to set tracking filter property to

valid MATLAB expression

Value to set the corresponding tracking filter property to, specified as a MATLAB expression. `value` must be a valid value of the corresponding property.

You can specify additional property-value pairs in any order.

Example: `'MeasurementNoise',eye(2,2),'MotionModel','2D Constant Acceleration'`

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

#### **Objects**

`trackingKF` | `trackingEKF` | `trackingUKF`

#### **Functions**

`getTrackFilterProperties`

**Introduced in R2021a**

# clusterDBSCAN

Density-based algorithm for clustering data

## Description

`clusterDBSCAN` clusters data points belonging to a  $P$ -dimensional feature space using the density-based spatial clustering of applications with noise (DBSCAN) algorithm. The clustering algorithm assigns points that are close to each other in feature space to a single cluster. For example, a radar system can return multiple detections of an extended target that are closely spaced in range, angle, and Doppler. `clusterDBSCAN` assigns these detections to a single detection.

- The DBSCAN algorithm assumes that clusters are dense regions in data space separated by regions of lower density and that all dense regions have similar densities.
- To measure density at a point, the algorithm counts the number of data points in a neighborhood of the point. A neighborhood is a  $P$ -dimensional ellipse (hyperellipse) in the feature space. The radii of the ellipse are defined by the  $P$ -vector  $\epsilon$ .  $\epsilon$  can be a scalar, in which case, the hyperellipse becomes a hypersphere. Distances between points in feature space are calculated using the Euclidean distance metric. The neighborhood is called an  $\epsilon$ -neighborhood. The value of  $\epsilon$  is defined by the `Epsilon` property. `Epsilon` can either be a scalar or  $P$ -vector:
  - A vector is used when different dimensions in feature space have different units.
  - A scalar applies the same value to all dimensions.
- Clustering starts by finding all *core* points. If a point has a sufficient number of points in its  $\epsilon$ -neighborhood, the point is called a core point. The minimum number of points required for a point to become a core point is set by the `MinNumPoints` property.
- The remaining points in the  $\epsilon$ -neighborhood of a core point can be core points themselves. If not, they are *border* points. All points in the  $\epsilon$ -neighborhood are called *directly density reachable* from the core point.
- If the  $\epsilon$ -neighborhood of a core point contains other core points, the points in the  $\epsilon$ -neighborhoods of all the core points merge together to form a union of  $\epsilon$ -neighborhoods. This process continues until no more core points can be added.
  - All points in the union of  $\epsilon$ -neighborhoods are *density reachable* from the first core point. In fact, all points in the union are density reachable from all core points in the union.
  - All points in the union of  $\epsilon$ -neighborhoods are also termed *density connected* even though border points are not necessarily *reachable* from each other. A *cluster* is a maximal set of density-connected points and can have an arbitrary shape.
- Points that are not core or border points are *noise* points. They do not belong to any cluster.
- The `clusterDBSCAN` object can estimate  $\epsilon$  using a  $k$ -nearest neighbor search, or you can specify values. To let the object estimate  $\epsilon$ , set the `EpsilonSource` property to `'Auto'`.
- The `clusterDBSCAN` object can disambiguate data containing ambiguities. Range and Doppler are examples of possibly ambiguous data. Set `Enabledisambiguation` property to `true` to disambiguate data.

To cluster detections:

- 1 Create the `clusterDBSCAN` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
clusterer = clusterDBSCAN
clusterer = clusterDBSCAN(Name,Value)
```

### Description

`clusterer = clusterDBSCAN` creates a `clusterDBSCAN` object, `clusterer`, object with default property values.

“Effect of Epsilon on Clustering” on page 4-147

`clusterer = clusterDBSCAN(Name,Value)` creates a `clusterDBSCAN` object, `clusterer`, with each specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as `(Name1,Value1,...,NameN,ValueN)`. Any unspecified properties take default values. For example,

```
clusterer = clusterDBSCAN('MinNumPoints',3,'Epsilon',2, ...
'EnableDisambiguation',true,'AmbiguousDimension',[1 2]);
```

creates a `clusterer` with the `EnableDisambiguation` property set to `true` and the `AmbiguousDimension` set to `[1,2]`.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

#### EpsilonSource — Source of epsilon

'Property' (default) | 'Auto'

Source of epsilon values defining an  $\epsilon$ -neighborhood, specified as 'Property' or 'Auto'.

- When you set the `EpsilonSource` property to 'Property',  $\epsilon$  is obtained from the `Epsilon` property.
- When you set the `EpsilonSource` property to 'Auto',  $\epsilon$  is estimated automatically using a  $k$ -nearest neighbor ( $k$ -NN) search over a range of  $k$  values from  $k_{\min}$  to  $k_{\max}$ .

$$k_{\min} = \text{MinNumPoints} - 1$$

$$k_{\max} = \text{MaxNumPoints} - 1$$

The subtraction of one is needed because the number of neighbors of a point does not include the point itself, whereas `MinNumPoints` and `MaxNumPoints` refer to the total number of points in a neighborhood.

Data Types: `char` | `string`

### **Epsilon — Radius for neighborhood search**

10.0 (default) | positive scalar | positive, real-valued 1-by- $P$  row vector

Radius for a neighborhood search, specified as a positive scalar or positive, real-valued 1-by- $P$  row vector.  $P$  is the number of features in the input data,  $X$ .

`Epsilon` defines the radii of an ellipse around any point to create an  $\varepsilon$ -neighborhood. When `Epsilon` is a scalar, the same radius applies to all feature dimensions. You can apply different epsilon values for different features by specifying a positive, real-valued 1-by- $P$  row vector. A row vector creates a multidimensional ellipse (hyperellipse) search area, useful when the data features have different physical meanings, such as range and Doppler. See “Estimate Epsilon” on page 4-154 for more information about this property.

You can use the `clusterDBSCAN.estimateEpsilon` or `clusterDBSCAN.discoverClusters` object functions to help estimate a scalar value for epsilon.

Example: [11 21.0]

**Tunable:** Yes

### **Dependencies**

To enable this property, set the `EpsilonSource` property to 'Property'.

Data Types: `double`

### **MinNumPoints — Minimum number of points required for cluster**

3 (default) | positive integer

Minimum number of points in an  $\varepsilon$ -neighborhood of a point for that point to become a core point, specified as a positive integer. See “Choosing the Minimum Number of Points” on page 4-157 for more information. When the object automatically estimates epsilon using a  $k$ -NN search, the starting value of  $k$  ( $k_{\min}$ ) is `MinNumPoints` - 1.

Example: 5

Data Types: `double`

### **MaxNumPoints — Set end of $k$ -NN search range**

10 (default) | positive integer

Set end of  $k$ -NN search range, specified as a positive integer. When the object automatically estimates epsilon using a  $k$ -NN search, the ending value of  $k$  ( $k_{\max}$ ) is `MaxNumPoints` - 1.

Example: 13

### **Dependencies**

To enable this property, set the `EpsilonSource` property to 'Auto'.

Data Types: `double`

**EpsilonHistoryLength — Length of cluster threshold epsilon history**

10 (default) | positive integer

Length of the stored epsilon history, specified as a positive integer. When set to one, the history is memory-less, meaning that each epsilon estimate is immediately used and no moving-average smoothing occurs. When greater than one, epsilon is averaged over the history length specified.

Example: 5

**Dependencies**

To enable this property, set the `EpsilonSource` property to 'Auto'.

Data Types: double

**EnableDisambiguation — Enable disambiguation of dimensions**

false (default) | true

Switch to enable disambiguation of dimensions, specified as `false` or `true`. When `true`, clustering can occur across boundaries defined by the input `amblims` at execution. Use the `AmbiguousDimensions` property to specify the column indices of `X` in which ambiguities can occur. You can disambiguate up to two dimensions. Turning on disambiguation is not recommended for large data sets.

Data Types: logical

**AmbiguousDimension — Indices of ambiguous dimensions**

1 (default) | positive integer | 1-by-2 vector of positive integers

Indices of ambiguous dimensions, specified as a positive integer or 1-by-2 vector of positive integers. This property specifies the column of `X` in which to apply disambiguation. A positive integer indicates a single ambiguous dimension in the input data matrix `X`. A 1-by-2 row vector specifies two ambiguous dimensions. The size and order of `AmbiguousDimension` must be consistent with the object input `amblims`.

Example: [3 4]

**Dependencies**

To enable this property, set the `EnableDisambiguation` property to `true`.

Data Types: double

**Usage****Syntax**

```
idx = clusterer(X)
[idx,clusterids] = clusterer(X)
[___] = clusterer(X,amblims)
[___] = clusterer(X,update)
[___] = clusterer(X,amblims,update)
```

**Description**

`idx = clusterer(X)` clusters the points in the input data, `X`. `idx` contains a list of IDs identifying the cluster to which each row of `X` belongs. Noise points are assigned as '-1'.

`[idx,clusterids] = clusterer(X)` also returns an alternate set of cluster IDs, `clusterids`, for use in the `phased.RangeEstimator` and `phased.DopplerEstimator` objects. `clusterids` assigns a unique ID to each noise point.

`[ ___ ] = clusterer(X,amblims)` also specifies the minimum and maximum ambiguity limits, `amblims`, to apply to the data.

To enable this syntax, set the `EnableDisambiguation` property to `true`.

`[ ___ ] = clusterer(X,update)` automatically estimates epsilon from the input data matrix, `X`, when `update` is set to `true`. The estimation uses a  $k$ -NN search to create a set of search curves. For more information, see “Estimate Epsilon” on page 4-154. The estimate is an average of the  $L$  most recent Epsilon values where  $L$  is specified in `EpsilonHistoryLength`

To enable this syntax, set the `EpsilonSource` property to `'Auto'`, optionally set the `MaxNumPoints` property, and also optionally set the `EpsilonHistoryLength` property.

`[ ___ ] = clusterer(X,amblims,update)` sets ambiguity limits and estimates epsilon when `update` is set to `true`. To enable this syntax, set `EnableDisambiguation` to `true` and set `EpsilonSource` to `'Auto'`.

## Input Arguments

### **X — Input feature data**

real-valued  $N$ -by- $P$  matrix

Input feature data, specified as a real-valued  $N$ -by- $P$  matrix. The  $N$  rows correspond to feature points in a  $P$ -dimensional feature space. The  $P$  columns contain the values of the features over which clustering takes place. The DBSCAN algorithm can cluster any type of data with appropriate `MinNumPoints` and `Epsilon` settings. For example, a two-column input can contain the  $xy$  Cartesian coordinates, or range and Doppler.

Data Types: `double`

### **amblims — Ambiguity limits**

1-by-2 real-valued vector (default) | 2-by-2 real-valued matrix

Ambiguity limits, specified as a real-valued 1-by-2 vector or real-valued 2-by-2 matrix. For a single ambiguity dimension, specify the limits as a 1-by-2 vector `[MinAmbiguityLimitDimension1,MaxAmbiguityLimitDimension1]`. For two ambiguity dimensions, specify the limits as a 2-by-2 matrix `[MinAmbiguityLimitDimension1, MaxAmbiguityLimitDimension1; MinAmbiguityLimitDimension2,MaxAmbiguityLimitDimension2]`. Ambiguity limits allow clustering across boundaries to ensure that ambiguous detections are appropriately clustered.

The ambiguous columns of `X` are defined in the `AmbiguousDimension` property. `amblims` defines the minimum and maximum ambiguity limits in the same units as the data in the `AmbiguousDimension` columns of `X`.

Example: `[0 20; -40 40]`

## Dependencies

To enable this argument, set `EnableDisambiguation` to `true` and set the `AmbiguousDimension` property.

Data Types: `double`

**update — Enable automatic update of epsilon**`false (default) | true`

Enable automatic update of the epsilon estimate, specified as `false` or `true`.

- When `true`, the epsilon threshold is first estimated as the average of the knees of  $k$ -NN search curves. The estimate is then added to a buffer whose length  $L$  is set in the `EpsilonHistoryLength` property. The final epsilon that is used is calculated as the average of the  $L$ -length epsilon history buffer. If `EpsilonHistoryLength` is set to 1, the estimate is memory-less. Memory-less means that each epsilon estimate is immediately used and no moving-average smoothing occurs.
- When `false`, a previous epsilon estimate is used. Estimating epsilon is computationally intensive and not recommended for large data sets.

**Dependencies**

To enable this argument, set the `EpsilonSource` property to 'Auto' and specify the `MaxNumPoints` property.

Data Types: `double`

**Output Arguments****idx — Cluster indices**

$N$ -by-1 integer-valued column vector

Cluster indices, returned as an integer-valued  $N$ -by-1 column vector. `idx` represents the clustering results of the DBSCAN algorithm. Positive `idx` values correspond to clusters that satisfy the DBSCAN clustering criteria. A value of '-1' indicates a DBSCAN noise point.

Data Types: `double`

**clusterids — Alternative cluster IDs**

1-by- $N$  integer-valued row vector

Alternative cluster IDs, returned as a 1-by- $N$  row vector of positive integers. Each value is a unique identifier indicating a hypothetical target cluster. This argument contains unique positive cluster IDs for all points including noise. In contrast, the `idx` output argument labels noise points with '-1'. Use `clusterids` as the input to Phased Array System Toolbox objects such as `phased.RangeEstimator` and `phased.DopplerEstimator`.

Data Types: `double`

**Object Functions**

To use an object function, specify the `System` object as the first input argument. For example, to release system resources of a `System` object named `obj`, use this syntax:

```
release(obj)
```

**Specific to clusterDBSCAN**

<code>clusterDBSCAN.discoverClusters</code>	Find cluster hierarchy in data
<code>clusterDBSCAN.estimateEpsilon</code>	Estimate neighborhood clustering threshold
<code>clusterDBSCAN.plot</code>	Plot clusters



## Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

## Examples

### Cluster Detections in Range and Doppler

Create detections of extended objects with measurements in range and Doppler. Assume the maximum unambiguous range is 20 m and the unambiguous Doppler span extends from  $-30$  Hz to 30 Hz. Data for this example is contained in the `dataClusterDBSCAN.mat` file. The first column of the data matrix represents range, and the second column represents Doppler.

The input data contains the following extended targets and false alarms:

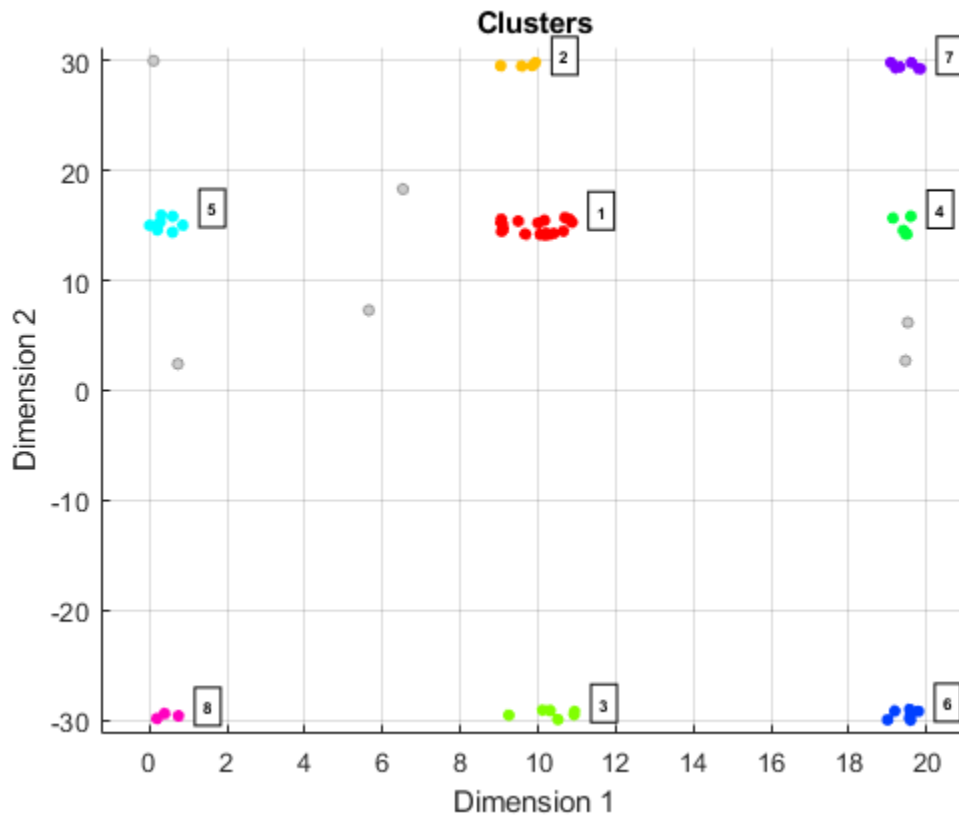
- an unambiguous target located at (10, 15)
- an ambiguous target in Doppler located at (10,  $-30$ )
- an ambiguous target in range located at (20, 15)
- an ambiguous target in range and Doppler located at (20, 30)
- 5 false alarms

Create a `clusterDBSCAN` object and specify that disambiguation is not performed by setting `EnableDisambiguation` to `false`. Solve for the cluster indices.

```
load('dataClusterDBSCAN.mat');
cluster1 = clusterDBSCAN('MinNumPoints',3,'Epsilon',2, ...
    'EnableDisambiguation',false);
idx = cluster1(x);
```

Use the `clusterDBSCAN` plot object function to display the clusters.

```
plot(cluster1,x,idx)
```



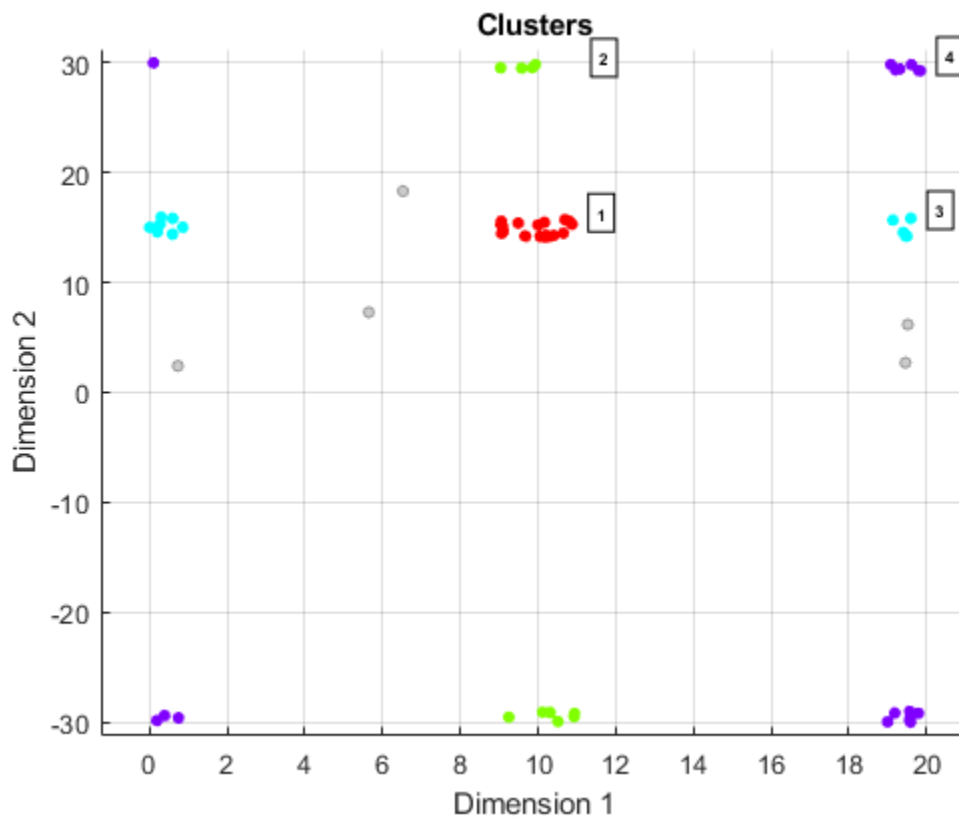
The plot indicates that there are eight apparent clusters and six noise points. The 'Dimension 1' label corresponds to range and the 'Dimension 2' label corresponds to Doppler.

Next, create another `clusterDBSCAN` object and set `EnableDisambiguation` to `true` to specify that clustering is performed across the range and Doppler ambiguity boundaries.

```
cluster2 = clusterDBSCAN('MinNumPoints',3,'Epsilon',2, ...
    'EnableDisambiguation',true,'AmbiguousDimension',[1 2]);
```

Perform the clustering using ambiguity limits and then plot the clustering results. The DBSCAN clustering results correctly show four clusters and five noise points. For example, the points at ranges close to zero are clustered with points near 20 m because the maximum unambiguous range is 20 m.

```
amblims = [0 maxRange; minDoppler maxDoppler];
idx = cluster2(x,amblims);
plot(cluster2,x,idx)
```

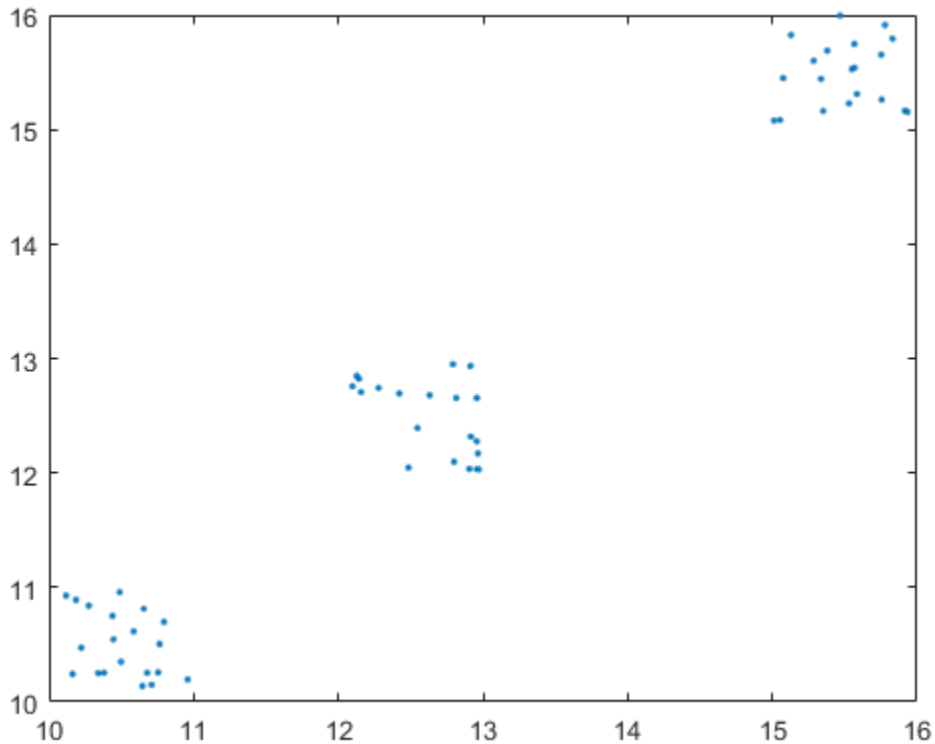


### Effect of Epsilon on Clustering

Cluster two-dimensional Cartesian position data using `clusterDBSCAN`. To illustrate how the choice of epsilon affects clustering, compare the results of clustering with Epsilon set to 1 and Epsilon set to 3.

Create random target position data in  $xy$  Cartesian coordinates.

```
x = [rand(20,2)+12; rand(20,2)+10; rand(20,2)+15];
plot(x(:,1),x(:,2),'.')
```



Create a `clusterDBSCAN` object with the `Epsilon` property set to 1 and the `MinNumPoints` property set to 3.

```
clusterer = clusterDBSCAN('Epsilon',1,'MinNumPoints',3);
```

Cluster the data when `Epsilon` equals 1.

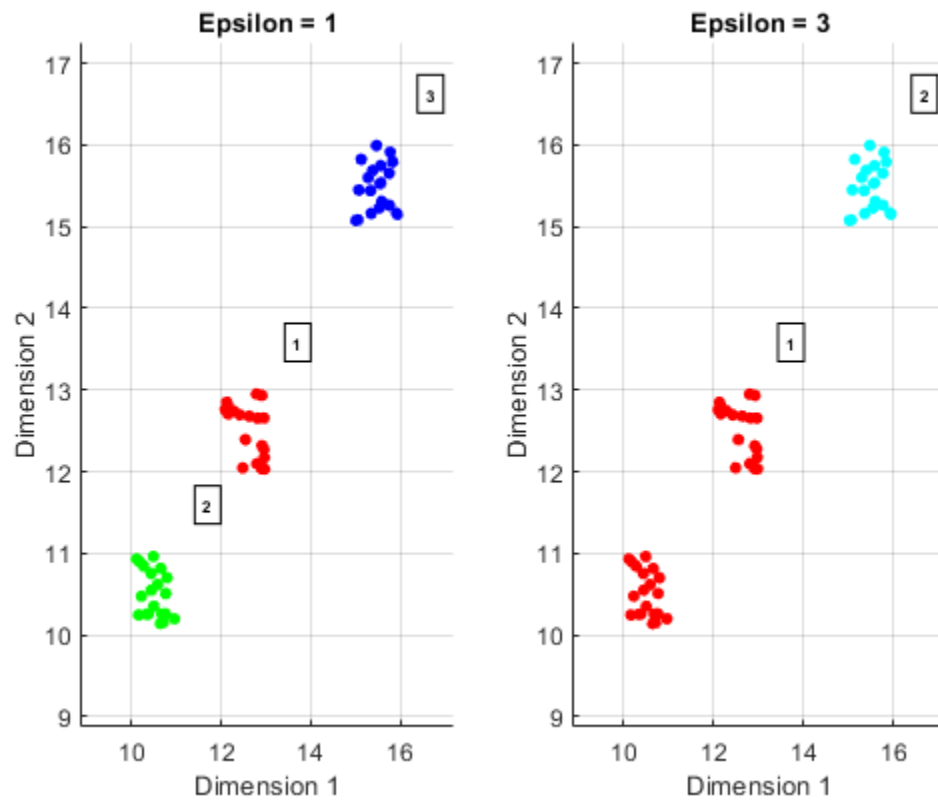
```
idxEpsilon1 = clusterer(x);
```

Cluster the data again but with `Epsilon` set to 3. You can change the value of `Epsilon` because it is a tunable property.

```
clusterer.Epsilon = 3;
idxEpsilon2 = clusterer(x);
```

Plot the clustering results side-by-side. Do this by passing in the axes handles and titles into the `plot` method. The plot shows that for `Epsilon` set to 1, three clusters appear. When `Epsilon` is 3, the two lower clusters are merged into one.

```
hAx1 = subplot(1,2,1);
plot(clusterer,x,idxEpsilon1, ...
      'Parent',hAx1,'Title','Epsilon = 1')
hAx2 = subplot(1,2,2);
plot(clusterer,x,idxEpsilon2, ...
      'Parent',hAx2,'Title','Epsilon = 3')
```

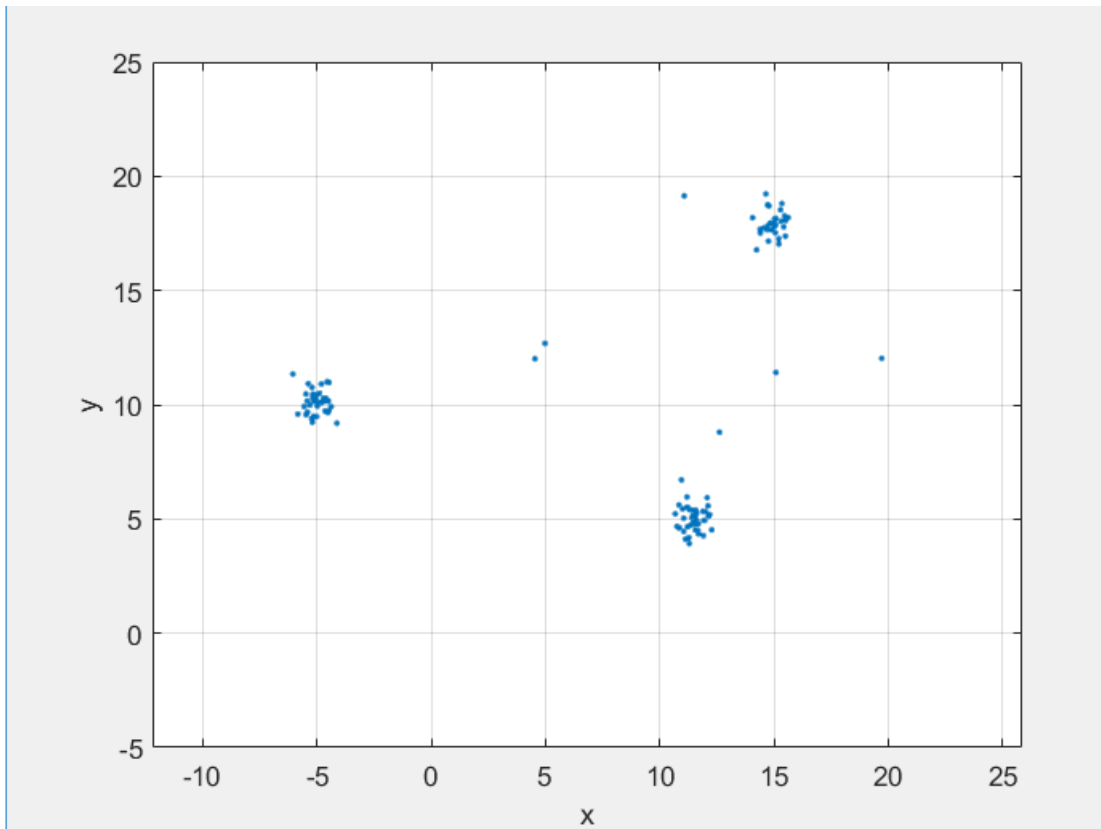


## Algorithms

### Clustering Algorithm

#### Clustering Overview

This section illustrates the basic principles of cluster formation. The figure shows points in a two-dimensional feature space. The clusters are compact and well-separated. A few noise points appear.



#### Clusters Formed from a Single $\varepsilon$ -Neighborhood

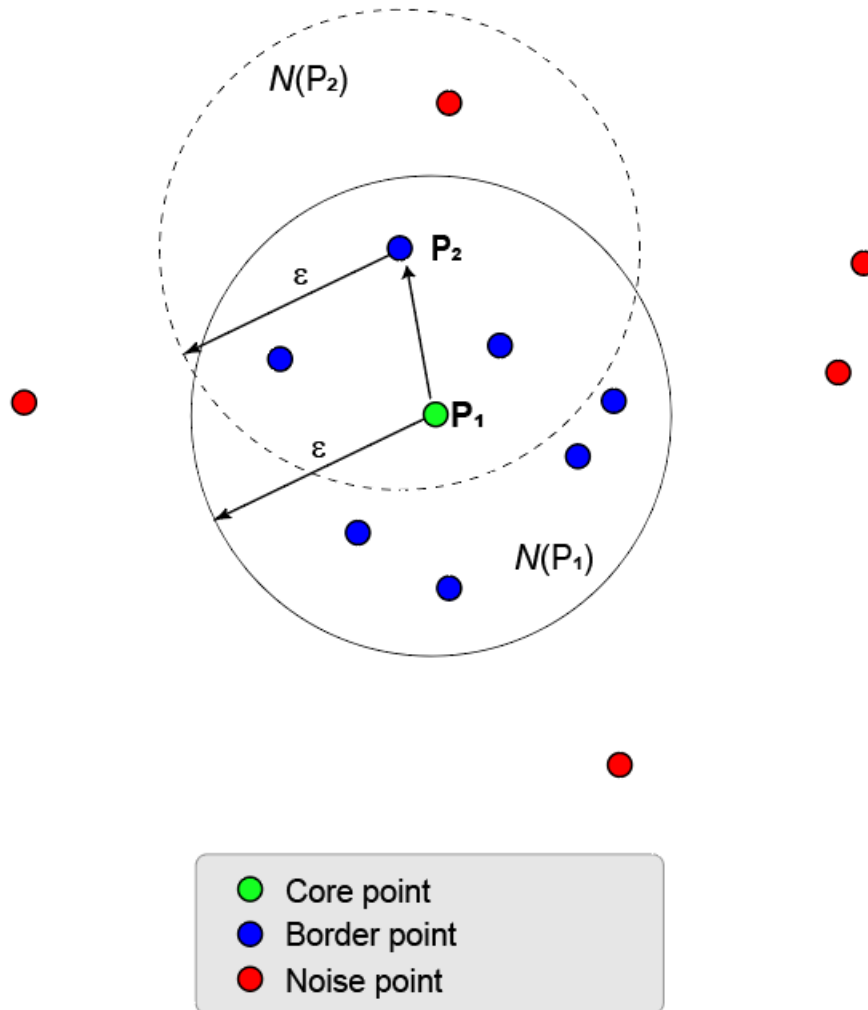
- Clusters start from core points. The first step in the algorithm is identifying all core points.

The figure here shows the point  $P_1$  and its  $\varepsilon$ -neighborhood  $N_\varepsilon(P_1)$ . The  $\varepsilon$ -neighborhood has eight points (including itself) within a radius  $\varepsilon$ . Using the `MinNumPoints` property to set the threshold to 8 means that  $P_1$  is a core point. The blue points that lie within  $N_\varepsilon$  are called *border points*. These border points are *directly density reachable* from the core point  $P_1$ .

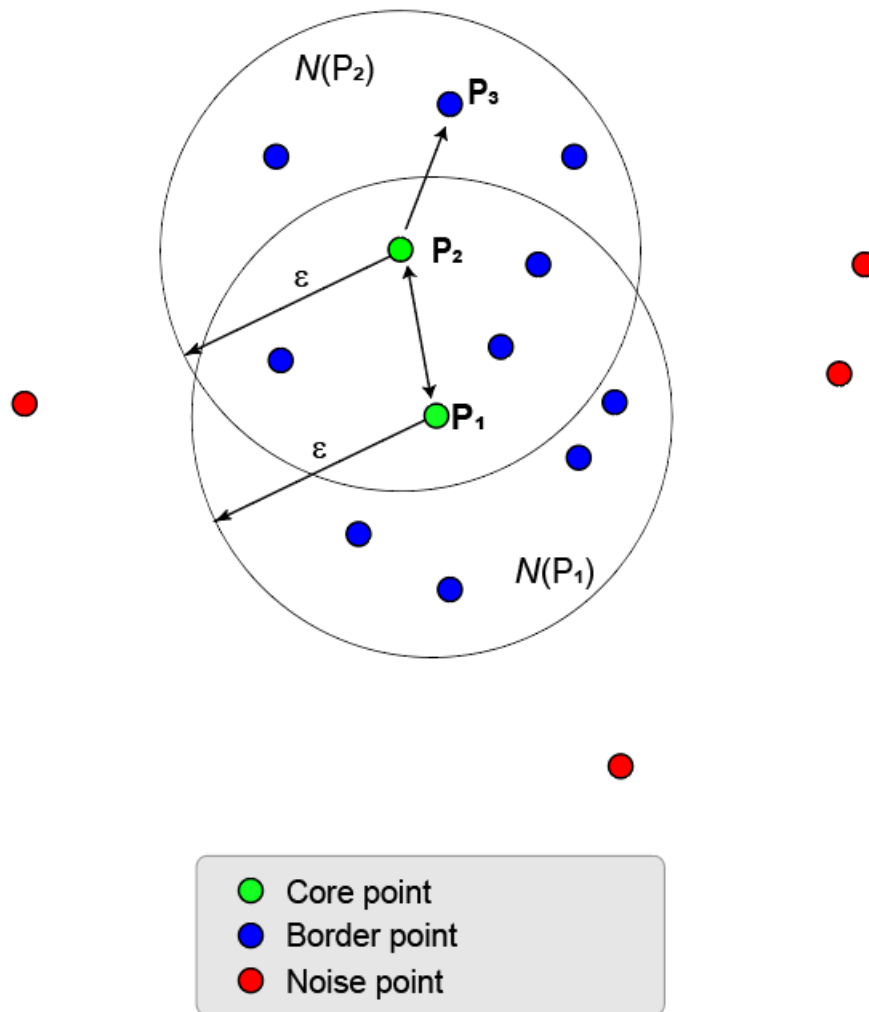
- No other points in the figure have enough neighboring points in their  $\varepsilon$ -neighborhood to become a core point.  $P_2$  is not a core point because it has only five points within its neighborhood.  $P_2$  is directly density reachable from  $P_1$ . The reverse is not true because  $P_2$  is not a core point. The one-way arrow connecting the two points shows this asymmetry.
- Points that fall outside  $N_\varepsilon(P_1)$  are *noise points* (red) and do not belong to the cluster.
- Because no other points are core points, the core point and border points are a maximal set of density-connected points and therefore form a cluster.

#### Cluster of Points from Two $\varepsilon$ -Neighborhoods

- The next figure shows a larger set of points containing two core points,  $P_1$  and  $P_2$ .  $P_2$  is a border point of  $P_1$  but  $P_2$  also has enough points in its own neighborhood to become a core point. Because they are both core points,  $P_1$  is directly density reachable from  $P_2$ , and  $P_1$  is directly density reachable from  $P_2$ . The two-way arrow connecting them shows this symmetry.



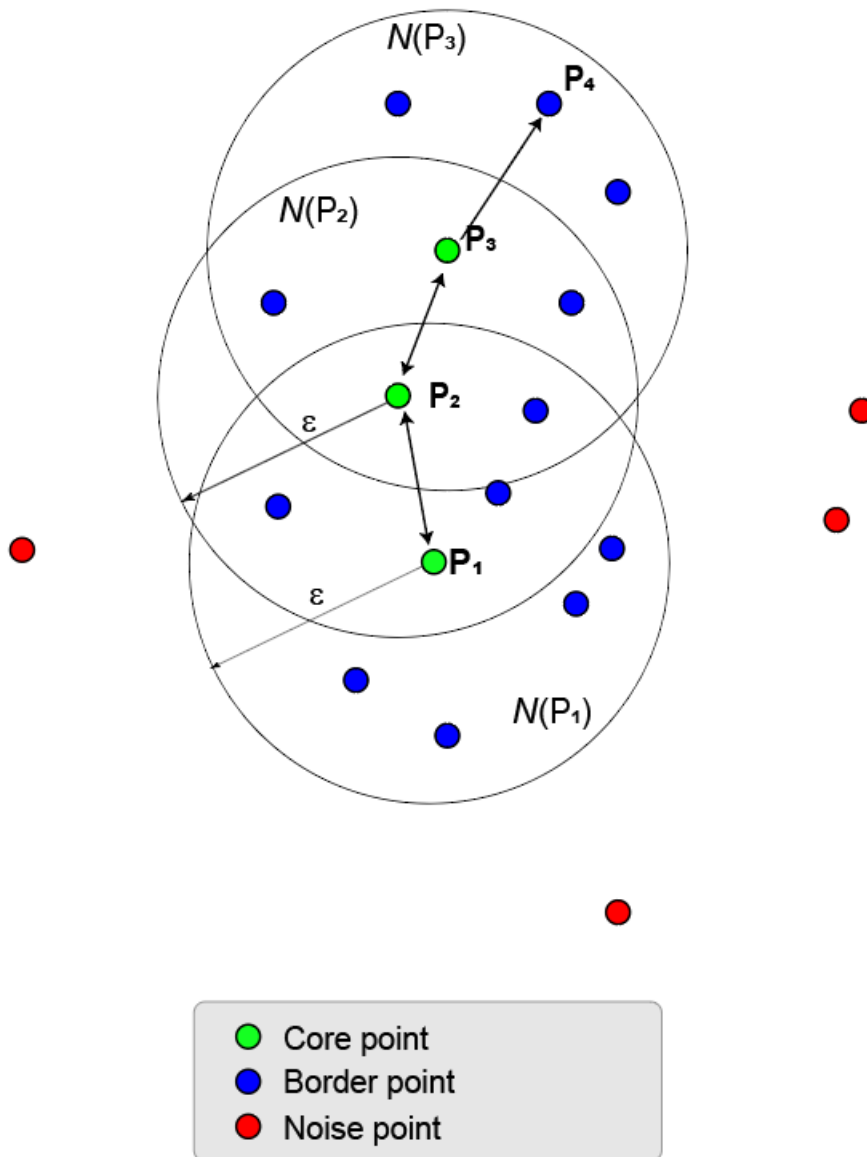
- $P_3$  is directly density reachable from  $P_2$  but not from  $P_1$  (as indicated by the one-way arrow). However,  $P_3$  is called simply *density reachable* from  $P_1$ .
- Because no other points are core points, the two core points and their border points form a maximal set of density-connected points and form one cluster.



#### Cluster Points in Adjacent $\epsilon$ -Neighborhoods

- This process of growing a cluster can be extended from core point to core point until there are no more core points to add. The core points and the border points belong to the same cluster. In general, a point  $P_n$  is density reachable from point  $P_1$  when there is a chain of core points,  $P_1, P_2, P_3, \dots, P_{n-1}$  such that each core point  $P_{i+1}$  is directly density reachable from  $P_i$ , and  $P_n$  is directly density reachable from  $P_{n-1}$ .



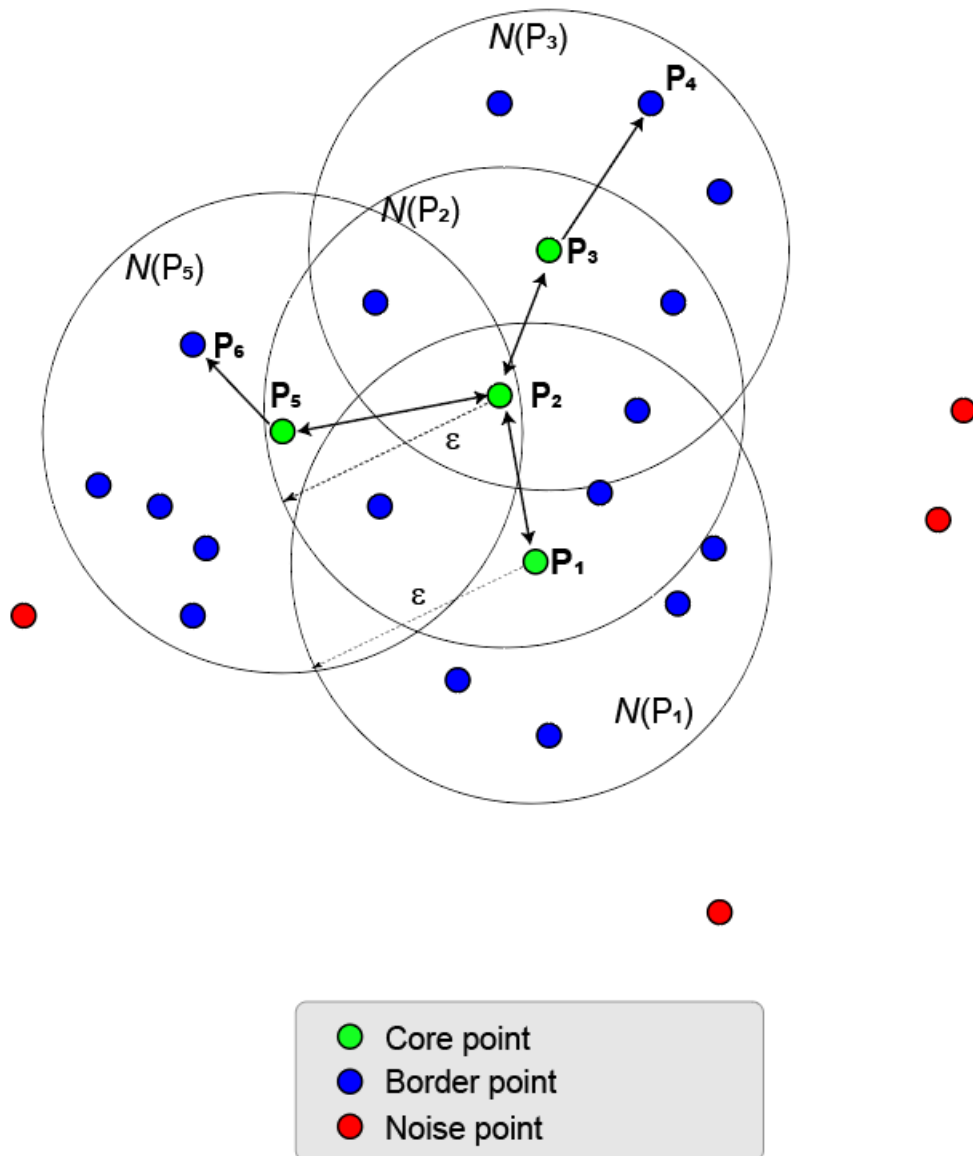


### Density Connectivity

The next figure illustrates some properties of density connectivity.

- A cluster can have multiple branching chains, for example  $(P_1, P_2, P_3, P_4)$  and  $(P_1, P_2, P_5, P_6)$ .
- Two points,  $P_6$  and  $P_4$ , are *density connected* when there is a third point  $P_2$  such that  $P_6$  and  $P_4$  are density reachable from  $P_2$ .

- Two density connected points are not necessarily density reachable from one another.
- A maximal set of density connected points define a cluster. It does not matter which core point is the starting core point.
- All points in a cluster are density reachable from all core points.



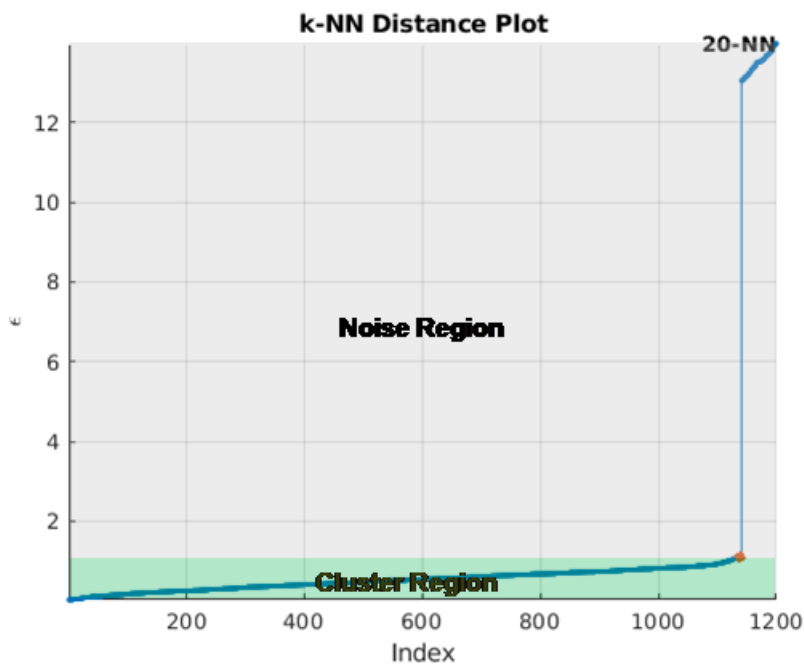
### Estimate Epsilon

DBSCAN clustering requires a value for the neighborhood size parameter  $\epsilon$ . The `clusterDBSCAN` object and the `clusterDBSCAN.estimateEpsilon` function use a  $k$ -nearest-neighbor search to

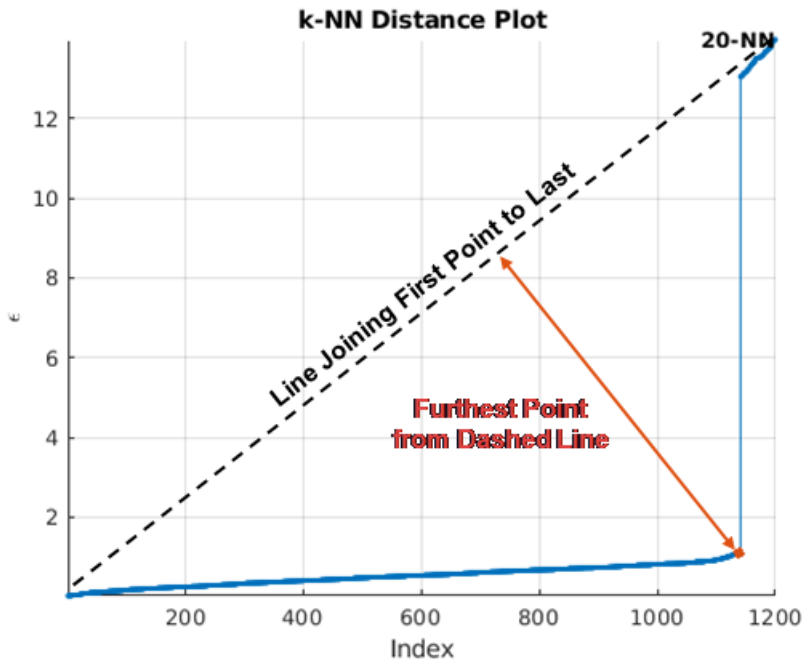
estimate a scalar epsilon. Let  $D$  be the distance of any point  $P$  to its  $k^{\text{th}}$  nearest neighbor. Define a  $D_k(P)$ -neighborhood as a neighborhood surrounding  $P$  that contains its  $k$ -nearest neighbors. There are  $k + 1$  points in the  $D_k(P)$ -neighborhood including the point  $P$  itself. An outline of the estimation algorithm is:

- For each point, find all the points in its  $D_k(P)$ -neighborhood
- Accumulate the distances in all  $D_k(P)$ -neighborhoods for all points into a single vector.
- Sort the vector by increasing distance.
- Plot the sorted  $k$ -dist graph, which is the sorted distance against point number.
- Find the knee of the curve. The value of the distance at that point is an estimate of epsilon.

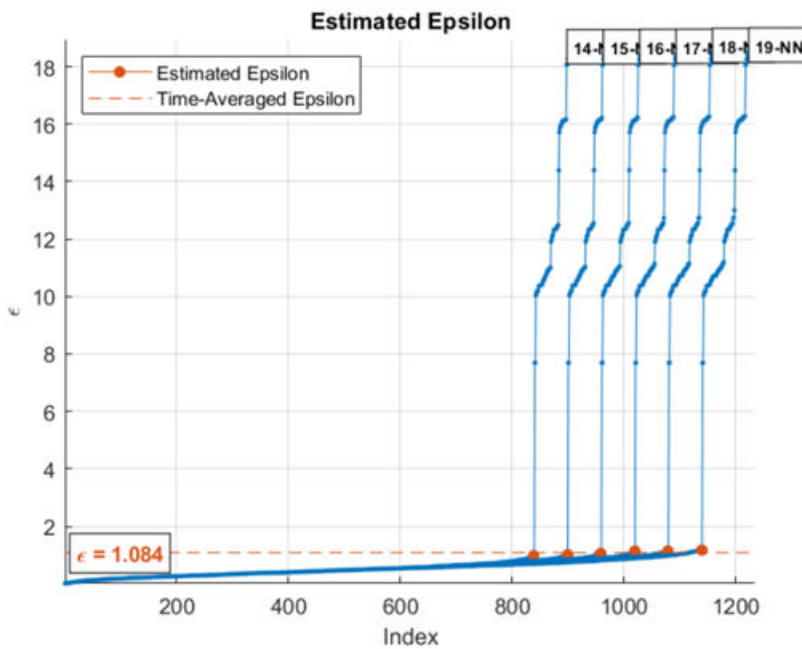
The figure here shows distance plotted against point index for  $k = 20$ . The knee occurs at approximately 1.5. Any points below this threshold belong to a cluster. Any points above this value are noise.



There are several methods to find the knee of the curve. `clusterDBSCAN` and `clusterDBSCAN.estimateEpsilon` first define the line connecting the first and last points of the curve. The ordinate of the point on the sorted  $k$ -dist graph furthest from the line and perpendicular to the line defines epsilon.



When you specify a range of  $k$  values, the algorithm averages the estimate epsilon values for all curves. This figure shows that epsilon is fairly insensitive to  $k$  for  $k$  ranging from 14 through 19.



To create a single  $k$ -NN distance graph, set the `MinNumPoints` property equal to the `MaxNumPoints` property.

## Choosing the Minimum Number of Points

The purpose of `MinNumPoints` is to smooth the density estimates. Because a cluster is a maximal set of density-connected points, choose smaller values when the expected number of detections in a cluster is unknown. However, smaller values make the DBSCAN algorithm more susceptible to noise. A general guideline for choosing `MinNumPoints` is:

- Generally, set `MinNumPoints = 2P` where  $P$  is the number of feature dimensions in  $X$ .
- For data sets that have one or more of the following properties:
  - many noise points
  - large number of points,  $N$
  - large dimensionality,  $P$
  - many duplicates

increasing `MinNumPoints` can often improve clustering results.

## Ambiguous Data

The clustering algorithm is general enough to process ambiguities in any feature, but applying clustering to range and Doppler ambiguities in radar are important applications.

### Range Ambiguity

The time delay between pulse transmission and reception determines the range,  $R$ , of a target.  $R$  is proportional to time delay,  $t$ , by

$$R = \frac{ct}{2}$$

where  $c$  is the speed of light. Time is measured from the transmission time of the pulse. If only one pulse is transmitted, the equation accurately determines the range.

Often, the radar transmits multiple pulses spaced at intervals  $T$ , the pulse repetition interval (PRI). Range ambiguities occur when the echoes from one pulse are not received before the next pulse is transmitted. Range is computed from the time difference of the arrival of the received pulse from the transmission time of the most recent transmitted pulse. Therefore the range can be incorrect by some integer multiple of the unambiguous range. The unambiguous range of a radar system is the maximum range at which a target can be located to guarantee that the reflected pulse from that target corresponds to the most recent transmitted pulse. The PRI determines the unambiguous range.

$$R_{\max} = \frac{cT}{2}$$

The range of a detection less than  $R_{\max}$  is an unambiguous range. Range disambiguation clusters detections that cross ambiguous range boundaries.

Turn on disambiguation by setting the `EnableDisambiguation` to `true`. Then, use the `AmbiguousDimension` property to select the column in the input data corresponding to range. Set the actual ambiguity limits for range using the `amblims` argument at execution time.

### Doppler Ambiguity

Doppler aliasing occurs when echoes arrive from targets that move fast enough for the Doppler frequency to exceed the pulse repetition frequency (PRF). If the Doppler shift is greater than  $\frac{1}{2}$  PRF

or less than  $-\frac{1}{2}$  PRF, the Doppler shift is aliased into the range  $(-\frac{1}{2}$  PRF,  $\frac{1}{2}$  PRF). This range is called the unambiguous Doppler. Turn on disambiguation by setting the `Enabledisambiguation` to `true`. Then, use the `AmbiguousDimension` property to select the column in the input data corresponding to Doppler. Set the actual ambiguity limits for Doppler using the `amblims` argument at execution time. Doppler ambiguity implies radial speed ambiguity as well. Make sure that `amblims` matches the interpretation of the feature.

## References

- [1] Ester M., Kriegel H.-P., Sander J., and Xu X. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". *Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining*, Portland, OR, AAAI Press, 1996, pp. 226-231.
- [2] Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 2017. "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN". *ACM Trans. Database Syst.* 42, 3, Article 19 (July 2017), 21 pages.
- [3] Dominik Kellner, Jens Klappstein and Klaus Dietmayer, "Grid-Based DBSCAN for Clustering Extended Objects in Radar Data", *2012 IEEE Intelligent Vehicles Symposium*.
- [4] Thomas Wagner, Reinhard Feger, and Andreas Stelzer, "A Fast Grid-Based Clustering Algorithm for Range/Doppler/DoA Measurements", *Proceedings of the 13th European Radar Conference*.
- [5] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, Jörg Sander, "OPTICS: Ordering Points To Identify the Clustering Structure", *Proc. ACM SIGMOD'99 Int. Conf. on Management of Data*, Philadelphia PA, 1999.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`clusterDBSCAN.discoverClusters` | `clusterDBSCAN.estimateEpsilon` | `clusterDBSCAN.plot`

**Introduced in R2021a**

# clusterDBSCAN.discoverClusters

Find cluster hierarchy in data

## Syntax

```
[order,reachdist] = clusterDBSCAN.discoverClusters(X,maxepsilon,minnumpoints)
clusterDBSCAN.discoverClusters(X,maxepsilon,minnumpoints)
```

## Description

`[order,reachdist] = clusterDBSCAN.discoverClusters(X,maxepsilon,minnumpoints)` returns a cluster-ordered list of points, `order`, and the reachability distances, `reachdist`, for each point in the data `X`. Specify the maximum epsilon, `maxepsilon`, and the minimum number of points, `minnumpoints`. The method implements the *Ordering Points To Identify the Clustering Structure* (OPTICS) algorithm. The OPTICS algorithm is useful when clusters have varying densities.

`clusterDBSCAN.discoverClusters(X,maxepsilon,minnumpoints)` displays a bar graph representing the cluster hierarchy.

## Examples

### Display Cluster Hierarchy

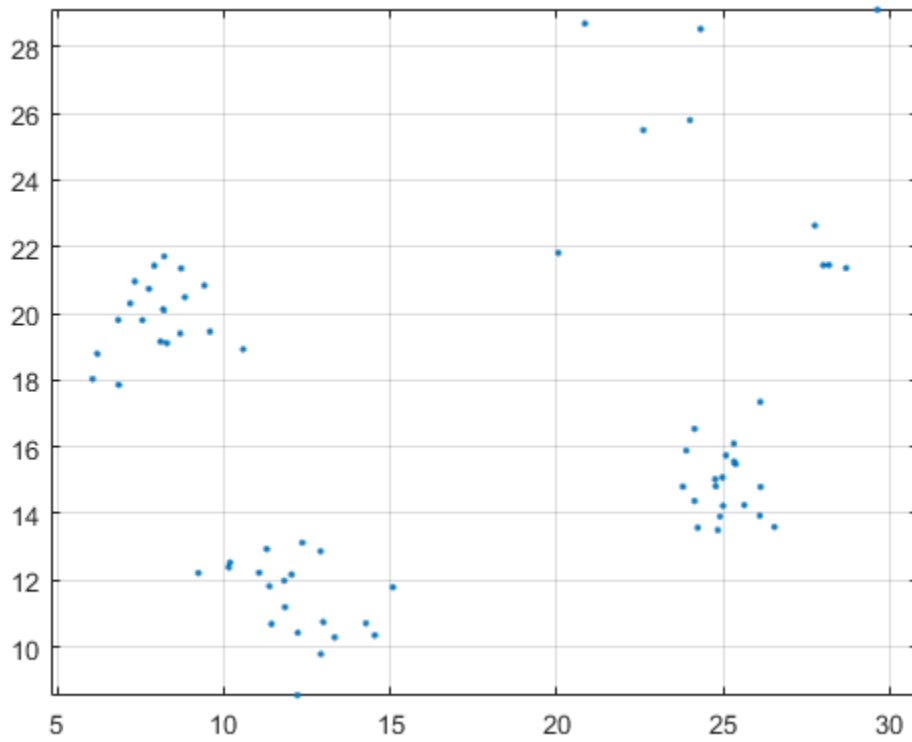
Create target data with random detections in xy Cartesian coordinates. Use the `clusterDBSCAN.discoverClusters` object functions to reveal the underlying cluster hierarchy.

First, set `clusterDBSCAN.discoverClusters` parameters.

```
maxEpsilon = 10;
minNumPoints = 6;
```

Create random target data.

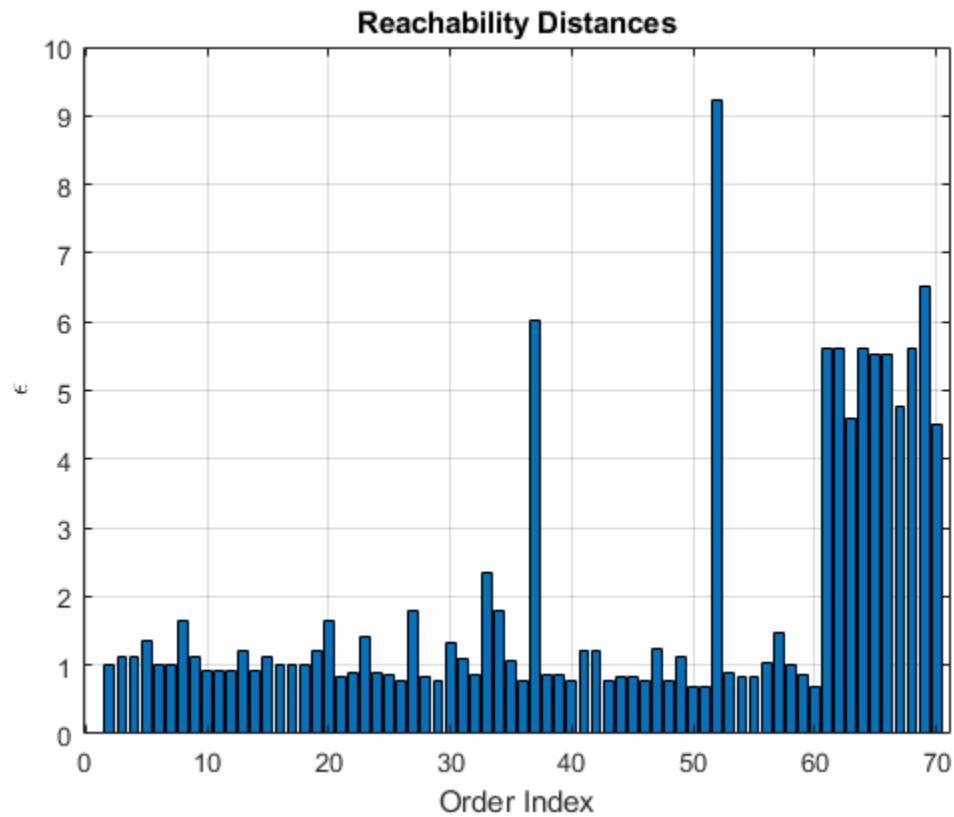
```
X = [randn(20,2) + [11.5,11.5]; randn(20,2) + [25,15]; randn(20,2) + [8,20]; 10*rand(10,2) + [20,20];
plot(X(:,1),X(:,2),'.')
axis equal
grid
```



Plot the cluster hierarchy.

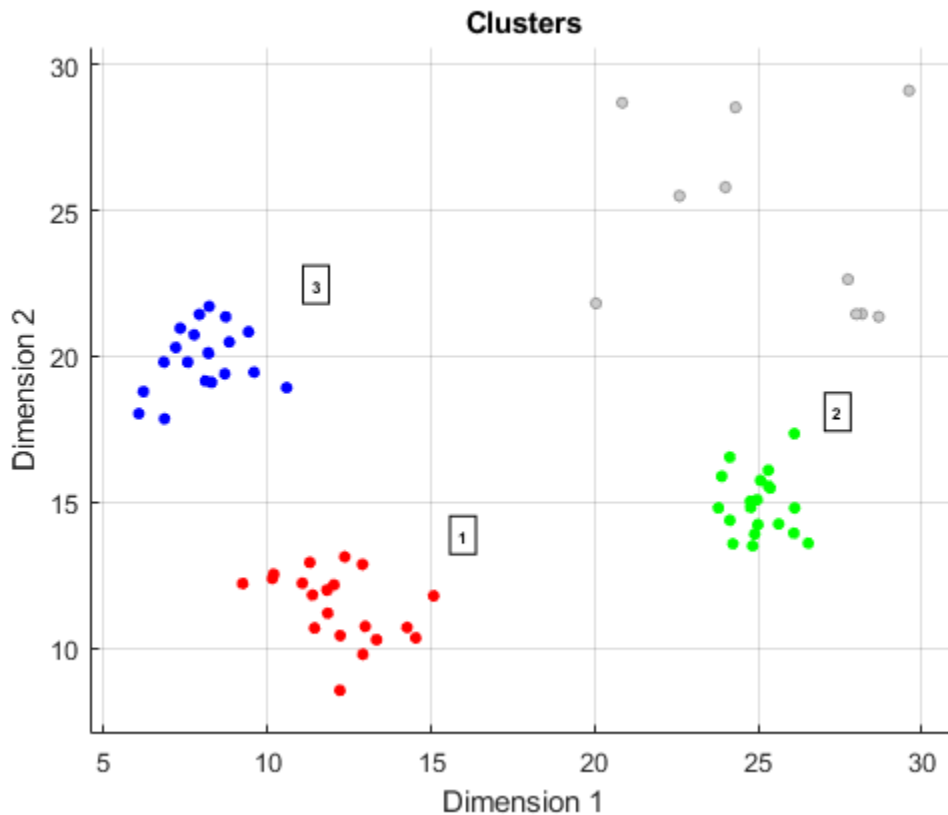
```
clusterDBSCAN.discoverClusters(X,maxEpsilon,minNumPoints)
```





From a visual inspection of the plot, choose Epsilon as 2 and then perform the clustering using the `clusterDBSCAN` object and plot the resultant clusters.

```
clusterer = clusterDBSCAN('MinNumPoints',6,'Epsilon',2, ...
    'EnableDisambiguation',false);
[idx,cidx] = clusterer(X);
plot(clusterer,X,idx)
```



## Input Arguments

### **X** — Input feature data

real-valued  $N$ -by- $P$  matrix

Input feature data, specified as a real-valued  $N$ -by- $P$  matrix. The  $N$  rows correspond to feature points in a  $P$ -dimensional feature space. The  $P$  columns contain the values of the features over which clustering takes place. The DBSCAN algorithm can cluster any type of data with appropriate `MinNumPoints` and `Epsilon` settings. For example, a two-column input can contain the  $xy$  Cartesian coordinates, or range and Doppler.

Data Types: double

### **maxepsilon** — Maximum epsilon size

positive scalar

Maximum epsilon size to use in the cluster hierarchy search, specified as a positive scalar. The epsilon parameter defines the clustering neighborhood around a point. Reducing `maxepsilon` results in shorter run times. Setting `maxepsilon` to `inf` identifies all possible clusters.

The OPTICS algorithm is relatively insensitive to parameter settings, but choosing larger parameters can improve results.

Example: 5.0

Data Types: double

**minnumpoints — Minimum number of points**

positive integer

Minimum number of points used as a threshold, specified as a positive integer. The threshold sets the minimum number of points for a cluster.

The OPTICS algorithm is relatively insensitive to parameter settings, but choosing larger parameters can improve results.

Example: 10

Data Types: double

**Output Arguments****order — Cluster hierarchy**integer-valued 1-by- $N$  row vector

Cluster ordered list of sample indices, returned as an integer-valued 1-by- $N$  row vector.  $N$  is the number of rows in the input data matrix  $X$ .

**reachdist — Reachability distance**positive, real-valued 1-by- $N$  row vector

Reachability distance, returned as a positive, real-valued 1-by- $N$  row vector.  $N$  is the number of rows in the input data matrix  $X$ .

Data Types: double

**Algorithms**

The outputs of `clusterDBSCAN.discoverClusters` let you create a reachability-plot from which the hierarchical structure of the clusters can be visualized. A reachability-plot contains ordered points on the  $x$ -axis and the reachability distances on the  $y$ -axis. Use the outputs to examine the cluster structure over a broad range of parameter settings. You can use the output to help estimate appropriate epsilon clustering thresholds for the DBSCAN algorithm. Points belonging to a cluster have small reachability distances to their nearest neighbor, and clusters appear as valleys in the reachability plot. Deeper valleys correspond to denser clusters. Determine epsilon from the ordinate of the bottom of the valleys.

OPTICS assumes that dense clusters are entirely contained by less dense clusters. OPTICS processes data in the correct order by tracking the point density neighborhoods. This process is performed by ordering data points by the shortest reachability distances, guaranteeing that clusters with higher density are identified first.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Code generation is not supported for graphics output.

**See Also**

`clusterDBSCAN` | `clusterDBSCAN.estimateEpsilon` | `clusterDBSCAN.plot`

**Introduced in R2021a**

# clusterDBSCAN.estimateEpsilon

Estimate neighborhood clustering threshold

## Syntax

```
epsilon = clusterDBSCAN.estimateEpsilon(X,MinNumPoints,MaxNumPoints)
clusterDBSCAN.estimateEpsilon(X,MinNumPoints,MaxNumPoints)
```

## Description

`epsilon = clusterDBSCAN.estimateEpsilon(X,MinNumPoints,MaxNumPoints)` returns an estimate of the neighborhood clustering threshold, `epsilon`, used in the density-based spatial clustering of applications with noise (DBSCAN) algorithm. `epsilon` is computed from input data `X` using a  $k$ -nearest neighbor ( $k$ -NN) search. `MinNumPoints` and `MaxNumPoints` set a range of  $k$ -values for which `epsilon` is calculated. The range extends from `MinNumPoints - 1` through `MaxNumPoints - 1`.  $k$  is the number of neighbors of a point, which is one less than the number of points in a neighborhood.

`clusterDBSCAN.estimateEpsilon(X,MinNumPoints,MaxNumPoints)` displays a figure showing the  $k$ -NN search curves and the estimated `epsilon`.

## Examples

### Estimate Epsilon from Data

Create simulated target data and use the `clusterDBSCAN.estimateEpsilon` function to calculate an appropriate `epsilon` threshold.

Create the target data as  $xy$  Cartesian coordinates.

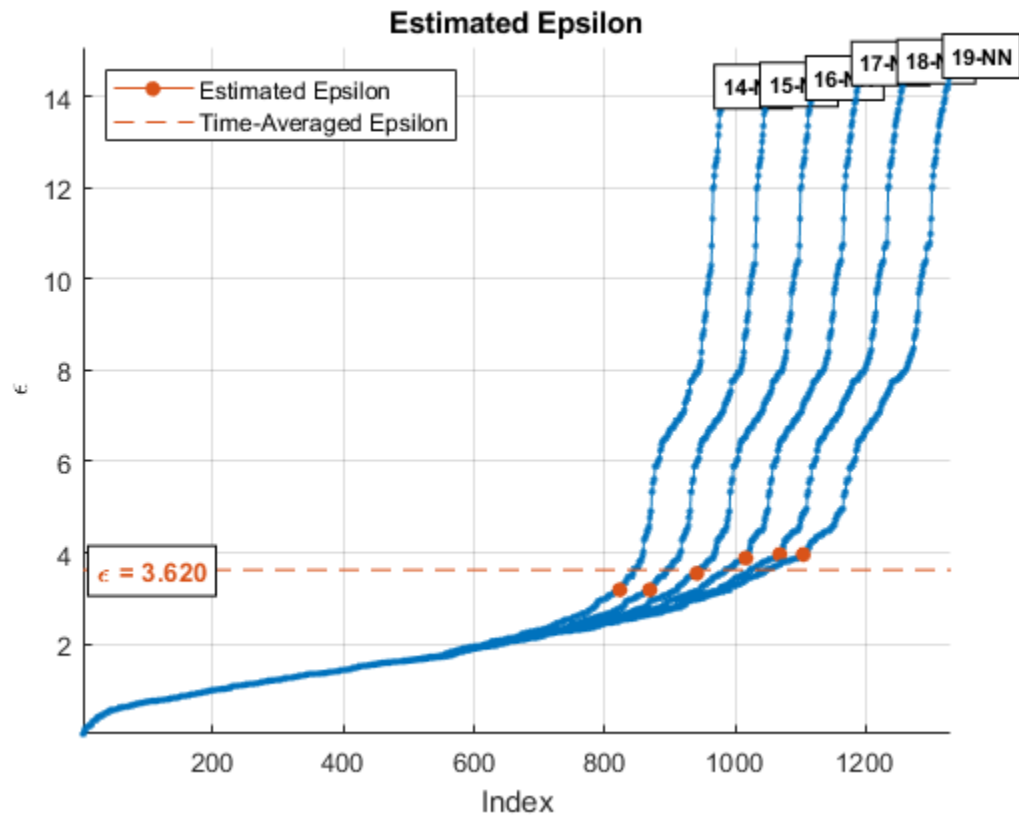
```
X = [randn(20,2) + [11.5,11.5]; randn(20,2) + [25,15]; ...
     randn(20,2) + [8,20]; 10*rand(10,2) + [20,20]];
```

Set the range of values for the  $k$ -NN search.

```
minNumPoints = 15;
maxNumPoints = 20;
```

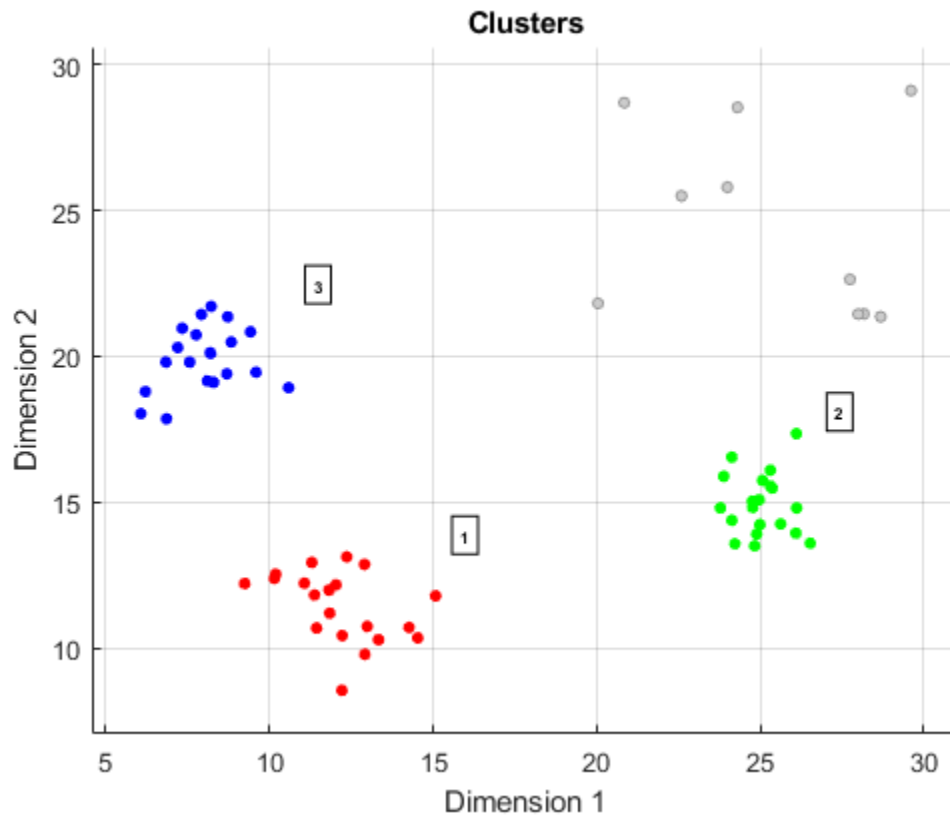
Estimate the clustering threshold `epsilon` and display its value on a plot.

```
clusterDBSCAN.estimateEpsilon(X,minNumPoints,maxNumPoints)
```



Use the estimated Epsilon value, 3.62, in the `clusterDBSCAN` clusterer. Then, plot the clusters.

```
clusterer = clusterDBSCAN('MinNumPoints',6,'Epsilon',3.62, ...  
    'EnableDisambiguation',false);  
[idx,cidx] = clusterer(X);  
plot(clusterer,X,idx)
```



## Input Arguments

### **X** — Input feature data

real-valued  $N$ -by- $P$  matrix

Input feature data, specified as a real-valued  $N$ -by- $P$  matrix. The  $N$  rows correspond to feature points in a  $P$ -dimensional feature space. The  $P$  columns contain the values of the features over which clustering takes place. The DBSCAN algorithm can cluster any type of data with appropriate `MinNumPoints` and `Epsilon` settings. For example, a two-column input can contain the  $xy$  Cartesian coordinates, or range and Doppler.

Data Types: double

### **MinNumPoints** — Starting value of $k$ -NN search range

positive integer

The starting value of the  $k$ -NN search range, specified as a positive integer. `MinNumPoints` is used to specify the starting value of  $k$  in the  $k$ -NN search range. The starting value of  $k$  is one less than `MinNumPoints`.

Example: 10

Data Types: double

### **MaxNumPoints** — Set end value of $k$ -NN search range

positive integer

The end value of  $k$ -NN search range, specified as a positive integer. `MaxNumPoints` is used to specify the ending value of  $k$  in the  $k$ -NN search range. The ending value of  $k$  is one less than `MaxNumPoints`.

## Output Arguments

### **epsilon** — Estimated epsilon

positive scalar

Estimated epsilon, returned as a positive scalar.

## Algorithms

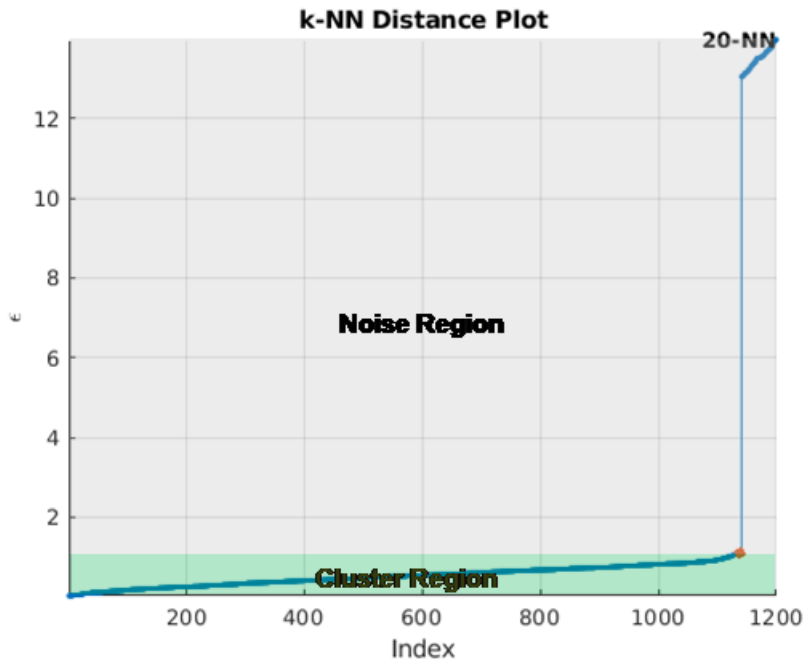
### Estimate Epsilon

DBSCAN clustering requires a value for the neighborhood size parameter  $\epsilon$ . The `clusterDBSCAN` object and the `clusterDBSCAN.estimateEpsilon` function use a  $k$ -nearest-neighbor search to estimate a scalar epsilon. Let  $D$  be the distance of any point  $P$  to its  $k^{\text{th}}$  nearest neighbor. Define a  $D_k(P)$ -neighborhood as a neighborhood surrounding  $P$  that contains its  $k$ -nearest neighbors. There are  $k + 1$  points in the  $D_k(P)$ -neighborhood including the point  $P$  itself. An outline of the estimation algorithm is:

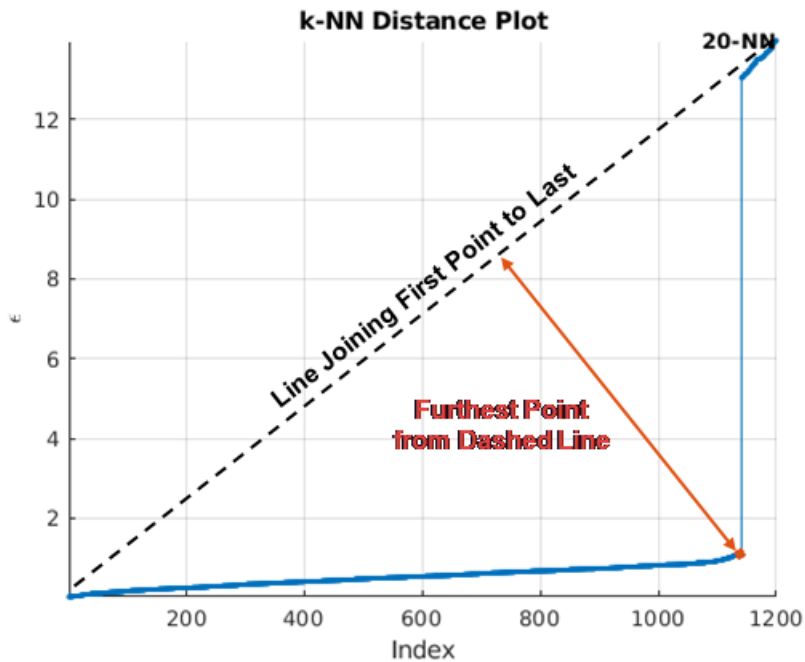
- For each point, find all the points in its  $D_k(P)$ -neighborhood
- Accumulate the distances in all  $D_k(P)$ -neighborhoods for all points into a single vector.
- Sort the vector by increasing distance.
- Plot the sorted  $k$ -dist graph, which is the sorted distance against point number.
- Find the knee of the curve. The value of the distance at that point is an estimate of epsilon.

The figure here shows distance plotted against point index for  $k = 20$ . The knee occurs at approximately 1.5. Any points below this threshold belong to a cluster. Any points above this value are noise.

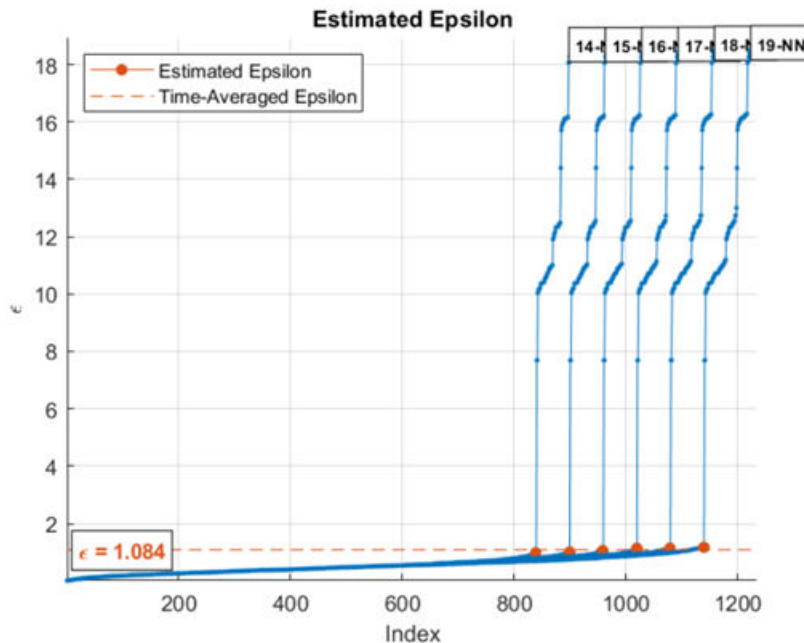




There are several methods to find the knee of the curve. `clusterDBSCAN` and `clusterDBSCAN.estimateEpsilon` first define the line connecting the first and last points of the curve. The ordinate of the point on the sorted  $k$ -dist graph furthest from the line and perpendicular to the line defines epsilon.



When you specify a range of  $k$  values, the algorithm averages the estimate epsilon values for all curves. This figure shows that epsilon is fairly insensitive to  $k$  for  $k$  ranging from 14 through 19.



To create a single  $k$ -NN distance graph, set the `MinNumPoints` property equal to the `MaxNumPoints` property.

### Choosing the Minimum and Maximum Number of Points

The purpose of `MinNumPoints` is to smooth the density estimates. Because a cluster is a maximal set of density-connected points, choose smaller values when the expected number of detections in a cluster is unknown. However, smaller values make the DBSCAN algorithm more susceptible to noise. A general guideline for choosing `MinNumPoints` is:

- Generally, set `MinNumPoints` =  $2P$  where  $P$  is the number of feature dimensions in  $X$ .
- For data sets that have one or more of the following properties:
  - many noise points
  - large number of points,  $N$
  - large dimensionality,  $P$
  - many duplicates

increasing `MinNumPoints` can often improve clustering results.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Code generation is not supported for graphics output.

## **See Also**

`clusterDBSCAN.discoverClusters` | `clusterDBSCAN.plot` | `clusterDBSCAN`

**Introduced in R2021a**

## clusterDBSCAN.plot

Plot clusters

### Syntax

```
fh = plot(clusterer,X,idx)
fh = plot( __ , 'Parent',ax)
fh = plot( __ , 'Title',titlestr)
```

### Description

`fh = plot(clusterer,X,idx)` displays a plot of DBSCAN clustering results and returns a figure handle, `fh`. Inputs are the cluster object, `clusterer`, the input data matrix, `X`, and cluster indices, `idx`.

`fh = plot( __ , 'Parent',ax)` also specifies the axes, `ax`, of the cluster results plot.

`fh = plot( __ , 'Title',titlestr)` also specifies the title, `titlestr`, of the cluster results plot.

### Examples

#### Cluster Detections in Range and Doppler

Create detections of extended objects with measurements in range and Doppler. Assume the maximum unambiguous range is 20 m and the unambiguous Doppler span extends from  $-30$  Hz to 30 Hz. Data for this example is contained in the `dataClusterDBSCAN.mat` file. The first column of the data matrix represents range, and the second column represents Doppler.

The input data contains the following extended targets and false alarms:

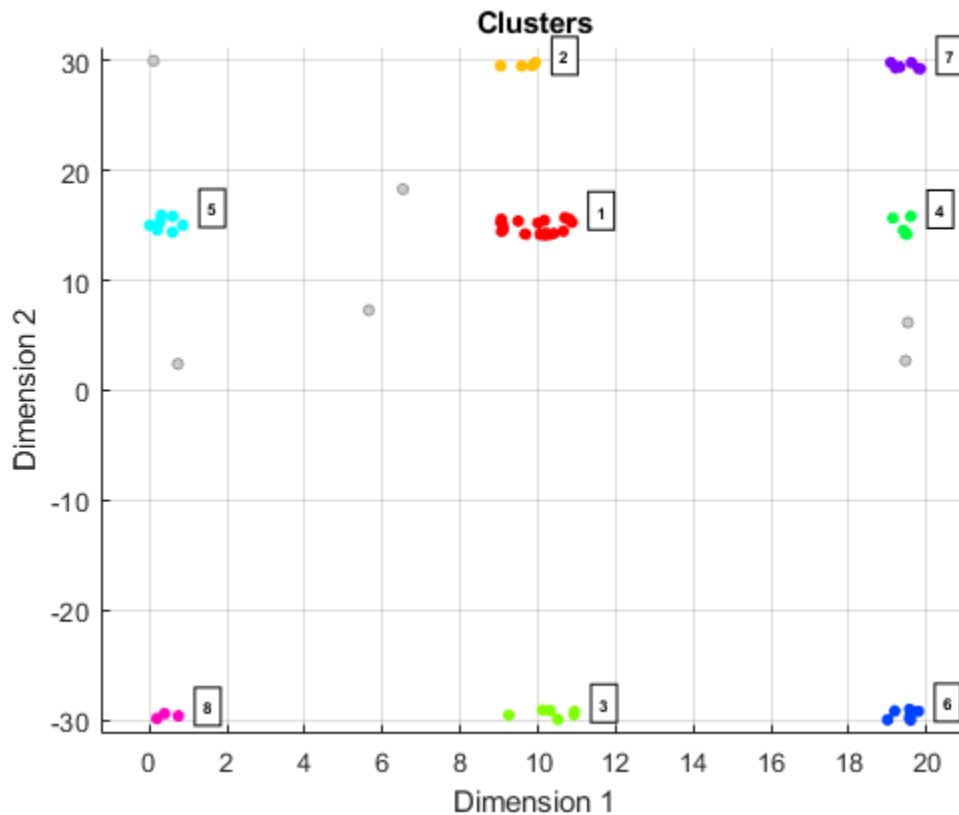
- an unambiguous target located at (10, 15)
- an ambiguous target in Doppler located at(10,  $-30$ )
- an ambiguous target in range located at (20, 15)
- an ambiguous target in range and Doppler located at (20, 30)
- 5 false alarms

Create a `clusterDBSCAN` object and specify that disambiguation is not performed by setting `EnableDisambiguation` to `false`. Solve for the cluster indices.

```
load('dataClusterDBSCAN.mat');
cluster1 = clusterDBSCAN('MinNumPoints',3,'Epsilon',2, ...
    'EnableDisambiguation',false);
idx = cluster1(x);
```

Use the `clusterDBSCAN` plot object function to display the clusters.

```
plot(cluster1,x,idx)
```



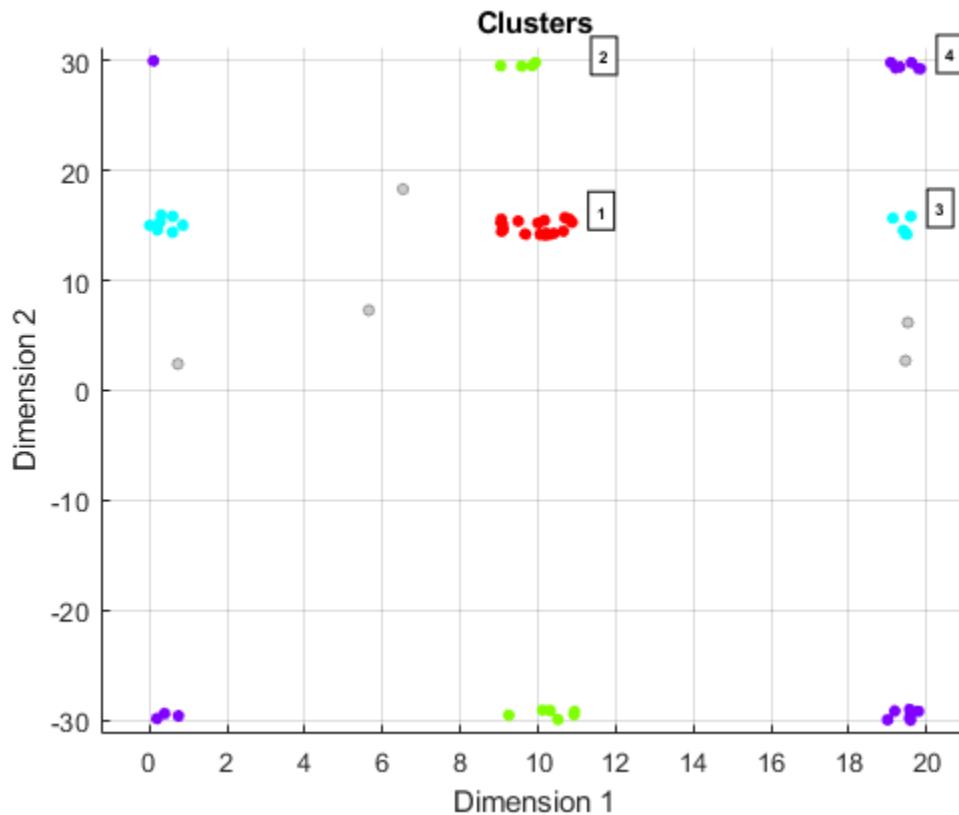
The plot indicates that there are eight apparent clusters and six noise points. The 'Dimension 1' label corresponds to range and the 'Dimension 2' label corresponds to Doppler.

Next, create another `clusterDBSCAN` object and set `EnableDisambiguation` to `true` to specify that clustering is performed across the range and Doppler ambiguity boundaries.

```
cluster2 = clusterDBSCAN('MinNumPoints',3,'Epsilon',2, ...
    'EnableDisambiguation',true,'AmbiguousDimension',[1 2]);
```

Perform the clustering using ambiguity limits and then plot the clustering results. The DBSCAN clustering results correctly show four clusters and five noise points. For example, the points at ranges close to zero are clustered with points near 20 m because the maximum unambiguous range is 20 m.

```
amblims = [0 maxRange; minDoppler maxDoppler];
idx = cluster2(x,amblims);
plot(cluster2,x,idx)
```



## Input Arguments

### **clusterer** – Clusterer object

`clusterDBSCAN` object

Clusterer object, specified as a `clusterDBSCAN` object.

### **X** – Input data to cluster

real-valued  $N$ -by- $P$  matrix

Input data, specified as a real-valued  $N$ -by- $P$  matrix. The  $N$  rows correspond to points in a  $P$ -dimensional feature space. The  $P$  columns contain the values of the features over which clustering takes place. For example, a two-column input can contain Cartesian coordinates  $x$  and  $y$ , or range and Doppler.

Data Types: `double`

### **idx** – Cluster indices

$N$ -by-1 integer-valued column vector

Cluster indices, specified as an  $N$ -by-1 integer-valued column vector. Cluster indices represent the clustering results of the DBSCAN algorithm contained in the first output argument of `clusterDBSCAN`. `idx` values start at one and are consecutively numbered. The plot object function labels each cluster with the cluster index. A value of `-1` in `idx` indicates a DBSCAN noise point. Noise points are not labeled.

Data Types: double

**ax — Axes of plot**

Axes handle

Axes of plot, specified as an Axes object handle.

Data Types: double

**titlestr — Plot title**

character vector | string

Plot title, specified as a character vector or string.

Example: 'Range-Doppler Clusters'

Data Types: char | string

**Output Arguments****fh — Figure handle of plot**

positive scalar

Figure handle of plot, returned as a positive scalar.

**See Also**

[clusterDBSCAN.discoverClusters](#) | [clusterDBSCAN](#) | [clusterDBSCAN.estimateEpsilon](#)

**Introduced in R2021a**

## radarDataGenerator

Generate radar detections and tracks

### Description

The `radarDataGenerator` System object™ generates detection or track reports of targets. You can specify the detection mode of the sensor as monostatic, bistatic, or electronic support measure (ESM) through the `DetectionMode` property. You can use `radarDataGenerator` to simulate clustered or unclustered detections with added random noise, and also generate false alarm detections. You can fuse the generated detections with other sensor data and track objects using a `radarTracker` object. You can also output tracks directly from the `radarDataGenerator` object. To configure whether targets are output as clustered detections, unclustered detections, or tracks, use the `TargetReportFormat` property. You can add `radarDataGenerator` to a `Platform` and then use the radar in a `radarScenario`.

Using a single-exponential model, the radar computes range and elevation biases caused by propagation through the troposphere. A range bias means that measured ranges are greater than the line-of-sight range to the target. Elevation bias means that the measured elevations are above their true elevations. Biases are larger when the line-of-sight path between the radar and target passes through lower altitudes because the atmosphere is thicker at these altitudes. See “References” on page 4-207 for more details.

To generate radar detection and track reports:

- 1 Create the `radarDataGenerator` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```

rdr = radarDataGenerator
rdr = radarDataGenerator(id)
rdr = radarDataGenerator( ___, scanConfig)
rdr = radarDataGenerator( ___, Name, Value)

```

### Description

`rdr = radarDataGenerator` creates a monostatic radar sensor that reports clustered detections and uses default property values.

`rdr = radarDataGenerator(id)` sets the `SensorIndex` property to the specified `id`.

`rdr = radarDataGenerator( ___, scanConfig)` is a convenience syntax that creates a monostatic radar sensor and sets its scanning configuration to a predefined `scanConfig`, in addition



to any input arguments from previous syntaxes. You can specify `scanConfig` as 'No scanning', 'Raster', 'Rotator', or 'Sector'. See “Convenience Syntaxes” on page 4-203 for more details on these configurations.

`rdr = radarDataGenerator( ____, Name, Value)` sets “Properties” on page 4-177 using one or more name-value pairs. Enclose each property name in quotes. For example, `radarDataGenerator('TargetReportFormat','Tracks','FilterInitializationFcn',@initcvkf)` creates a radar sensor that generates track reports using a tracker initialized by a constant-velocity linear Kalman filter.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### Sensor Identification

#### SensorIndex — Unique sensor identifier

0 (default) | positive integer

Unique sensor identifier, specified as a positive integer. Use this property to distinguish between detections or tracks that come from different sensors in a multisensor system. Specify a unique value for each sensor. If you do not update `SensorIndex` from the default value of 0, then the radar returns an error at the start of simulation.

Data Types: double

#### UpdateRate — Sensor update rate (Hz)

1 (default) | positive real scalar

Sensor update rate, in hertz, specified as a positive real scalar. The reciprocal of the update rate must be an integer multiple of the simulation time interval. The radar generates new reports at intervals defined by this reciprocal value. Any sensor update requested between update intervals contains no detections or tracks.

Data Types: double

### Sensor Mounting

#### MountingLocation — Mounting location of radar on platform (m)

[0 0 0] (default) | 1-by-3 real-valued vector

Mounting location of the radar on the platform, in meters, specified as a 1-by-3 real-valued vector of the form  $[x \ y \ z]$ . This property defines the coordinates of the sensor along the x-axis, y-axis, and z-axis relative to the platform body frame.

Data Types: double

#### MountingAngles — Mounting rotation angles of radar (deg)

[0 0 0] (default) | 1-by-3 real-valued vector of form  $[z_{yaw} \ y_{pitch} \ x_{roll}]$

Mounting rotation angles of the radar, in degrees, specified as a 1-by-3 real-valued vector of the form  $[z_{yaw} \ y_{pitch} \ x_{roll}]$ . This property defines the intrinsic Euler angle rotation of the sensor around the z-axis, y-axis, and x-axis with respect to the platform body frame, where:

- $z_{yaw}$ , or yaw angle, rotates the sensor around the z-axis of the platform body frame.
- $y_{pitch}$ , or pitch angle, rotates the sensor around the y-axis of the platform body frame. This rotation is relative to the sensor position that results from the  $z_{yaw}$  rotation.
- $x_{roll}$ , or roll angle, rotates the sensor about the x-axis of the platform body frame. This rotation is relative to the sensor position that results from the  $z_{yaw}$  and  $y_{pitch}$  rotations.

These angles are clockwise-positive when looking in the forward direction of the z-axis, y-axis, and x-axis, respectively.

Data Types: double

### Scanning Settings

#### ScanMode — Scanning mode of radar

'Mechanical' (default) | 'Electronic' | 'Mechanical and electronic' | 'No scanning' | 'Custom'

Scanning mode of the radar, specified as 'Mechanical', 'Electronic', 'Mechanical and electronic', 'No scanning', or 'Custom'.

ScanMode	Purpose
'Mechanical'	The sensor scans mechanically across the azimuth and elevation limits specified by the <code>MechanicalAzimuthLimits</code> and <code>MechanicalElevationLimits</code> properties. The scan direction increments by the radar field of view angle between dwells.
'Electronic'	The sensor scans electronically across the azimuth and elevation limits specified by the <code>ElectronicAzimuthLimits</code> and <code>ElectronicElevationLimits</code> properties. The scan direction increments by the radar field of view angle between dwells.
'Mechanical and electronic'	The sensor mechanically scans the antenna boresight across the mechanical scan limits and electronically scans beams relative to the mechanical angles across the electronic scan limits. The total field of regard scanned in this mode is the combination of the mechanical and electronic scan limits. The scan direction increments by the field of view angle between dwells.
'No scanning'	The sensor beam points along the antenna boresight defined by the <code>MountingAngles</code> property.
'Custom'	The sensor points the beam in the direction specified by the <code>LookAngle</code> property.

Example: 'No scanning'

### **MaxAzimuthScanRate — Maximum mechanical azimuth scan rate (deg/s)**

75 (default) | nonnegative scalar

Maximum mechanical azimuth scan rate, specified as a nonnegative scalar in degrees per second. This property sets the maximum scan rate at which the sensor can mechanically scan in azimuth. The sensor sets its scan rate to step the radar mechanical angle by the field of view. If the required scan rate exceeds the maximum scan rate, the maximum scan rate is used.

#### **Dependencies**

To enable this property, set the ScanMode property to 'Mechanical' or 'Mechanical and electronic'.

Data Types: double

### **MaxElevationScanRate — Maximum mechanical elevation scan rate (deg/s)**

75 (default) | nonnegative scalar

Maximum mechanical elevation scan rate, specified as a nonnegative scalar in degrees per second. The property sets the maximum scan rate at which the sensor can mechanically scan in elevation. The sensor sets its scan rate to step the radar mechanical angle by the field of view. If the required scan rate exceeds the maximum scan rate, the maximum scan rate is used.

#### **Dependencies**

To enable this property, set the ScanMode property to 'Mechanical' or 'Mechanical and electronic'. Also, set the HasElevation property to true.

Data Types: double

### **MechanicalAzimuthLimits — Mechanical azimuth scan limits (deg)**

[0 360] (default) | two-element real-valued vector

Mechanical azimuth scan limits, specified as a two-element real-valued vector of the form [*azMin* *azMax*], where  $azMin \leq azMax$  and  $azMax - azMin \leq 360$ . The limits define the minimum and maximum mechanical azimuth angles, in degrees, the sensor can scan from its mounted orientation.

Example: [-10 20]

#### **Dependencies**

To enable this property, set the ScanMode property to 'Mechanical' or 'Mechanical and electronic'.

Data Types: double

### **MechanicalElevationLimits — Mechanical elevation scan limits (deg)**

[-10 0] (default) | two-element real-valued vector

Mechanical elevation scan limits, specified as a two-element real-valued vector of the form [*elMin* *elMax*], where  $-90 \leq elMin \leq elMax \leq 90$ . The limits define the minimum and maximum mechanical elevation angles, in degrees, the sensor can scan from its mounted orientation.

Example: [-50 20]

**Dependencies**

To enable this property, set the `ScanMode` property to 'Mechanical' or 'Mechanical and electronic'. Also, set the `HasElevation` property to `true`.

Data Types: `double`

**ElectronicAzimuthLimits — Electronic azimuth scan limits (deg)**

`[-45 45]` (default) | two-element real-valued vector

Electronic azimuth scan limits, specified as a two-element real-valued vector of the form `[azMin azMax]`, where  $-90 \leq azMin \leq azMax \leq 90$ . The limits define the minimum and maximum electronic azimuth angles, in degrees, the sensor can scan from its mounted orientation.

Example: `[-50 20]`

**Dependencies**

To enable this property, set the `ScanMode` property to 'Electronic' or 'Mechanical and electronic'.

Data Types: `double`

**ElectronicElevationLimits — Electronic elevation scan limits (deg)**

`[-45 45]` (default) | two-element real-valued vector

Electronic elevation scan limits, specified as a two-element real-valued vector of the form `[elMin elMax]`, where  $-90 \leq elMin \leq elMax \leq 90$ . The limits define the minimum and maximum electronic elevation angles, in degrees, the sensor can scan from its mounted orientation.

Example: `[-50 20]`

**Dependencies**

To enable this property, set the `ScanMode` property to 'Electronic' or 'Mechanical and electronic'. Also, set the `HasElevation` property to `true`.

Data Types: `double`

**MechanicalAngle — Current mechanical scan angle**

two-element real-valued vector

This property is read-only.

Current mechanical scan angle of radar, specified as a two-element real-valued vector of the form `[az el]`. `az` and `el` represent the mechanical azimuth and elevation scan angles, respectively, relative to the mounted angle of the radar on the platform.

Data Types: `double`

**ElectronicAngle — Current electronic scan angle**

two-element real-valued vector

This property is read-only.

Current electronic scan angle of radar, specified as a two-element real-valued vector of the form `[az el]`. `az` and `el` represent the electronic azimuth and elevation scan angles, respectively, relative to the current mechanical angle.

Data Types: double

### LookAngle — Current look angle of sensor

two-element real-valued vector

This property is read-only unless ScanMode is specified as 'Custom'.

Current look angle of the sensor, specified as a two-element real-valued vector of the form  $[az\ el]$ .  $az$  and  $el$  represent the azimuth and elevation look angles, respectively. Look angle is a combination of the mechanical angle and electronic angle, depending on the ScanMode property.

ScanMode	LookAngle
'Mechanical'	MechanicalAngle
'Electronic'	ElectronicAngle
'Mechanical and electronic'	MechanicalAngle + ElectronicAngle
'No scanning'	[0 0]
'Custom'	LookAngle can be set to point the radar to a specific azimuth and elevation.

## Detection Reporting Specifications

### DetectionMode — Detection mode

'Monostatic' (default) | 'ESM' | 'Bistatic'

Detection mode, specified as 'Monostatic', 'ESM', or 'Bistatic'. When set to 'Monostatic', the sensor generates detections from reflected signals originating from a collocated radar emitter. When set to 'ESM', the sensor operates passively and can model ESM and (radar warning receiver) RWR systems. When set to 'Bistatic', the sensor generates detections from reflected signals originating from a separate radar emitter. For more details on detection mode, see “Radar Sensor Detection Modes” on page 4-204.

Example: 'Monostatic'

### HasElevation — Enable radar to scan in elevation and measure target elevation angles

false or 0 (default) | true or 1

Enable the radar to scan in elevation and measure target elevation angles, specified as a logical 0 (false) or 1 (true). Set this property to true to model a radar sensor that can estimate target elevation.

Data Types: logical

### HasRangeRate — Enable radar to measure target range rates

false or 0 (default) | true or 1

Enable the radar to measure target range rates, specified as a logical 0 (false) or 1 (true). Set this property to true to model a radar sensor that can measure range rates from target detections.

Data Types: logical

### HasNoise — Enable addition of noise to radar sensor measurements

true or 1 (default) | false or 0

Enable the addition of noise to radar sensor measurements, specified as a logical 1 (true) or 0 (false). Set this property to true to add noise to the radar measurements. Otherwise, the

measurements have no noise. Even if you set `HasNoise` to `false`, the sensor reports the measurement noise covariance matrix specified in the `MeasurementNoise` property of its object detection outputs.

When the sensor reports tracks, the sensor uses the measurement covariance matrix to estimate the track state and state covariance matrix.

Data Types: `logical`

#### **HasFalseAlarms — Enable creating false alarm radar detections**

`true` or `1` (default) | `false` or `0`

Enable creating false alarm radar measurements, specified as a logical `1` (`true`) or `0` (`false`). Set this property to `true` to report false alarms. Otherwise, the radar reports only actual detections.

Data Types: `logical`

#### **HasOcclusion — Enable occlusion from extended objects**

`true` or `1` (default) | `false` or `0`

Enable occlusion from extended objects, specified as a logical `1` (`true`) or `0` (`false`). Set this property to `true` to model occlusion from extended objects. The sensor models two types of occlusion, self occlusion and inter-object occlusion. Self occlusion occurs when one side of an extended object occludes another side. Inter-object occlusion occurs when one extended object stands in the line of sight of another extended object or a point target. Note that both extended objects and point targets can be occluded by extended objects, but a point target cannot occlude another point target or an extended object.

Data Types: `logical`

#### **HasGhosts — Enable ghost targets in target reports**

`true` or `1` (default) | `false` or `0`

Enable ghost targets in target reports, specified as a logical `1` (`true`) or `0` (`false`). The sensor generates ghost targets for multipath propagation paths up to three reflections between transmission and reception of the radar signal. The sensor only generates ghost targets when the `DetectionMode` property is set to `'Monostatic'`.

Data Types: `logical`

#### **HasRangeAmbiguities — Enable range ambiguities**

`false` or `0` (default) | `true` or `1`

Enable range ambiguities, specified as a logical `0` (`false`) or `1` (`true`). Set this property to `true` to enable sensor range ambiguities. In this case, the sensor does not resolve range ambiguities, and target ranges beyond the `MaxUnambiguousRange` are wrapped into the interval `[0, MaxUnambiguousRange]`. When `false`, the sensor reports targets at their unambiguous range.

Data Types: `logical`

#### **HasRangeRateAmbiguities — Enable range-rate ambiguities**

`false` or `0` (default) | `true` or `1`

Enable range-rate ambiguities, specified as a logical `0` (`false`) or `1` (`true`). Set this property to `true` to enable sensor range-rate ambiguities. When `true`, the sensor does not resolve range rate ambiguities. Target range rates beyond the `MaxUnambiguousRadialSpeed` are wrapped into the

interval `[0, MaxUnambiguousRadialSpeed]`. When `false`, the sensor reports targets at their unambiguous range rates.

#### Dependencies

To enable this property, set the `HasRangeRate` property to `true`.

Data Types: `logical`

#### HasINS — Enable inertial navigation system (INS) input

`false` or `0` (default) | `true` or `1`

Enable the INS input argument, which passes the current estimate of the sensor platform pose to the sensor, specified as a logical `0` (`false`) or `1` (`true`). When `true`, pose information is added to the `MeasurementParameters` structure of the reported detections or the `StateParameters` structure of the reported tracks, based on the `TargetReportFormat` property. Pose information enables tracking and fusion algorithms to estimate the state of the target in the scenario frame.

Data Types: `logical`

#### HasScanLoss — Enable losses due to electronic scanning off-broadside

`false` (default) | `true`

Enable scan loss, specified as `false` or `true`. Scan loss models the effect of antenna array beam broadening when the radar points to an off-broadside angle.

#### Dependencies

To enable this property, set the `ScanMode` property to `'Custom'`.

Data Types: `logical`

#### MaxNumReportsSource — Source of maximum for number of detection or track reports

`'Auto'` (default) | `'Property'`

Source of the maximum for the number of detection or track reports, specified as one of these options:

- `'Auto'` — The sensor reports all detections or tracks.
- `'Property'` — The sensor reports the first  $N$  valid detections or tracks, where  $N$  is equal to the `MaxNumReports` property value.

#### MaxNumReports — Maximum number of detection or track reports

`100` (default) | positive integer

Maximum number of detection or track reports, specified as a positive integer. The sensor reports detections, in order of increasing distance from the sensor, until reaching this maximum number.

#### Dependencies

To enable this property, set the `MaxNumReportsSource` property to `'Property'`.

Data Types: `double`

#### TargetReportFormat — Format of generated target reports

`'Clustered detections'` (default) | `'Tracks'` | `'Detections'`

Format of generated target reports, specified as one of these options:

- `'Clustered detections'` — The sensor generates target reports as clustered detections, where each target is reported as a single detection that is the centroid of the unclustered target detections. The sensor returns clustered detections as a cell array of `objectDetection` objects. To enable this option, set the `DetectionMode` property to `'Monostatic'` and set the `EmissionsInputPort` property to `false`.
- `'Tracks'` — The sensor generates target reports as tracks, which are clustered detections that have been processed by a tracking filter. The sensor returns tracks as an array of `objectTrack` objects. To enable this option, set the `DetectionMode` property to `'Monostatic'` and set the `EmissionsInputPort` property to `false`.
- `'Detections'` — The sensor generates target reports as unclustered detections, where each target can have multiple detections. The sensor returns unclustered detections as a cell array of `objectDetection` objects.

### DetectionCoordinates — Coordinate system used to report detections

`'Body'` | `'Scenario'` | `'Sensor rectangular'` | `'Sensor spherical'`

Coordinate system used to report detections, specified as one of these options:

- `'Scenario'` — Detections are reported in the rectangular scenario coordinate frame. The scenario coordinate system is defined as the local navigation frame at simulation start time. To enable this value, set the `HasINS` property to `true`.
- `'Body'` — Detections are reported in the rectangular body system of the sensor platform.
- `'Sensor rectangular'` — Detections are reported in the sensor rectangular body coordinate system.
- `'Sensor spherical'` — Detections are reported in a spherical coordinate system derived from the sensor rectangular body coordinate system. This coordinate system is centered at the sensor and aligned with the orientation of the radar on the platform.

When the `DetectionMode` property is set to `'Monostatic'`, you can specify the `DetectionCoordinates` as `'Body'` (default for `'Monostatic'`), `'Scenario'`, `'Sensor rectangular'`, or `'Sensor spherical'`. When the `DetectionMode` property is set to `'ESM'` or `'Bistatic'`, the default value of the `DetectionCoordinates` property is `'Sensor spherical'`, which cannot be changed.

Example: `'Sensor spherical'`

### Measurement Resolution and Bias

#### AzimuthResolution — Azimuth resolution of radar (deg)

1 (default) | positive real scalar

Azimuth resolution of the radar, in degrees, specified as a positive scalar. The azimuth resolution defines the minimum separation in azimuth angle at which the radar can distinguish between two targets. The azimuth resolution is typically the half-power beamwidth of the azimuth angle beamwidth of the radar.

Data Types: `double`

#### ElevationResolution — Elevation resolution of radar (deg)

5 (default) | positive real scalar

Elevation resolution of the radar, in degrees, specified as a positive real scalar. The elevation resolution defines the minimum separation in elevation angle at which the radar can distinguish



between two targets. The elevation resolution is typically the half-power beamwidth of the elevation angle beamwidth of the radar.

**Dependencies**

To enable this property, set the `HasElevation` property to `true`.

Data Types: `double`

**RangeResolution — Range resolution of radar (m)**

100 (default) | positive real scalar

Range resolution of the radar, in meters, specified as a positive real scalar. The range resolution defines the minimum separation in range at which the radar can distinguish between two targets.

Data Types: `double`

**RangeRateResolution — Range-rate resolution of radar (m/s)**

10 (default) | positive real scalar

Range-rate resolution of the radar, in meters per second, specified as a positive real scalar. The range rate resolution defines the minimum separation in range rate at which the radar can distinguish between two targets.

**Dependencies**

To enable this property, set the `HasRangeRate` property to `true`.

Data Types: `double`

**AzimuthBiasFraction — Azimuth bias fraction of radar**

0.1 (default) | nonnegative scalar

Azimuth bias fraction of the radar, specified as a nonnegative scalar. Azimuth bias is expressed as a fraction of the azimuth resolution specified in the `AzimuthResolution` property. This value sets a lower bound on the azimuthal accuracy of the radar and is dimensionless.

Data Types: `double`

**ElevationBiasFraction — Elevation bias fraction of radar**

0.1 (default) | nonnegative scalar

Elevation bias fraction of the radar, specified as a nonnegative scalar. Elevation bias is expressed as a fraction of the elevation resolution specified by the `ElevationResolution` property. This value sets a lower bound on the elevation accuracy of the radar and is dimensionless.

**Dependencies**

To enable this property, set the `HasElevation` property to `true`.

Data Types: `double`

**RangeBiasFraction — Range bias fraction**

0.05 (default) | nonnegative scalar

Range bias fraction of the radar, specified as a nonnegative scalar. Range bias is expressed as a fraction of the range resolution specified by the `RangeResolution` property. This property sets a lower bound on the range accuracy of the radar and is dimensionless.

Data Types: double

**RangeRateBiasFraction — Range-rate bias fraction**

0.05 (default) | nonnegative scalar

Range-rate bias fraction of the radar, specified as a nonnegative scalar. Range-rate bias is expressed as a fraction of the range-rate resolution specified by the `RangeRateResolution` property. This property sets a lower bound on the range rate accuracy of the radar and is dimensionless.

**Dependencies**

To enable this property, set the `HasRangeRate` property to `true`.

Data Types: double

**Detection Settings****CenterFrequency — Center frequency of radar band (Hz)**

300e6 (default) | positive real scalar

Center frequency of the radar band, in hertz, specified as a positive real scalar.

Data Types: double

**Bandwidth — Radar waveform bandwidth**

3e6 (default) | positive real scalar

Radar waveform bandwidth, in hertz, specified as a positive real scalar.

Example: 100e3

Data Types: double

**WaveformTypes — Types of detectable waveforms**

0 (default) |  $L$ -element vector of nonnegative integers

Types of detectable waveforms, specified as an  $L$ -element vector of nonnegative integers. Each integer represents a type of waveform detectable by the radar.

Example: [1 4 5]

Data Types: double

**ConfusionMatrix — Probability of correct classification of detected waveform**

1 (default) | positive scalar |  $L$ -element vector of nonnegative real values |  $L$ -by- $L$  matrix of nonnegative real values

Probability of correct classification of a detected waveform, specified as a positive scalar, an  $L$ -element vector of nonnegative real values, or an  $L$ -by- $L$  matrix of nonnegative real values, where  $L$  is the number of waveform types detectable by the sensor, as indicated by the value set in the `WaveformTypes` property. Matrix values must be in the range  $[0, 1]$ .

The  $(i, j)$  matrix element represents the probability of classifying the  $i$ th waveform as the  $j$ th waveform. When you specify this property as a scalar from 0 through 1, the value is expanded along the diagonal of the confusion matrix. When specified as a vector, the vector is aligned as the diagonal of the confusion matrix. When defined as a scalar or a vector, the off-diagonal values are set to  $(1 - val)/(L - 1)$ , where  $val$  is the value of the diagonal element.

Data Types: double

**Sensitivity — Minimum operational sensitivity of receiver**

-50 (default) | scalar

Minimum operational sensitivity of receiver, specified as a scalar. Sensitivity includes isotropic antenna receiver gain. Units are in dBmi.

Example: -10

Data Types: double

**DetectionThreshold — Minimum SNR required to declare detection**

5 (default) | scalar

Minimum signal-to-noise ratio (SNR) required to declare a detection, specified as a scalar. Units are in dB.

Example: -1

Data Types: double

**DetectionProbability — Probability of detecting target**

0.9 (default) | scalar in range (0, 1]

Probability of detecting a target, specified as a scalar in the range (0, 1]. This property defines the probability of detecting a target with a radar cross-section (RCS), ReferenceRCS, at the reference detection range, ReferenceRange.

**Tunable:** Yes

Data Types: double

**ReferenceRange — Reference range for given probability of detection (m)**

100e3 (default) | positive real scalar

Reference range for the given probability of detection and the given reference radar cross-section (RCS), in meters, specified as a positive real scalar. The reference range is the range, at which a target having a radar cross-section specified by the ReferenceRCS property is detected with a probability of detection specified by the DetectionProbability property.

**Tunable:** Yes

Data Types: double

**ReferenceRCS — Reference radar cross-section for given probability of detection (dBsm)**

0 (default) | real scalar

Reference radar cross-section (RCS) for a given probability of detection and reference range, in decibel square meters, specified as a real scalar. The reference RCS is the RCS value at which a target is detected with a probability specified by DetectionProbability at the specified ReferenceRange value.

**Tunable:** Yes

Data Types: double

**FalseAlarmRate — False alarm report rate**1e-6 (default) | positive real scalar in range  $[10^{-7}, 10^{-3}]$

False alarm report rate within each radar resolution cell, specified as a positive real scalar in the range  $[10^{-7}, 10^{-3}]$ . Units are dimensionless. The object determines resolution cells from the `AzimuthResolution` and `RangeResolution` properties and, when enabled, from the `ElevationResolution` and `RangeRateResolution` properties.

**Tunable:** Yes

Data Types: double

### **FieldOfView — Angular field of view of radar (deg)**

[1 5] | 1-by-2 positive real-valued vector

Angular field of view of the radar, in degrees, specified as a 1-by-2 positive real-valued vector of the form  $[azfov\ elfov]$ . The field of view defines the total angular extent spanned by the sensor. The azimuth field of view, *azfov*, must be in the range (0, 360]. The elevation field of view, *elfov*, must be in the range (0, 180].

Data Types: double

### **RangeLimits — Minimum and maximum range of radar (m)**

[0 100e3] (default) | 1-by-2 nonnegative real-valued vector

Minimum and maximum range of radar, in meters, specified as a 1-by-2 nonnegative real-valued vector of the form  $[\min, \max]$ . The radar does not detect targets that are outside this range. The maximum range, *max*, must be greater than the minimum range, *min*.

### **RangeRateLimits — Minimum and maximum range rate of radar (m/s)**

[-200 200] (default) | 1-by-2 real-valued vector

Minimum and maximum range rate of radar, in meters per second, specified as a 1-by-2 real-valued vector of the form  $[\min, \max]$ . The radar does not detect targets that are outside this range rate. The maximum range rate, *max*, must be greater than the minimum range rate, *min*.

### **Dependencies**

To enable this property, set the `HasRangeRate` property to `true`.

### **MaxUnambiguousRange — Maximum unambiguous detection range**

100e3 (default) | positive scalar

Maximum unambiguous detection range, specified as a positive scalar in meters. Maximum unambiguous range defines the maximum range for which the radar can unambiguously resolve the range of a target. When `HasRangeAmbiguities` is set to `true`, targets detected at ranges beyond the maximum unambiguous range are wrapped into the range interval  $[0, \text{MaxUnambiguousRange}]$ .

This property also applies to false target detections when you set the `HasFalseAlarms` property to `true`. In this case, the property defines the maximum range at which false alarms can be generated.

Example: `5e3`

### **Dependencies**

To enable this property, set the `HasRangeAmbiguities` property to `true`.

Data Types: double

**MaxUnambiguousRadialSpeed — Maximum unambiguous radial speed**

200 (default) | positive scalar

Maximum unambiguous radial speed, specified as a positive scalar in meters per second. Radial speed is the magnitude of the target range rate. Maximum unambiguous radial speed defines the radial speed for which the radar can unambiguously resolve the range rate of a target. When `HasRangeRateAmbiguities` is set to `true`, targets detected at range rates beyond the maximum unambiguous radial speed are wrapped into the range rate interval  $[-\text{MaxUnambiguousRadialSpeed}, \text{MaxUnambiguousRadialSpeed}]$ .

This property also applies to false target detections obtained when you set both the `HasRangeRate` and `HasFalseAlarms` properties to `true`. In this case, the property defines the maximum radial speed at which false alarms can be generated.

**Dependencies**

To enable this property, set `HasRangeRate` and `HasRangeRateAmbiguities` to `true`.

Data Types: `double`

**RadarLoopGain — Radar loop gain**

real scalar

This property is read-only.

Radar loop gain, specified as a real scalar. `RadarLoopGain` depends on the values of the `DetectionProbability`, `ReferenceRange`, `ReferenceRCS`, and `FalseAlarmRate` properties. Radar loop gain is a function of the reported signal-to-noise ratio of the radar,  $SNR$ , the target radar cross-section,  $RCS$ , and the target range,  $R$ , as described by this equation:

$$SNR = \text{RadarLoopGain} + RCS - 40\log_{10}(R)$$

$SNR$  and  $RCS$  are in decibels and decibel square meters, respectively,  $R$  is in meters, and `RadarLoopGain` is in decibels.

Data Types: `double`

**Interference and Emission Inputs****InterferenceInputPort — Enable interference input**

false or 0 (default) | true or 1

Enable interference input, specified as a logical 0 (false) or 1 (true). Set this property to `true` to enable interference input when running the radar.

**Dependencies**

To enable this property, set `DetectionMode` to 'Monostatic' and set `EmissionsInputPort` to `false`.

Data Types: `logical`

**EmissionsInputPort — Enable emissions input**

false or 0 (default) | true or 1

Enable emissions input, specified as a logical 0 (false) or 1 (true). Set this property to `true` to enable emissions input when running the radar.

**Dependencies**

To enable this property, set `DetectionMode` to 'Monostatic' and set `InterferenceInputPort` to `false`.

Data Types: `logical`

**EmitterIndex — Unique identifier of monostatic emitter**

1 (default) | positive integer

Unique identifier of the monostatic emitter, specified as a positive integer. Use this index to identify the monostatic emitter providing the reference emission for the radar.

**Dependencies**

To enable this property, set `DetectionMode` to 'Monostatic' and set `EmissionsInputPort` to `true`.

Data Types: `double`

**Tracking Settings****FilterInitializationFcn — Kalman filter initialization function**

@initcvekf (default) | function handle | character vector | string scalar

Kalman filter initialization function, specified as a function handle or as a character vector or string scalar of the name of a valid Kalman filter initialization function.

The table shows the initialization functions that you can use to specify `FilterInitializationFcn`.

Initialization Function	Function Definition
<code>initcaabf</code>	Initialize constant-acceleration alpha-beta Kalman filter
<code>initcvabf</code>	Initialize constant-velocity alpha-beta Kalman filter
<code>initcakf</code>	Initialize constant-acceleration linear Kalman filter.
<code>initcvkf</code>	Initialize constant-velocity linear Kalman filter.
<code>initcaekf</code>	Initialize constant-acceleration extended Kalman filter.
<code>initctekf</code>	Initialize constant-turnrate extended Kalman filter.
<code>initcvekf</code>	Initialize constant-velocity extended Kalman filter.
<code>initcaukf</code>	Initialize constant-acceleration unscented Kalman filter.
<code>initctukf</code>	Initialize constant-turnrate unscented Kalman filter.
<code>initcvukf</code>	Initialize constant-velocity unscented Kalman filter.

You can also write your own initialization function. The function must have the following syntax:

```
filter = filterInitializationFcn(detection)
```

The input to this function is a detection report like those created by an `objectDetection` object. The output of this function must be a tracking filter object, such as `trackingKF`, `trackingEKF`, `trackingUKF`, or `trackingABF`.

To guide you in writing this function, you can examine the details of the supplied functions from within MATLAB. For example:

```
type initcvekf
```

### Dependencies

To enable this property, set the `TargetReportFormat` property to 'Tracks'.

Data Types: `function_handle` | `char` | `string`

### ConfirmationThreshold — Threshold for track confirmation

[2 3] (default) | 1-by-2 vector of positive integers

Threshold for track confirmation, specified as a 1-by-2 vector of positive integers of the form [M N]. A track is confirmed if it receives at least M detections in the last N updates. M must be less than or equal to N.

- When setting M, take into account the probability of object detection for the sensors. The probability of detection depends on factors such as occlusion or clutter. You can reduce M when tracks fail to be confirmed or increase M when too many false detections are assigned to tracks.
- When setting N, consider the number of times you want the tracker to update before it makes a confirmation decision. For example, if a tracker updates every 0.05 seconds, and you want to allow 0.5 seconds to make a confirmation decision, set N = 10.

Example: [3 5]

### Dependencies

To enable this property, set the `TargetReportFormat` property to 'Tracks'.

Data Types: `double`

### DeletionThreshold — Threshold for track deletion

[5 5] (default) | 1-by-2 vector of positive integers

Threshold for track deletion, specified as a 1-by-2 vector of positive integers of the form [P R]. If a confirmed track is not assigned to any detection P times in the last R tracker updates, then the track is deleted. P must be less than or equal to R.

- To reduce how long the radar maintains tracks, decrease R or increase P.
- To maintain tracks for a longer time, increase R or decrease P.

Example: [3 5]

### Dependencies

To enable this property, set the `TargetReportFormat` property to 'Tracks'.

Data Types: `double`

**TrackCoordinates — Coordinate system of reported tracks**

'Scenario' | 'Body' | 'Sensor'

Coordinate system used to report tracks, specified as one of these options:

- 'Scenario' — Tracks are reported in the rectangular scenario coordinate frame. The scenario coordinate system is defined as the local navigation frame at simulation start time. To enable this option, set the "HasINS" on page 4-0 property to `true`.
- 'Body' — Tracks are reported in the rectangular body system of the sensor platform.
- 'Sensor' — Tracks are reported in the sensor rectangular body coordinate system.

**Dependencies**

To enable this property, set the `TargetReportFormat` property to 'Tracks'.

**Target Profiles****Profiles — Physical characteristics of target platforms**

structure | array of structures

Physical characteristics of target platforms, specified as a structure or an array of structures. Unspecified fields take default values.

- If you specify the property as a structure, then the structure applies to all target platforms.
- If you specify the property as an array of structures, then each structure in the array applies to the corresponding target platform based on the `PlatformID` field. In this case, you must specify each `PlatformID` field as a positive integer and must not leave the field as empty.

Field	Description	Default Value
<code>PlatformID</code>	Scenario-defined platform identifier, defined as a positive integer.	empty
<code>ClassID</code>	User-defined platform classification identifier, defined as a nonnegative integer.	0
<code>Dimensions</code>	Platform dimensions, defined as a structure with these fields: <ul style="list-style-type: none"> <li>• Length</li> <li>• Width</li> <li>• Height</li> <li>• OriginOffset</li> </ul>	0
<code>Signatures</code>	Platform signatures, defined as a cell array containing an <code>rccSignature</code> object, which specifies the RCS signature of the platform.	The default <code>rccSignature</code> object

See `Platform` for more details on these fields.

Data Types: `struct`



## Usage

### Syntax

```
reports = rdr(targetPoses, simTime)
reports = rdr(targetPoses, interferences, simTime)
reports = rdr(emissions, emitterConfigs, simTime)
```

```
reports = rdr(emissions, simTime)
```

```
reports = rdr( ____, insPose, simTime)
```

```
[reports, numReports, config] = rdr( ____ )
```

### Description

#### Monostatic Detection Mode

These syntaxes apply when you set the `DetectionMode` property to 'Monostatic'.

`reports = rdr(targetPoses, simTime)` returns monostatic target reports from the target poses, `targetPoses`, at the current simulation time, `simTime`. The object can generate reports for multiple targets. To enable this syntax:

- Set the `DetectionMode` property to 'Monostatic'.
- Set the `InterferenceInputPort` property to false.
- Set the `EmissionsInputPort` property to false.

`reports = rdr(targetPoses, interferences, simTime)` specifies the interference signals, `interferences`, in the radar signal transmission. To enable this syntax:

- Set the `DetectionMode` property to 'Monostatic'.
- Set the `InterferenceInputPort` property to true.
- Set the `EmissionsInputPort` property to false.

`reports = rdr(emissions, emitterConfigs, simTime)` returns monostatic target reports based on the emission signal, `emissions`, and the configurations of the corresponding emitters, `emitterConfigs`, that generate the emissions. To enable this syntax:

- Set the `DetectionMode` property to 'Monostatic'.
- Set the `InterferenceInputPort` property to false.
- Set the `EmissionsInputPort` property to true.

#### Bistatic or ESM Detection Mode

This syntax applies when you set the `DetectionMode` property to 'Bistatic' or 'ESM'. In these two modes, the `TargetReportFormat` can only be 'Detections' and the `DetectionCoordinates` can only be 'Sensor spherical'.

`reports = rdr(emissions, simTime)` returns Bistatic or ESM reports from the radar signal emissions at the simulation time, `simTime`.

**Provide INS Input**

This syntax applies when you set the `HasINS` property to `true`.

`reports = rdr( ____, insPose, simTime)` specifies the pose information of the radar platform through an INS estimate. The `insPose` argument is the second to the last argument before the `simTime` argument. This syntax can be used with any of the previous syntaxes. See the “HasINS” on page 4-0 property for more details.

**Output Additional Information**

Use this syntax if you want to output additional information of the reports.

`[reports, numReports, config] = rdr( ____, )` returns the number of reports, `numReports`, and the configuration of the radar, `config`, at the current simulation time.

**Input Arguments****targetPoses — Target poses**

array of structures

Radar scenario target poses, specified as an array of structures. Each structure corresponds to a target. You can generate the structure using the `targetPoses` object function of a platform. You can also create such a structure manually. This table shows the fields of the structure:

Field	Description
PlatformID	Unique identifier for the platform, specified as a positive integer. This is a required field with no default value.
ClassID	User-defined integer used to classify the type of target, specified as a nonnegative integer. 0 is reserved for unclassified platform types and is the default value.
Position	Position of the target in platform coordinates, specified as a real-valued, 1-by-3 vector. This is a required field with no default value. Units are in meters.
Velocity	Velocity of the target in platform coordinates, specified as a real-valued, 1-by-3 vector. Units are in meters per second. The default is <code>[0 0 0]</code> .
Acceleration	Acceleration of the target in platform coordinates specified as a 1-by-3 row vector. Units are in meters per second-squared. The default is <code>[0 0 0]</code> .
Orientation	Orientation of the target with respect to platform coordinates, specified as a scalar quaternion or a 3-by-3 rotation matrix. Orientation defines the frame rotation from the platform coordinate system to the current target body coordinate system. Units are dimensionless. The default is <code>quaternion(1,0,0,0)</code> .

Field	Description
AngularVelocity	Angular velocity of the target in platform coordinates, specified as a real-valued, 1-by-3 vector. The magnitude of the vector defines the angular speed. The direction defines the axis of clockwise rotation. Units are in degrees per second. The default is [0 0 0].

The values of the `Position`, `Velocity`, and `Orientation` fields are defined with respect to the platform body frame.

If the dimensions of the target or RCS signature change with respect to time, you can specify these two additional fields in the structure:

Field	Description
Dimensions	Platform dimensions, specified as a structure with these fields: <ul style="list-style-type: none"> <li>• Length</li> <li>• Width</li> <li>• Height</li> <li>• OriginOffset</li> </ul>
Signatures	Platform signatures, specified as a cell array containing an <code>rccSignature</code> object, which specifies the RCS signature of the platform.

If the dimensions of the target and RCS signature remain static with respect to time, you can specify its dimensions and RCS signature using the `Profiles` property.

#### **interferences — Interference radar emissions**

array of `radarEmission` objects | cell array of `radarEmission` objects | array of structure

Interference radar emissions, specified as an array or cell array of `radarEmission` objects. You can also specify `interferences` as an array of structures with field names corresponding to the property names of the `radarEmission` object.

#### **emissions — Radar emissions**

array of `radarEmission` objects | cell array of `radarEmission` objects | array of structures

Radar emissions, specified as an array or cell array of `radarEmission` objects. You can also specify `emissions` as an array of structures with field names corresponding to the property names of the `radarEmission` object.

#### **emitterConfigs — Emitter configurations**

array of structures

Emitter configurations, specified as an array of structures. This array must contain the configuration of the radar emitter whose `EmitterIndex` matches the value of the `EmitterIndex` property of the `radarDataGenerator`. Each structure has these fields:

Field	Description
-------	-------------

EmitterIndex	Unique emitter index.
IsValidTime	Valid emission time, returned as 0 or 1. The value of IsValidTime is 0 when emitter updates are requested at times that are between update intervals specified by UpdateInterval.
IsScanDone	IsScanDone is true when the emitter has completed a scan.
FieldOfView	Field of view of the emitter.
MeasurementParameters	MeasurementParameters is an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.

For more details on MeasurementParameters, see “Measurement Parameters” on page 4-205.

Data Types: struct

### **insPose — Platform pose from INS**

structure

Platform pose information from an inertial navigation system (INS), specified as a structure with these fields:

Field	Definition
Position	Position in the scenario frame, specified as a real-valued 1-by-3 vector. Units are in meters.
Velocity	Velocity in the scenario frame, specified as a real-valued 1-by-3 vector. Units are in meters per second.
Orientation	Orientation with respect to the scenario frame, specified as a quaternion or a 3-by-3 real-valued rotation matrix. The rotation is from the navigation frame to the current INS body frame. This is also referred to as a “parent to child” rotation.

### **simTime — Current simulation time**

nonnegative scalar

Current simulation time, specified as a nonnegative scalar. The radarScenario object calls the scan radar sensor at regular time intervals. The sensor only generates reports at simulation times corresponding to integer multiples of the update interval, which is given by the reciprocal of the UpdateRate property.

- When called at these intervals, targets are reported in reports, the number of reports is returned in numReports, and the IsValidTime field of the returned config structure is returned as true.
- When called at all other simulation times, the sensor returns an empty report, numReports is returned as 0, and the IsValidTime field of the returned config structure is returned as false.

Example: 10.5

Data Types: double

### Output Arguments

#### reports — Detection and track reports

cell array of objectDetection objects | cell array of objectTrack objects

Detection and track reports, returned as one of these options:

- A cell array of objectDetection objects, when the TargetReportFormat property is set to 'Detections' or 'Clustered detections'. Additionally, when the DetectionMode is set to 'ESM' or 'Bistatic', the sensor can only generate unclustered detections and cannot generate clustered detections.
- A cell array of objectTrack objects, when the TargetReportFormat property is set to 'Tracks'. The sensor can only output tracks when the DetectionMode is set to 'Monostatic'. The sensor returns only confirmed tracks, which are tracks that satisfy the confirmation threshold specified in the ConfirmationThreshold property. For these tracks, the IsConfirmed property of the object is true.

In generated code, reports return as equivalent structures with field names corresponding to the property names of the objectDetection object or the property names of the objectTrack objects, based on the TargetReportFormat property.

The format and coordinates of the measurement states or track states is determined by the specifications of the HasRangeRate, HasElevation, HasINS, TargetReportFormat, and DetectionCoordinates properties. For more details, see “Detection and Track State Coordinates” on page 4-205.

#### numReports — Number of reported detections or tracks

nonnegative integer

Number of reported detections or tracks, returned as a nonnegative integer. numReports is equal to the length of the reports argument.

Data Types: double

#### config — Current sensor configuration

structure

Current sensor configuration, specified as a structure. This output can be used to determine which objects fall within the radar beam during object execution.

Field	Description
SensorIndex	Unique sensor index, returned as a positive integer.
IsValidTime	Valid detection time, returned as true or false. IsValidTime is false when detection updates are requested between update intervals specified by the update rate.
IsScanDone	IsScanDone is true when the sensor has completed a scan.

FieldOfView	Field of view of the sensor, returned as a 2-by-1 vector of positive real values, [azfov;elfov]. azfov and elfov represent the field of view in azimuth and elevation, respectively.
RangeLimits	Minimum and maximum range of sensor, in meters, specified as a 1-by-2 nonnegative real-valued vector of the form [rmin,rmax].
RangeRateLimits	Minimum and maximum range rate of sensor, in meters per second, specified as a 1-by-2 real-valued vector of the form [rrmin,rrmax].
MeasurementParameters	Sensor measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current sensor frame.

Data Types: struct

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named obj, use this syntax:

```
release(obj)
```

### Specific to radarDataGenerator

coverageConfig    Sensor and emitter coverage configuration  
radarTransceiver    Create corresponding radar transceiver from radarDataGenerator  
perturb            Apply perturbations to object  
perturbations      Perturbation defined on object

### Common to All System Objects

step            Run System object algorithm  
release        Release resources and allow changes to System object property values and input characteristics  
reset         Reset internal states of System object

## Examples

### Model Air Traffic Control Tower Scanning

Create three targets by specifying their platform ID, position, and velocity.

```
tgt1 = struct('PlatformID',1, ...
    'Position',[0 -50e3 -1e3], ...
    'Velocity',[0 900*1e3/3600 0]);
```

```
tgt2 = struct('PlatformID',2, ...
    'Position',[20e3 0 -500], ...
```

```

    'Velocity',[700*1e3/3600 0 0]);

tgt3 = struct('PlatformID',3, ...
    'Position',[-20e3 0 -500], ...
    'Velocity',[300*1e3/3600 0 0]);

```

Create an airport surveillance radar that is 15 meters above the ground.

```

rpm = 12.5;
fov = [1.4; 5]; % [azimuth; elevation]
scanrate = rpm*360/60; % deg/s
updaterate = scanrate/fov(1); % Hz

sensor = radarDataGenerator(1,'Rotator', ...
    'UpdateRate',updaterate, ...
    'MountingLocation',[0 0 -15], ...
    'MaxAzimuthScanRate',scanrate, ...
    'FieldOfView',fov, ...
    'AzimuthResolution',fov(1));

```

Generate detections from a full scan of the radar.

```

simTime = 0;
detBuffer = {};
while true
    [dets,numDets,config] = sensor([tgt1 tgt2 tgt3],simTime);
    detBuffer = [detBuffer; dets]; %#ok<AGROW>

    % Is full scan complete?
    if config.IsScanDone
        break % yes
    end
    simTime = simTime + 1/sensor.UpdateRate;
end

```

```

radarPosition = [0 0 0];
tgtPositions = [tgt1.Position; tgt2.Position; tgt3.Position];

```

Visualize the results.

```

clrs = lines(3);

figure
hold on

% Plot radar position
plot3(radarPosition(1),radarPosition(2),radarPosition(3),'Marker','s', ...
    'DisplayName','Radar','MarkerFaceColor',clrs(1,:),'LineStyle','none')

% Plot truth
plot3(tgtPositions(:,1),tgtPositions(:,2),tgtPositions(:,3),'Marker','^', ...
    'DisplayName','Truth','MarkerFaceColor',clrs(2,:),'LineStyle','none')

% Plot detections
if ~isempty(detBuffer)
    detPos = cellfun(@(d)d.Measurement(1:3),detBuffer, ...
        'UniformOutput',false);
    detPos = cell2mat(detPos)';
end

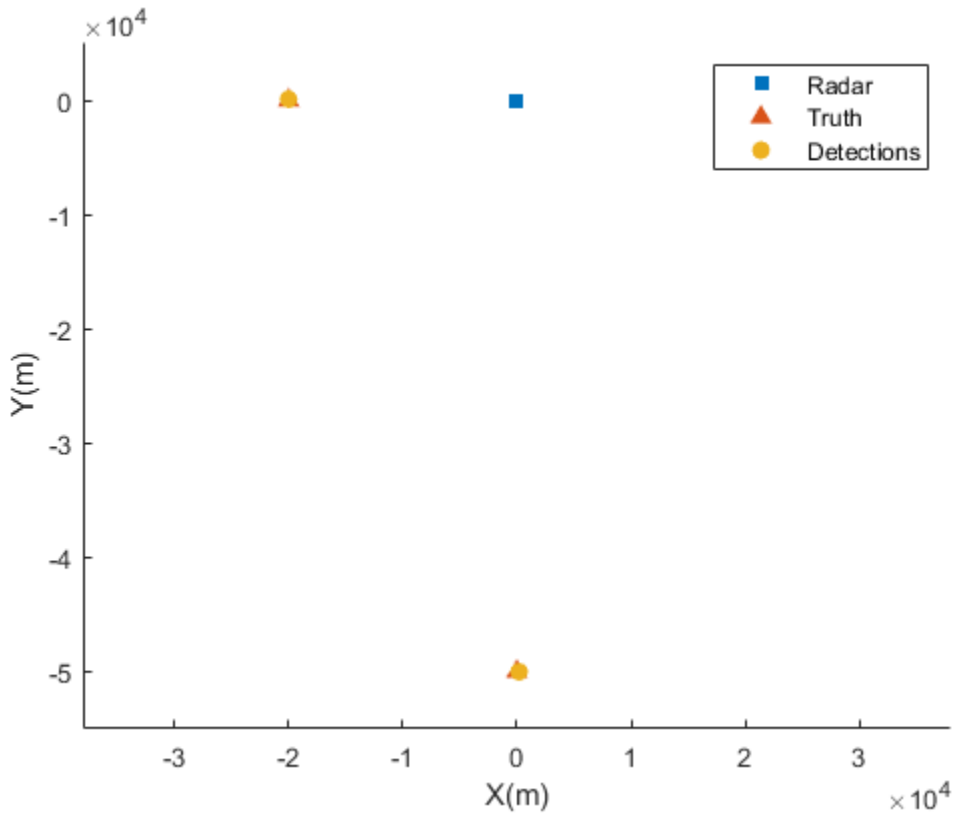
```

```

    plot3(detPos(:,1),detPos(:,2),detPos(:,3),'Marker','o', ...
          'DisplayName','Detections','MarkerFaceColor',clrs(3,:),'LineStyle','none')
end

xlabel('X(m)')
ylabel('Y(m)')
axis('equal')
legend

```



### Detect Radar Emission with radarDataGenerator

Create a radar emission and then detect the emission using a radarDataGenerator object.

First, create a radar emission.

```

orient = quaternion([180 0 0], 'eulerd', 'zyx', 'frame');
rfSig = radarEmission('PlatformID',1,'EmitterIndex',1,'EIRP',100, ...
  'OriginPosition',[30 0 0],'Orientation',orient);

```

Then, create an ESM sensor using radarDataGenerator.

```

sensor = radarDataGenerator(1,'DetectionMode','ESM');

```

Detect the RF emission.



```

time = 0;
[dets,numDets,config] = sensor(rfSig,time)

dets = 1x1 cell array
      {1x1 objectDetection}

numDets = 1

config = struct with fields:
    SensorIndex: 1
    IsValidTime: 1
    IsScanDone: 0
    FieldOfView: [1 5]
    RangeLimits: [0 Inf]
    RangeRateLimits: [0 Inf]
    MeasurementParameters: [1x1 struct]

```

### Point Radar at Target

Create a radar that can be pointed directly at targets of interest to generate statistical detections. This setup is useful in cases where the azimuth and elevation of the target are already estimated by a tracker. Thus the radar can be cued to detect the target to update the track in between surveillance updates and other target track updates. To specify such a radar, set the `ScanMode` property of `radarDataGenerator` to "Custom".

```
rdr = radarDataGenerator(1, 'ScanMode', 'Custom', 'HasElevation', true)
```

```

rdr =
  radarDataGenerator with properties:

    SensorIndex: 1
    UpdateRate: 1
    DetectionMode: 'Monostatic'
    ScanMode: 'Custom'
    InterferenceInputPort: 0

    MountingLocation: [0 0 0]
    MountingAngles: [0 0 0]

    FieldOfView: [1 5]
    RangeLimits: [0 100000]

    DetectionProbability: 0.9000
    FalseAlarmRate: 1.0000e-06
    ReferenceRange: 100000

    TargetReportFormat: 'Clustered detections'

  Show all properties

```

Create a target at which to point the radar. The target is located at a range of 1 km from the radar at an azimuth of 10 degrees and an elevation of 5 degrees.

```
tgtRg = 1e3;
tgtAz = 10;
tgtEl = 5;
```

```
[X,Y,Z] = sph2cart(deg2rad(tgtAz),deg2rad(tgtEl),tgtRg);
```

```
tgt = struct(PlatformID=1,Position=[X Y Z]);
```

Point the radar directly at the target. Generate the statistical detection.

```
rdr.LookAngle = [tgtAz tgtEl];
```

```
simTime = 0;
dets = rdr(tgt,simTime);
```

Compare the measured target location to the actual position.

```
detpos = dets{1}.Measurement;
```

```
tbt = table(detpos,tgt.Position', ...
    RowNames=["X" "Y" "Z"],VariableNames=["Measured" "Actual"])
```

```
tbt=3x2 table
      Measured   Actual
      _____   _____
      X      968.64    981.06
      Y      171.74    172.99
      Z      101.94     87.156
```

Create a theaterPlot object. Plot the radar, the target, and the radar detections. Overlay a plot of the radar coverage.

```
tp = theaterPlot(AxesUnits=["m" "m" "m"],XLimits=[0 2e3]);
```

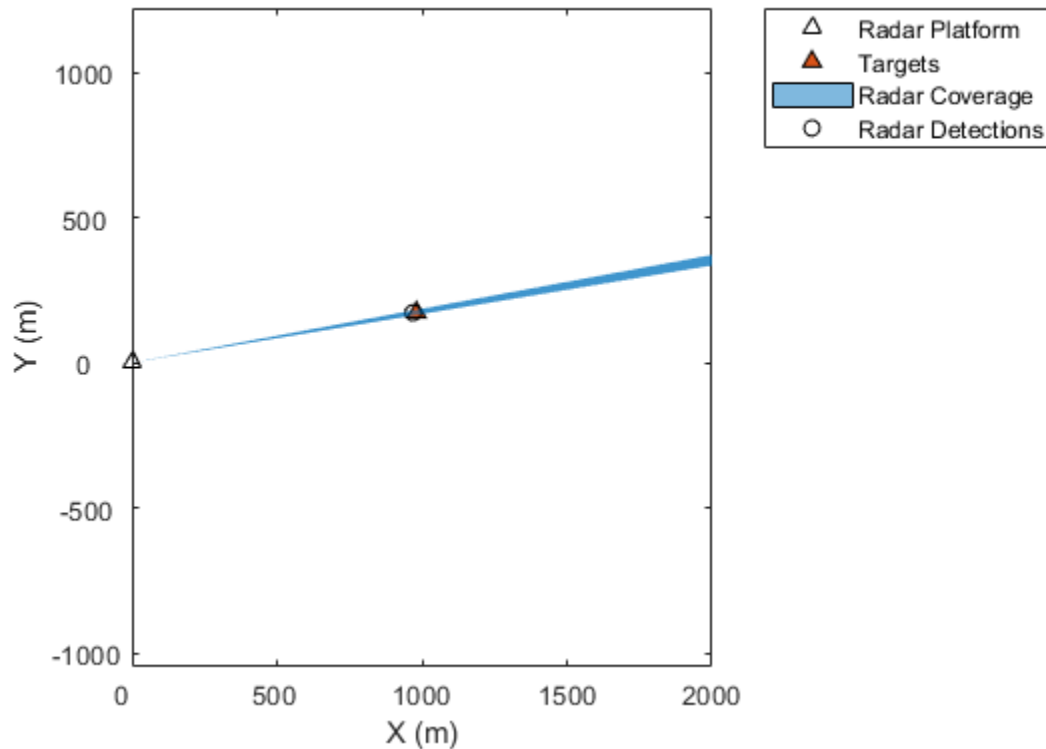
```
pltPlotter = platformPlotter(tp,DisplayName="Radar Platform");
tgtPlotter = platformPlotter(tp,DisplayName="Targets", ...
    MarkerFaceColor="#D95319");
```

```
plotPlatform(pltPlotter,[0 0 0])
plotPlatform(tgtPlotter,tgt.Position)
```

```
covPlotter = coveragePlotter(tp,DisplayName="Radar Coverage");
covcfg = coverageConfig(rdr);
plotCoverage(covPlotter,covcfg)
```

```
detPlotter = detectionPlotter(tp,DisplayName="Radar Detections");
plotDetection(detPlotter,detpos')
```

```
axis equal
```



## Algorithms

### Convenience Syntaxes

The convenience syntaxes set several properties together to model a specific type of radar.

#### No Scanning

Sets ScanMode to 'No scanning'.

#### Raster Scanning

This syntax sets these properties:

Property	Value
ScanMode	'Mechanical'
HasElevation	true
MaxMechanicalScanRate	[75; 75]
MechanicalScanLimits	[-45 45; -10 0]
ElectronicScanLimits	[-45 45; -10 0]

You can change the ScanMode property to 'Electronic' to perform an electronic raster scan over the same volume as a mechanical scan.

### Rotator Scanning

This syntax sets these properties:

Property	Value
ScanMode	'Mechanical'
FieldOfView	[1; 10]
HasElevation	false or true
MechanicalScanLimits	[0 360; -10 0]
ElevationResolution	10/sqrt(12)

### Sector Scanning

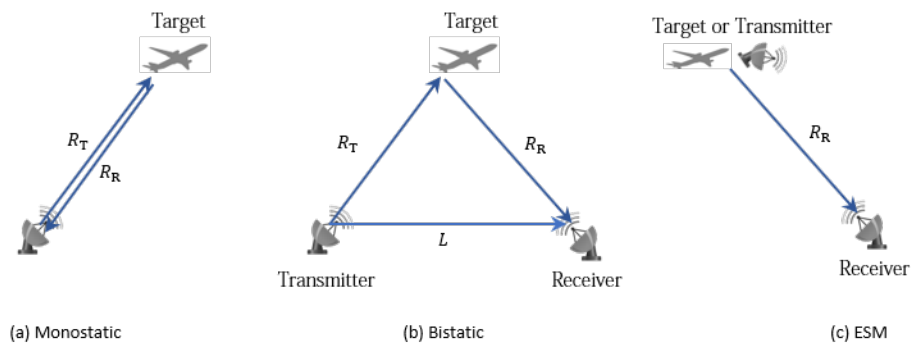
This syntax sets these properties:

Property	Value
ScanMode	'Mechanical'
FieldOfView	[1; 10]
HasElevation	false
MechanicalScanLimits	[-45 45; -10 0]
ElectronicScanLimits	[-45 45; -10 0]
ElevationResolution	10/sqrt(12)

Changing the ScanMode property to 'Electronic' lets you perform an electronic raster scan over the same volume as a mechanical scan.

### Radar Sensor Detection Modes

The radarDataGenerator System object can model three detection modes: monostatic, bistatic, and electronic support measures (ESM) as shown in the following figures.



For the monostatic detection mode, the transmitter and the receiver are collocated, as shown in figure (a). In this mode, the range measurement  $R$  can be expressed as  $R = R_T = R_R$ , where  $R_T$  and  $R_R$  are the ranges from the transmitter to the target and from the target to the receiver, respectively. In the radar sensor, the range measurement is  $R = ct/2$ , where  $c$  is the speed of light and  $t$  is the total time of the signal transmission. Other than the range measurement, a monostatic sensor can optionally report range-rate, azimuth, and elevation measurements of the target.

For the bistatic detection mode, the transmitter and the receiver are separated by a distance  $L$ . As shown in figure (b), the signal is emitted from the transmitter, reflected from the target, and received by the receiver. The bistatic range measurement  $R_b$  is defined as  $R_b = R_T + R_R - L$ . In the radar sensor, the bistatic range measurement is obtained by  $R_b = c\Delta t$ , where  $\Delta t$  is the time difference between the receiver receiving the direct signal from the transmitter and receiving the reflected signal from the target. Other than the bistatic range measurement, a bistatic sensor can also optionally report the bistatic range-rate, azimuth, and elevation measurements of the target. Since the bistatic range and the two bearing angles (azimuth and elevation) do not correspond to the same position vector, they cannot be combined into a position vector and reported in a Cartesian coordinate system. As a result, the measurements of a bistatic sensor can only be reported in a spherical coordinate system.

For the ESM detection mode, the receiver can only receive a signal reflected from the target or directly emitted from the transmitter, as shown in figure (c). Therefore, the only available measurements are the azimuth and elevation of the target or transmitter. These measurements can only be reported in a spherical coordinate system.

### Detection and Track State Coordinates

The format of the measurement states or track states is determined by the specifications of the `HasRangeRate`, `HasElevation`, `HasINS`, `TaregetReportFormat`, and `DetectionCoordinates` properties.

There are two general types of detection or track coordinates:

- Cartesian coordinates — Enabled by specifying the `DetectionCoordinates` property as `'Body'`, `'Scenario'`, or `'Sensor rectangular'`. The complete form of a Cartesian state is `[x; y; z; vx; vy; vz]`, where `x`, `y`, and `z` are the Cartesian positions and `vx`, `vy`, and `vz` are the corresponding velocities. You can only set `DetectionCoordinates` as `'Scenario'` when the `HasINS` property is set to `true`, so that the sensor can transform sensor detections or tracks to the scenario frame.
- Spherical coordinates — Enabled by specifying the `DetectionCoordinates` property as `'Sensor spherical'`. The complete form of a spherical state is `[az; el; rng; rr]`, where `az`, `el`, `rng`, and `rr` represent azimuth angle, elevation angle, range, and range rate, respectively. When the `DetectionMode` property of the sensor is set to `'ESM'` or `'Bistatic'`, the sensor can only report detections in the `'Sensor spherical'` frame.

When the `HasRangeRate` property is set to `false`, `vx`, `vy`, and `vz` are removed from the Cartesian state coordinates and `rr` is removed from the spherical coordinates.

When the `HasElevation` property is set to `false`, `z` and `vz` are removed from the Cartesian state coordinates and `el` is removed from the spherical coordinates.

When the `DetectionMode` property is set to `'ESM'`, the sensor can only report detections in the `'Sensor spherical'` frame as `[az; el]`.

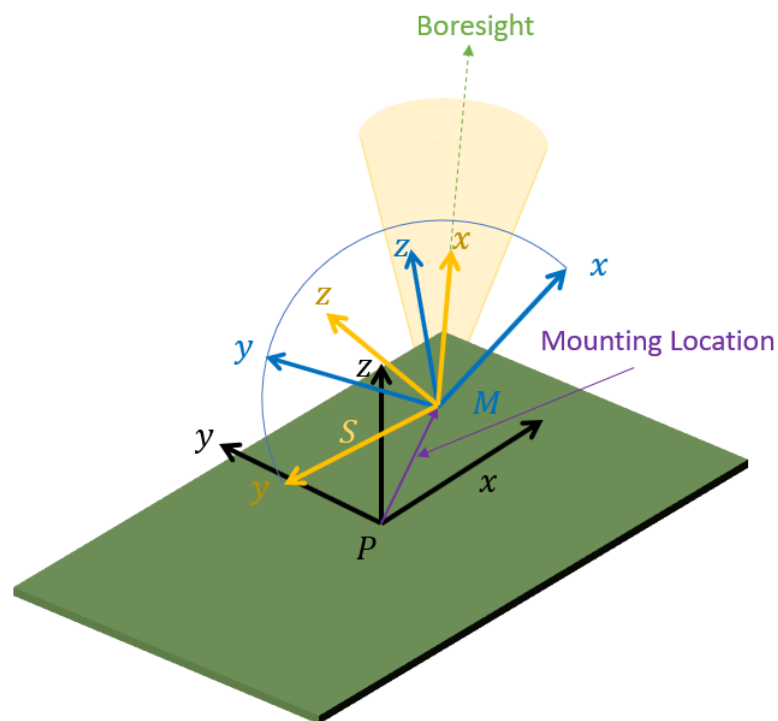
When the `DetectionMode` property is set to `'Bistatic'`, the sensor can only report detections in the `'Sensor spherical'` frame as `[az; el; rng; rr]`. Here, `rng` and `rr` are the bistatic range and range rate, respectively.

### Measurement Parameters

The `MeasurementParameters` property of an output detection consists of an array of structures that describes a sequence of coordinate transformations from a child frame to a parent frame, or the

inverse transformations. In most cases, the longest required sequence of transformations is Sensor → Platform → Scenario.

If the detections are reported in sensor spherical coordinates and `HasINS` is set to `false`, then the sequence consists only of one transformation from sensor to platform. In this transformation, the `OriginPosition` is same as the `MountingLocation` property of the sensor. The `Orientation` consists of two consecutive rotations. The first rotation, corresponding to the `MountingAngles` property of the sensor, accounts for the rotation from the platform frame ( $P$ ) to the sensor mounting frame ( $M$ ). The second rotation, corresponding to the azimuth and elevation angles of the sensor, accounts for the rotation from the sensor mounting frame ( $M$ ) to the sensor scanning frame ( $S$ ). In the  $S$  frame, the  $x$ -direction is the boresight direction, and the  $y$ -direction lies within the  $x$ - $y$  plane of the sensor mounting frame ( $M$ ).



If `HasINS` is `true`, the sequence of transformations consists of two transformations: first from the scenario frame to the platform frame, and then from the platform frame to the sensor scanning frame. In the first transformation, the `Orientation` is the rotation from the scenario frame to the platform frame, and the `OriginPosition` is the position of the platform frame origin relative to the scenario frame.

If the detections are reported in platform rectangular coordinates and `HasINS` is set to `false`, the transformation consists only of the identity.

The table shows the fields of the `MeasurementParameters` structure. Not all fields have to be present in the structure. The specific set of fields and their default values can depend on the type of sensor.

Field	Description
-------	-------------

Frame	Enumerated type indicating the frame used to report measurements. When detections are reported using a rectangular coordinate system, Frame is set to 'rectangular'. When detections are reported in spherical coordinates, Frame is set 'spherical' for the first structure.
OriginPosition	Position offset of the origin of the child frame relative to the parent frame, represented as a 3-by-1 vector.
OriginVelocity	Velocity offset of the origin of the child frame relative to the parent frame, represented as a 3-by-1 vector.
Orientation	3-by-3 real-valued orthonormal frame rotation matrix. The direction of the rotation depends on the IsParentToChild field.
IsParentToChild	A logical scalar indicating if Orientation performs a frame rotation from the parent coordinate frame to the child coordinate frame. If false, Orientation instead performs a frame rotation from the child coordinate frame to the parent coordinate frame.
HasElevation	A logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, if HasElevation is false, the measurements are reported assuming 0 degrees of elevation.
HasAzimuth	A logical scalar indicating if azimuth is included in the measurement.
HasRange	A logical scalar indicating if range is included in the measurement.
HasVelocity	A logical scalar indicating if the reported detections include velocity measurements. For measurements reported in a rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].

## References

- [1] Doerry, Armin W. "Earth Curvature and Atmospheric Refraction Effects on Radar Signal Propagation." Sandia Report SAND2012-10690, Sandia National Laboratories, Albuquerque, NM, January 2013. <https://prod.sandia.gov/techlib-noauth/access-control.cgi/2012/1210690.pdf>.
- [2] Doerry, Armin W. "Motion Measurement for Synthetic Aperture Radar." Sandia Report SAND2015-20818, Sandia National Laboratories, Albuquerque, NM, January 2015. <https://pdfs.semanticscholar.org/f8f8/cd6de8042a7a948d611bcfe3b79c48aa9dfa.pdf>.

**See Also**

radarScenario | radarTracker | radarEmitter | radarEmission | rcsSignature |  
radarChannel

**Introduced in R2021a**



# radarTransceiver

Create corresponding radar transceiver from radarDataGenerator

## Syntax

```
iqSensor = radarTransceiver(radarGenerator)
```

## Description

`iqSensor = radarTransceiver(radarGenerator)` creates a corresponding radar transceiver, `iqSensor`, based on the `radarDataGenerator` object, `radarGenerator`. The function configures the parameters in `iqSensor` so that you can process the signal it generates to obtain comparable detections to those returned from `radarGenerator`.

## Examples

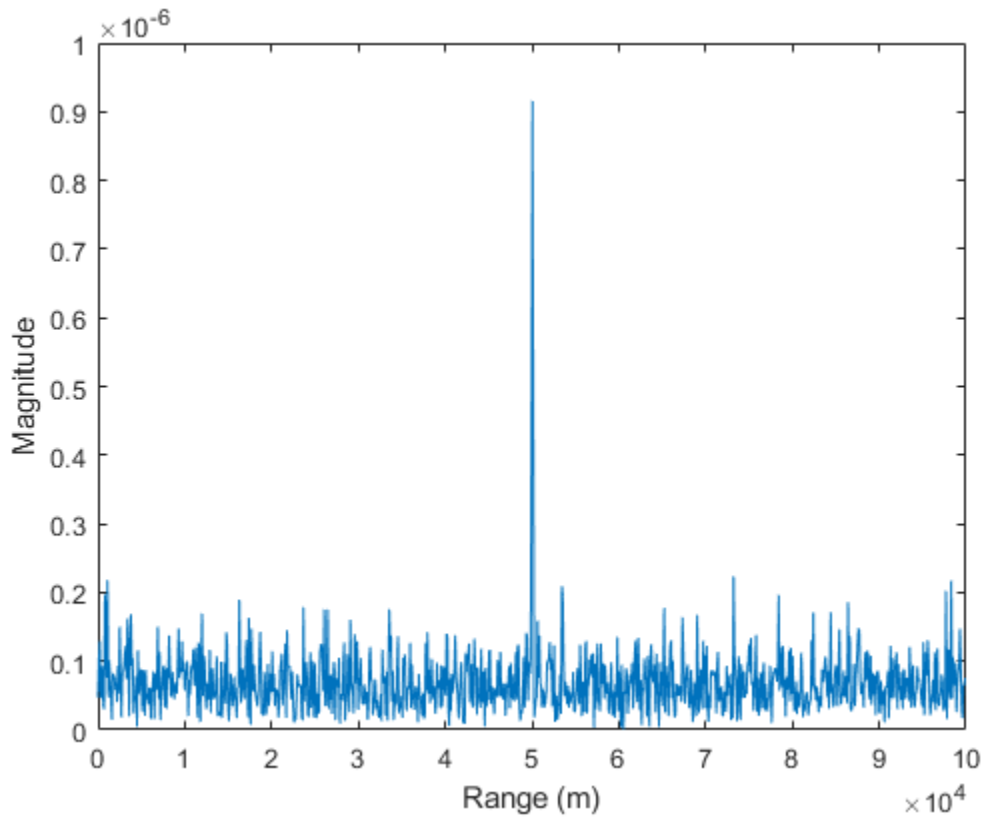
### Create Radar Transceiver from Radar Data Generator

Create a `radarDataGenerator` and generate a radar transceiver from it.

```
rdr = radarDataGenerator;  
iqsensor = radarTransceiver(rdr);
```

Produce radar signal from a target using the transceiver.

```
tgt = struct('Position',[50e3 0 0]);  
x = iqsensor(tgt,0);  
t = (0:numel(x)-1)/iqsensor.Waveform.SampleRate;  
plot(t*physconst('lightspeed')/2,abs(x))  
xlabel('Range (m)')  
ylabel('Magnitude')
```



## Input Arguments

### **radarGenerator** — Radar data generator

radarDataGenerator object

Radar data generator, specified as a radarDataGenerator object.

## Output Arguments

### **iqSensor** — Radar transceiver

radarTransceiver object

Radar transceiver, returned as a radarTransceiver object.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## **See Also**

radarDataGenerator | radarTransceiver | radarChannel

**Introduced in R2021a**

## surfaceReflectivityLand

Normalized reflectivity of land surface

### Description

Normalized reflectivity is the radar cross-section of a unit area of a land surface. Multiplying by the total area of a surface or the illuminated area of a surface gives the total radar cross-section. Normalized reflectivity is also referred to as surface  $\sigma^0$  and is a function of frequency and grazing angle.

To compute the normalized reflectivity:

- 1 Create the `surfaceReflectivityLand` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
refl = surfaceReflectivityLand  
refl = surfaceReflectivityLand(Name=Value)
```

#### Description

`refl = surfaceReflectivityLand` creates a normalized reflectivity object `refl` for a land surface. Use this object to generate a normalized radar cross section (NRCS). This syntax creates a normalized reflectivity object with a 'Barton' land Model and a 'Flatland' LandType.

`refl = surfaceReflectivityLand(Name=Value)` creates a normalized reflectivity object for a land surface with the specified property Name set to the specified Value. You can specify additional name-value pair arguments in any order as (Name1=Value1, ... ,NameN=ValueN).

Example: `refl = surfaceReflectivityLand(Model="GIT",LandType="Soil",SurfaceHeightStandardDeviation=1)` creates a normalized reflectivity object for land using the GIT model with a LandType of Soil and a SurfaceHeightStandardDeviation of 1.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**Model — Land reflectivity model**

'Barton' (default) | 'APL' | 'Billingsley' | 'GIT' | 'Morchin' | 'Nathanson' | 'UlabyDobson' | 'ConstantGamma'

Land reflectivity model, specified as 'Barton', 'APL', 'Billingsley', 'GIT', 'Morchin', 'Nathanson', 'Nathanson', 'UlabyDobson', or 'ConstantGamma'. Descriptions of the models and land types are shown in the table “Land Reflectivity Models and Land Types” on page 4-222.

**LandType — Land type**

char | string

Land type, specified as a char or string. The allowable land type and their default values depend on the Model property. If the Model property is not specified, the default land type is 'Flatland'. Descriptions of the models and land types are shown in the table “Land Reflectivity Models and Land Types” on page 4-222.

Data Types: char | string

**SurfaceHeightStandardDeviation — Height standard deviation**

0 (default) | scalar

Standard deviation of the surface height, specified as a positive scalar. Units are in meters.

**Dependencies**

To enable this property, set the Model property to 'GIT'.

Data Types: double

**Polarization — Polarization of reflectivity model**

'H' (default) | 'V'

Polarization of reflectivity model, specified as 'H' or 'V'. 'H' designates horizontal polarization and 'V' designates vertical polarization.

**Dependencies**

To enable this property, set the Model property to 'UlabyDobson'.

Data Types: char | string

**Gamma — Terrain gamma value**

-20 (default) | real scalar

Terrain gamma value used in the constant gamma clutter reflectivity model, specified as a scalar. The gamma value depends on both terrain type and the operating frequency. The default value is representative of flat land. Units are in dB.

Example: -15

**Dependencies**

To enable this property, set the Model property to 'ConstantGamma'.

Data Types: double

**Speckle — Speckle distribution type**

'None' (default) | 'Lognormal' | 'Rayleigh' | 'Weibull' | 'Custom'

Speckle distribution type, specified as 'None', 'Lognormal', 'Rayleigh', 'Weibull', or 'Custom'. Speckle is a multiplicative factor used to make clutter data appear noisier and is especially applicable to imaging applications.

Speckle is correlated with clutter RCS and is applied as  $I = \sigma * n$ , where  $\sigma$  represents the clutter RCS and  $n$  represents random numbers, which are often drawn from an independent identically-distributed unity mean noise statistical distribution.

- **None** - No speckle is applied.
- **Lognormal** - Speckle has a lognormal distribution. Define the distribution using the `SpeckleMean` and `SpeckleStandardDeviation` properties. Default values of these properties create speckle with a normalized mean lognormal distribution.
- **Rayleigh** - Speckle has a Rayleigh distribution. Define the distribution using the `SpeckleScale` property. The default value of this property creates speckle with a unit mean Rayleigh distribution.
- **Weibull** - Speckle has a Weibull distribution. Define the distribution using the `SpeckleScale` and `SpeckleShape` properties. The default values of these properties create speckle with a unit mean Rayleigh distribution.

Data Types: char | string

#### **SpeckleMean — Mean of value of lognormal-distributed speckle**

`-0.5*log(2)` (default) | scalar

Mean value of lognormal-distributed speckle, specified as a scalar.

##### **Dependencies**

To enable this property, set the `Speckle` property to 'Lognormal'.

Data Types: double

#### **SpeckleStandardDeviation — Standard deviation of lognormal-distributed speckle**

`sqrt(log(2))` (default) | non-negative scalar

Standard deviation of lognormal-distributed speckle, specified as a non-negative scalar.

##### **Dependencies**

To enable this property, set the `Speckle` property to 'Lognormal'.

Data Types: double

#### **SpeckleScale — Scale parameter for Weibull and Rayleigh speckle distribution**

`sqrt(4/pi)` (default) | non-negative scalar

Scale parameter for speckle for the Rayleigh and Weibull distributions, specified as a positive scalar.

##### **Dependencies**

To enable this property, set the `Speckle` property to 'Rayleigh' or 'Weibull'.

Data Types: double

#### **SpeckleShape — Shape value for Weibull distribution**

`2` (default) | positive scalar

Shape value for the Weibull speckle distribution, specified as a positive scalar.

## Dependencies

To enable this property, set the `Speckle` property to `Weibull`.

Data Types: `double`

## Usage

## Syntax

```
nrcs = refl(graz, freq)
```

## Description

`nrcs = refl(graz, freq)` returns the normalized radar cross section `nrcs` at grazing angle `graz` and frequency `freq`. When the `Model` property is set to `'Billingsley'`, `graz` is interpreted as a depression angles.

## Input Arguments

### **graz** — Grazing or depression angle

scalar |  $M$ -length vector of real values

Grazing or depression angle of a surface relative to the radar, specified as a scalar or an  $M$ -length row vector of real values. When the `land Model` property is set to `'Billingsley'`, the angle is interpreted as a depression angle `depressionang` between  $-90^\circ$  and  $90^\circ$ . For all other models, the angle is interpreted as a grazing angle `grazingang` ranging from  $0^\circ$  to  $90^\circ$ . Units are in degrees.

### **freq** — Transmitted frequencies

$10e9$  (default) | positive scalar |  $N$ -length vector of positive values

Transmitted frequencies, specified as a positive scalar or  $N$ -length vector of positive values. Units are in Hz.

Example: `freq = 7*10e9`

## Output Arguments

### **nrcs** — Normalized surface reflectivity

real-valued  $N$ -length row vector | real-valued  $M$ -by- $N$  matrix

Normalized surface reflectivity, returned as either a real-valued  $N$ -length row vector or a real-valued  $M$ -by- $N$  matrix. Normalized reflectivity is also called normalized radar cross section.  $M$  is the length of the grazing angle or depression angle vector `graz` and  $N$  is the length of the frequency vector `freq`. `nrcs` is dimensionless but often expressed as  $m^2/m^2$ .

## Object Functions

To use an object function, specify the `System` object as the first input argument. For example, to release system resources of a `System` object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

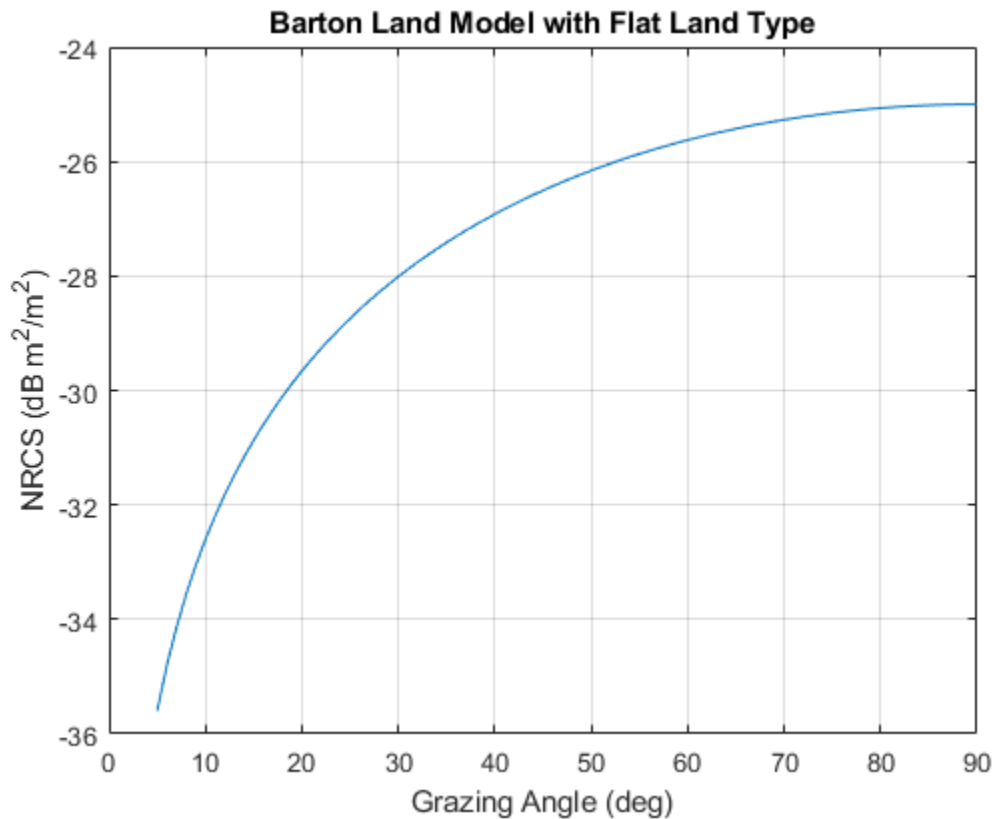
`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## Examples

### Surface Reflectivity of Default Land Model

Plot the normalized radar cross-section for grazing angles from 5 to 90 degrees. Assume the default 'Barton' land Model and 'Flatland' LandType. Set the radar frequency to 1 GHz.

```
grazAng = 5:90;  
freq = 1e9;  
reflectivity = surfaceReflectivityLand;  
nrCS = reflectivity(grazAng, freq);  
plot(grazAng, pow2db(nrCS))  
grid on  
xlabel('Grazing Angle (deg)')  
ylabel('NRCS (dB m2/m2)')  
title('Barton Land Model with Flat Land Type')
```





## Create Reflective Land Surface in Radar Scenario

Configure a `radarScenario` to simulate a reflective land surface. Add a land surface object to define the physical properties of the scenario surface. The surface is a simple 200-by-200 meter rectangle. Use the `surfaceReflectivityLand` function to create a constant-gamma reflectivity model with a gamma value of -10 dB. Use the scenario `landSurface` method to add the rectangular land region and the radar reflectivity model to the scenario. Use a surface reference height of 16 meters.

```
scene = radarScenario(UpdateRate = 0, IsEarthCentered = false);
refl = surfaceReflectivityLand(Model = "ConstantGamma", Gamma = -10);
srf = landSurface(scene,RadarReflectivity = refl, ...
    Boundary=[-100 100; -100 100],ReferenceHeight = 16)

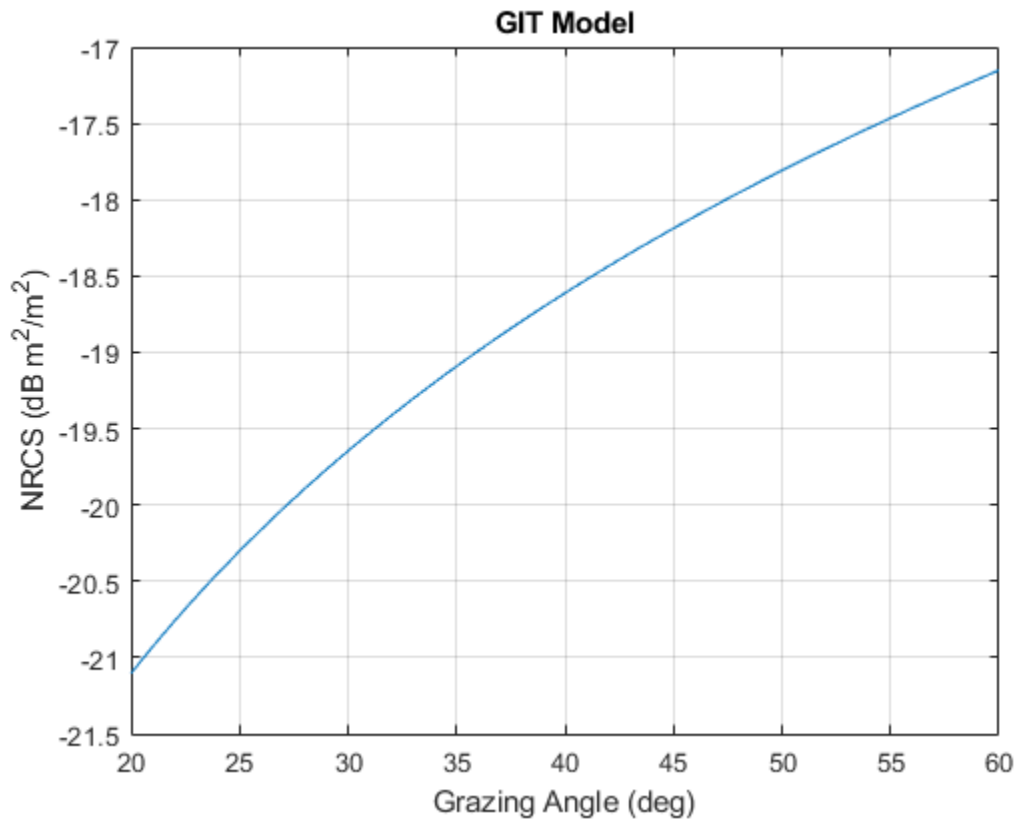
srf =
    LandSurface with properties:

        RadarReflectivity: [1x1 surfaceReflectivityLand]
        ReflectivityMap: 1
        ReferenceHeight: 16
        Boundary: [2x2 double]
        Terrain: []
```

## Surface Reflectivity of GIT Land Model

Create a normalized reflectivity object using the GIT 'Model' and a 'Soil' land type. Obtain the normalized radar cross-section at a frequency of 3 GHz over grazing angles from 20 to 60 degrees. Assume a surface height standard deviation of two meters. Plot the surface reflectivity.

```
grazAng = 20:60;
freq = 10e9;
reflectivity = surfaceReflectivityLand(Model="GIT", ...
    LandType="Soil",SurfaceHeightStandardDeviation=2);
nracs = reflectivity(grazAng,freq);
plot(grazAng,pow2db(nracs))
grid on
xlabel('Grazing Angle (deg)')
ylabel('NRCS (dB m^2/m^2)')
title('GIT Model')
```



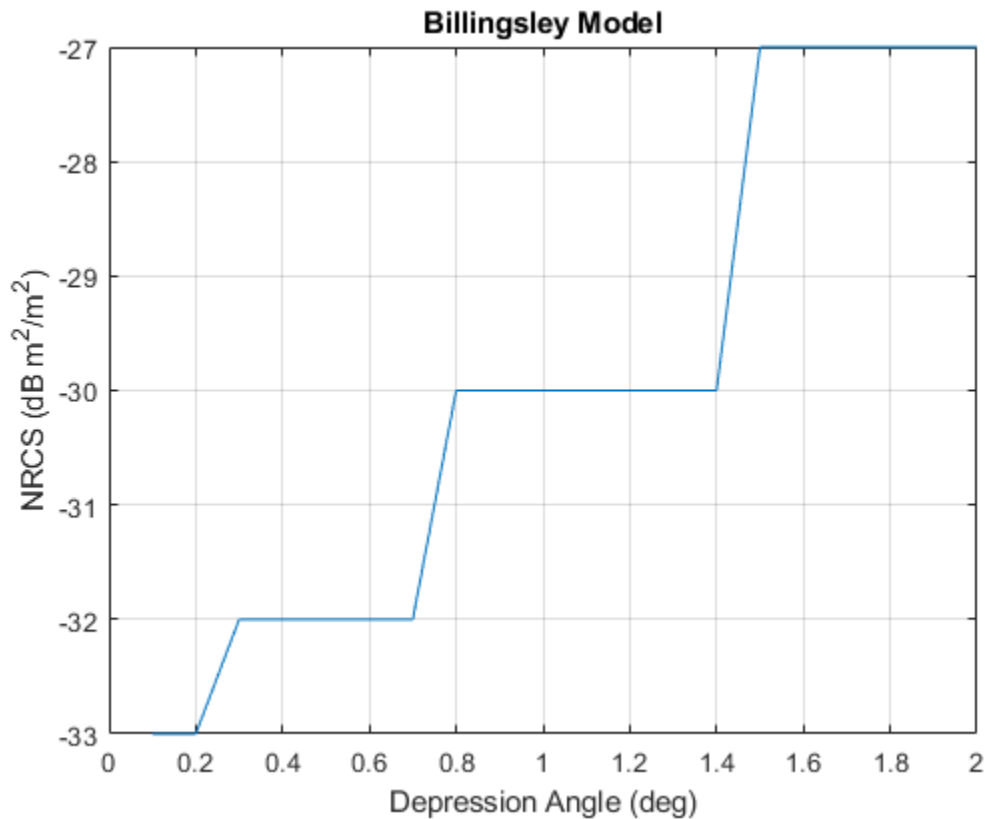
### Surface Reflectivity of Billingsley Land Model

Create a normalized reflectivity object using the Billingsley 'Model' and a 'LowReliefRural' land type. Obtain the normalized radar cross-section at a frequency of 3 GHz over depression angles from 0.1 to 3 degrees. Plot the surface reflectivity.

```

depAng = 0.1:0.1:2;
freq = 3e9;
reflectivity = surfaceReflectivityLand(Model="Billingsley", ...
    LandType="LowReliefRural");
nrCS = reflectivity(depAng, freq);
plot(depAng, pow2db(nrCS))
grid on
xlabel('Depression Angle (deg)')
ylabel('NRCS (dB m^2/m^2)')
title('Billingsley Model')

```



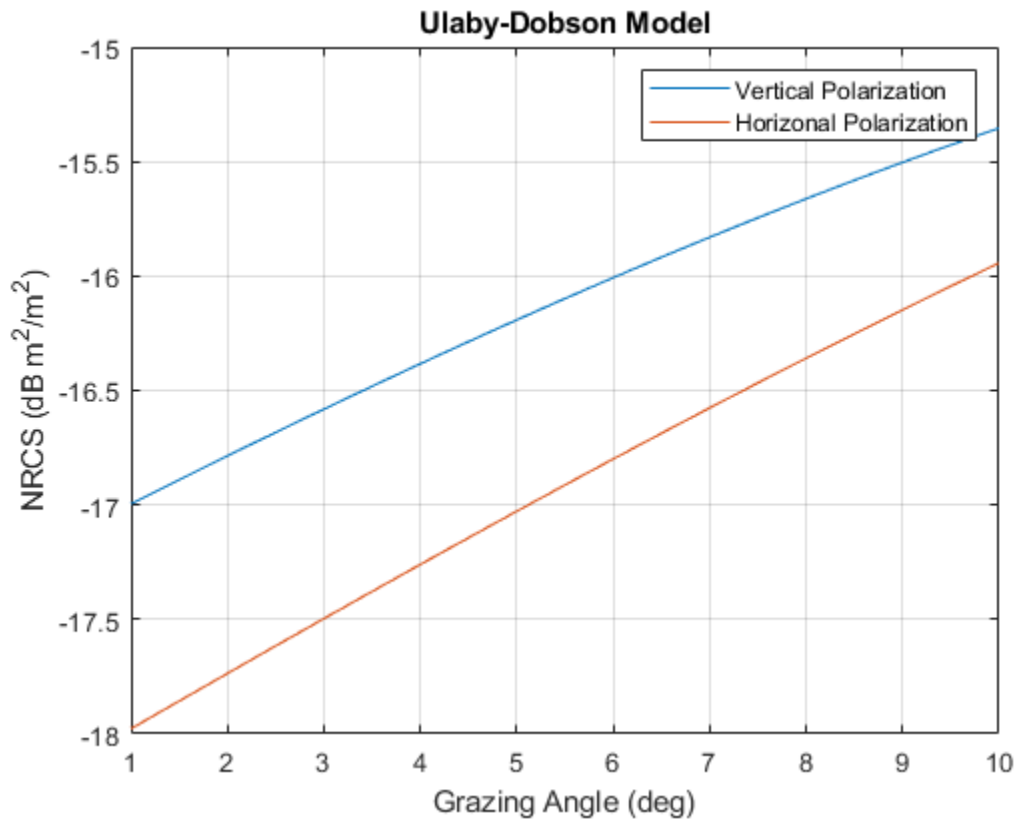
### Surface Reflectivity of Ulaby-Dobson Land Model

Create a normalized reflectivity object using the Ulaby-Dobson model for a grass land type. Obtain the normalized radar cross-section for both vertical and horizontal polarizations at a frequency of 10 GHz over grazing angles from 1 to 10 degrees. Plot the surface reflectivities.

```

grazAng = 1:0.1:10;
freq = 10e9;
reflectivity_v = surfaceReflectivityLand(Model="UlabyDobson", ...
    LandType="Grass",Polarization="V");
nrCS_v = reflectivity_v(grazAng,freq);
reflectivity_h = surfaceReflectivityLand(Model="UlabyDobson", ...
    LandType="Grass",Polarization="H");
nrCS_h = reflectivity_h(grazAng,freq);
plot(grazAng,pow2db(nrCS_v))
hold on
plot(grazAng,pow2db(nrCS_h))
grid on
legend('Vertical Polarization','Horizontal Polarization')
xlabel('Grazing Angle (deg)')
ylabel('NRCS (dB m^2/m^2)')
title('Ulaby-Dobson Model')

```



### Create Land Surface in Radar Scenario

Create a surface with two hills. Plot the surface on a 200-by-200 meter grid with grid points one meter apart. Add the surface to a radar scenario. Assume the surface has a radar reflectivity defined by a constant gamma model.

```
[x,y] = meshgrid(linspace(-100,100,201));
ht1 = 40*exp(-(x.^2 + y.^2)/30^2);
ht2 = 100*exp(-((x-60).^2 + y.^2)/25^2);
ht = ht1 + ht2;
p = surfc(x(1,:),y(:,1),ht);
axis equal
axis tight
shading interp
simTime = 3;
scene = radarScenario(UpdateRate = 1, ...
    IsEarthCentered = false,StopTime = simTime);
gammaDB = surfacegamma('Flatland');
refl = surfaceReflectivityLand(Model = 'ConstantGamma',Gamma = gammaDB);
srf = landSurface(scene,RadarReflectivity = refl, ...
    Terrain = ht,Boundary = [-100,100;-100,100]);
```

Use surface manager to identify the surface.

```
scene.SurfaceManager
```

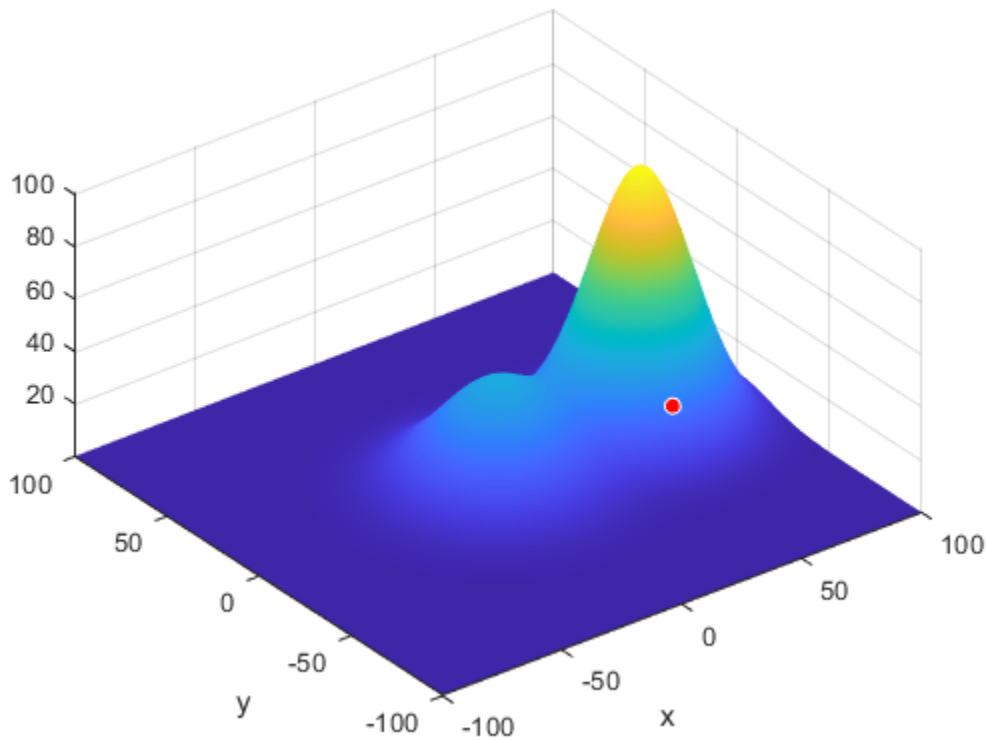
```
ans =  
  SurfaceManager with properties:  
  
    UseOcclusion: 1  
    Surfaces: [1x1 radar.scenario.LandSurface]
```

```
scene.SurfaceManager.Surfaces
```

```
ans =  
  LandSurface with properties:  
  
    RadarReflectivity: [1x1 surfaceReflectivityLand]  
    ReflectivityMap: 1  
    ReferenceHeight: 0  
    Boundary: [2x2 double]  
    Terrain: [201x201 double]
```

Obtain and plot the height of the surface at the point (50,-30).

```
xt = 50;  
yt = -30;  
htx = height(srf,[xt,yt])  
  
htx = 21.1046  
  
hold on  
plot3(xt,yt,htx+5,'ow','MarkerFaceColor','r')  
xlabel('x')  
ylabel('y')  
hold off
```



## More About

### Land Reflectivity Models and Land Types

Model	Land Type	Range of Validity	Settable Properties
'Barton' - Constant-gamma mathematical model generally applicable over medium grazing angles. 'Barton' is the default model. See [1] [2], and [3].	'RuggedMountains'	<ul style="list-style-type: none"> <li>Grazing angle 20 - 60 degrees</li> <li>Frequency 1 - 10 GHz</li> </ul>	LandType
	'Mountains'		Speckle
	'Metropolitan'		SpeckleMean
	'Urban'		SpeckleStandardDeviation
	'WoodedHills'		SpeckleScale
	'RollingHills'		SpeckleShape
	'Woods'		
	'Farm'		
	'Desert'		
	'Flatland' (default for model)		
'Smooth'			

Model	Land Type	Range of Validity	Settable Properties
'APL' - This model also known as the ADSAM model. Low-fidelity constant-gamma mathematical model that includes specular scattering. See [4].	'Urban'	<ul style="list-style-type: none"> <li>Grazing angle 0 - 90 degrees</li> <li>Frequency 1 - 100 GHz</li> </ul>	LandType
	'HighRelief'		Speckle
	'LowRelief' (default for model)		SpeckleMean
			SpeckleStandardDeviation
			SpeckleScale
			SpeckleShape
'Billingesley' - High-validity empirical model generally applicable for low depression angles less than 2 degrees. See [5].	'LowReliefRural' (default for model)	<ul style="list-style-type: none"> <li>Depression angle -0.75 - 2 degrees</li> <li>Frequency - VHF (0.030 - 0.3), UHF (0.3 - 1), L (1 - 2), S (2 - 4), X (8 - 12) GHz</li> </ul>	LandType
	'LowReliefForest'		Speckle
	'Farm'		SpeckleMean
	'Desert'		SpeckleStandardDeviation
	'Marsh'		SpeckleScale
	'Grassland'		SpeckleShape
	'HighReliefRural'		
	'HighReliefForest'		
	'Mountains'		
	'Urban'		
'LowReliefUrban'			
'GIT' - Georgia Institute of Technology semi-empirical model that takes into account terrain roughness. Generally applicable for medium grazing angles. See [6]	'Soil' (default for Model)	<ul style="list-style-type: none"> <li>Grazing angle 20 - 65 degrees</li> <li>Frequency 3 - 15 GHz</li> </ul>	LandType
	'Grass'		Speckle
	'TallGrass'		SpeckleMean
	'Trees'		SpeckleStandardDeviation
	'Urban'		SpeckleScale
		SpeckleShape	
		SurfaceHeightStandardDeviation	
'Morchin' - Mathematical model generally applicable for high grazing angles for frequencies from UHF to C-band. See [7].	'Desert'	<ul style="list-style-type: none"> <li>Grazing angle 70 - 90 degrees</li> <li>Frequencies UHF (0.3 - 1) L (1 - 2) S (2 - 4) C (4 - 8)</li> </ul>	LandType
	'Farm' (default for Model)		Speckle
	'Woods'		SpeckleMean
	'Mountains'		SpeckleStandardDeviation
			SpeckleScale
			SpeckleShape

Model	Land Type	Range of Validity	Settable Properties
'Nathanson' - Applicable up to Ka band for low grazing angle surface radars and medium grazing angle airborne radars for low mountains, farmland, and wooded areas. See [3].	'Desert' 'Farm' (default for Model) 'Woods' 'Jungle' 'RollingHills' 'Urban'	<ul style="list-style-type: none"> <li>Grazing angle 0 - 60 degrees</li> <li>Frequency L (1 - 2), S (2 - 4), C (4 - 8), X (8 --12), Ku (12 --18), Ka (32 -- 36) GHz</li> </ul>	LandType Speckle SpeckleMean SpeckleStandardDeviation SpeckleScale SpeckleShape
'UlabyDobson' - High-validity semi-empirical model for low to medium grazing angles covering L-band to Ku, taking into account polarization. See [8].	'Soil' (default for Model) 'Grass' 'Shrubs' 'ShortVegetation'	<ul style="list-style-type: none"> <li>Grazing angle 0 - 60 degrees</li> <li>Frequency L (1 - 2), S (2 - 4), C (4 - 8), X (8 --12), Ku (12 --18) GHz</li> </ul>	LandType Polarization Speckle SpeckleMean SpeckleStandardDeviation SpeckleScale SpeckleShape
'ConstantGamma' - Mathematical model for the normalized reflectivity. See "Constant Gamma Model" on page 4-224.			Gamma

### Constant Gamma Model

The constant-gamma model expresses a simple analytic relationship between the normalized radar cross-section and grazing angle.

$$\sigma = 10^{(\gamma/10)} \sin(\theta)$$

$\gamma$  is defined by the Gamma property and  $\theta$  is the grazing angle input argument `graz` given in degrees. The default value of Gamma is -20, which is representative of flat land.

### Speckle Model

Speckle is modeled as an uncorrelated, multiplicative factor  $I = \sigma \cdot n$ , where  $\sigma$  represents the clutter RCS and  $n$  are independent identically distributed (IDD) mean noise samples with unity mean. Because speckle is correlated with underlying terrain RCS, it is usually applied to radar intensity. The speckle noise models include Weibull, Rayleigh, and lognormal.

### References

[1] Barton, David Knox. *Radar Equations for Modern Radar*. Artech House, 2013.

[2] Long, Maurice W. *Radar Reflectivity of Land and Sea*. 3rd ed, Artech House, 2001.



- [3] Nathanson, Fred E., et al. *Radar Design Principles: Signal Processing and the Environment*. 2. ed., Repr, Scitech Publ, 2004.
- [4] Reilly, J. P., R. L. McDonald, and G. D. Dockery. "RF-Environment Models for the ADSAM Program." Report No. A1A97U-070, Laurel, MD: Johns Hopkins University Applied Physics Laboratory, August 22, 1997.
- [5] Billingsley, J. Barrie. *Low-Angle Radar Land Clutter: Measurements and Empirical Models*. William Andrew Pub. : SciTech Pub. ; Institution of Electrical Engineers, 2002.
- [6] Richards, M. A., et al., editors. *Principles of Modern Radar*. SciTech Pub, 2010.
- [7] Morchin, Fred E., J. Patrick Reilly, and Marvin Cohen. *Radar Design Principles: Signal Processing and the Environment*. 2nd ed. New York: McGraw-Hill, 1991.
- [8] Ulaby, Fawwaz T., and M. Craig Dobson. *Handbook of Radar Scattering Statistics for Terrain*. Artech House, 1989.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[clutterSurfaceRCS](#) | [landroughness](#) | [landreflectivity](#) | [searoughness](#) | [seareflectivity](#) | [surfaceReflectivity](#) | [surfaceReflectivitySea](#) | [surfaceReflectivityCustom](#) | [grazingang](#) | [depressionang](#) | [radarScenario](#) | [radarDataGenerator](#)

### Topics

"Introduction to Radar Scenario Clutter Simulation"  
"Generate Clutter and Target Returns for MTI Radar"  
"Simulate Radar Detections of Surface Targets in Clutter"

### Introduced in R2022a

## surfaceReflectivitySea

Normalized reflectivity of sea surface

### Description

Normalized reflectivity is the radar cross-section of a unit area of a sea surface. Multiplying by the total area of a surface or the illuminated area of a surface gives the total radar cross-section. Normalized reflectivity is also referred to as surface  $\sigma^0$  and is a function of frequency and grazing angle.

To compute the normalized reflectivity:

- 1 Create the `surfaceReflectivitySea` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
refl = surfaceReflectivitySea  
refl = surfaceReflectivitySea(Name=Value)
```

#### Description

`refl = surfaceReflectivitySea` creates a normalized reflectivity object, `refl`, for a sea surface. Use this object to generate a normalized radar cross section (NRCS). This syntax assumes a 'NRL' sea model with a sea state of zero.

`refl = surfaceReflectivitySea(Name=Value)` also creates a normalized reflectivity object for a sea surface with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as `(Name1=Value1,...,NameN=ValueN)`.

Example: `refl = surfaceReflectivitySea(Model="Hybrid",SeaState=2,Speckle="Rayleigh")` creates a normalized reflectivity object for a sea surface using the Hybrid model with a `SeaState` of 2 and a Rayleigh `Speckle` type.

#### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**Model — Sea reflectivity model**

'NRL' (default) | 'APL' | 'GIT' | 'Hybrid' | 'Masuko' | 'Nathanson' | 'RRE' | 'Sittrop' | 'TSC' | 'ConstantGamma'

Sea reflectivity model, specified as 'NRL', 'APL', 'GIT', 'Hybrid', 'Masuko', 'Nathanson', 'RRE', 'Sittrop', 'TSC', or 'ConstantGamma'. The table “Sea Reflectivity Models” on page 4-233 summarize the sea surface models available in the radar simulation and their domain of application

**SeaState — Sea state**

1 (default) | nonnegative integer

Sea state, specified as a nonnegative integer from 0 - 8.

Data Types: double

**Polarization — Polarization of reflectivity model**

'H' (default) | 'V'

Polarization of reflectivity model, specified as 'H' for horizontal polarization or 'V' for vertical polarization.

**Dependencies**

To enable this property, set the Model property to any value except 'ConstantGamma'.

**Gamma — Sea gamma value**

-40 (default) | real scalar

Sea gamma value used in the constant gamma clutter reflectivity model, specified as a scalar. The gamma value depends on both sea state and the operating frequency. Units are in dB.

Example: -25

**Dependencies**

To enable this property, set the Model property to ConstantGamma.

Data Types: double

**Speckle — Speckle distribution type**

'None' (default) | 'Lognormal' | 'Rayleigh' | 'Weibull' | 'Custom'

Speckle distribution type, specified as 'None', 'Lognormal', 'Rayleigh', 'Weibull', or 'Custom'. Speckle is a multiplicative factor used to make clutter data appear noisier and is especially applicable to imaging applications.

Speckle is correlated with clutter RCS and is applied as  $I = \sigma * n$ , where  $\sigma$  represents the clutter RCS and  $n$  represents random numbers, which are often drawn from an independent identically-distributed unity mean noise statistical distribution.

- **None** - No speckle is applied.
- **Lognormal** - Speckle has a lognormal distribution. Define the distribution using the `SpeckleMean` and `SpeckleStandardDeviation` properties. Default values of these properties create speckle with a normalized mean lognormal distribution.
- **Rayleigh** - Speckle has a Rayleigh distribution. Define the distribution using the `SpeckleScale` property. The default value of this property creates speckle with a unit mean Rayleigh distribution.

- **Weibull** – Speckle has a Weibull distribution. Define the distribution using the `SpeckleScale` and `SpeckleShape` properties. The default values of these properties create speckle with a unit mean Rayleigh distribution.

Data Types: `char` | `string`

### **SpeckleMean — Mean of value of lognormal-distributed speckle**

`-0.5*log(2)` (default) | `scalar`

Mean value of lognormal-distributed speckle, specified as a scalar.

#### **Dependencies**

To enable this property, set the `Speckle` property to `'Lognormal'`.

Data Types: `double`

### **SpeckleStandardDeviation — Standard deviation of lognormal-distributed speckle**

`sqrt(log(2))` (default) | `non-negative scalar`

Standard deviation of lognormal-distributed speckle, specified as a non-negative scalar.

#### **Dependencies**

To enable this property, set the `Speckle` property to `'Lognormal'`.

Data Types: `double`

### **SpeckleScale — Scale parameter for Weibull and Rayleigh speckle distribution**

`sqrt(4/pi)` (default) | `non-negative scalar`

Scale parameter for speckle for the Rayleigh and Weibull distributions, specified as a positive scalar.

#### **Dependencies**

To enable this property, set the `Speckle` property to `'Rayleigh'` or `'Weibull'`.

Data Types: `double`

### **SpeckleShape — Shape value for Weibull distribution**

`2` (default) | `positive scalar`

Shape value for the Weibull speckle distribution, specified as a positive scalar.

#### **Dependencies**

To enable this property, set the `Speckle` property to `Weibull`.

Data Types: `double`

## **Usage**

## **Syntax**

```
nrcs = refl(graz, freq)
nrcs = refl(graz, freq, lookang)
```

## Description

`nrcs = refl(graz, freq)` returns the normalized radar cross section `nrcs` at grazing angle `graz` and frequency `freq`.

`nrcs = refl(graz, freq, lookang)` also specifies the radar look angle `lookang` with respect to the wind direction. To enable this syntax, set the `Model` property to `'APL'`, `'GIT'`, `'Hybrid'`, `'Masuko'`, `'Sittrop'`, or `'TSC'`.

## Input Arguments

### **graz** — Grazing angle

nonnegative scalar | length-*M* row vector of nonnegative values

Grazing angle of surface relative to radar, specified as a scalar or a length-*M* row vector of nonnegative values. Grazing angles must lie between 0° and 90°. Units are in degrees.

### **freq** — Transmitted frequencies

10e9 (default) | positive scalar | *N*-length vector of positive values

Transmitted frequencies, specified as a positive scalar or *N*-length vector of positive values. Units are in Hz.

Example: `freq = 7*10e9`

### **lookangle** — Look angle with respect to wind direction

0 (default) | scalar

Look angle with respect to wind direction, specified as a scalar between 0° and 180°. The look angle is zero when looking upwind.

## Dependencies

To enable this argument, set the `Model` property to `'APL'`, `'GIT'`, `'Hybrid'`, `'Masuko'`, `'Sittrop'`, or `'TSC'`.

Data Types: `double`

## Output Arguments

### **nrcs** — Normalized surface reflectivity

real-valued *N*-length row vector | real-valued *M*-by-*N* matrix

Normalized surface reflectivity, returned as either a real-valued *N*-length row vector or a real-valued *M*-by-*N* matrix. Normalized reflectivity is also called normalized radar cross section. *M* is the length of the grazing angle or depression angle vector `graz` and *N* is the length of the frequency vector `freq`. `nrcs` is dimensionless but often expressed as  $\text{m}^2/\text{m}^2$ .

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

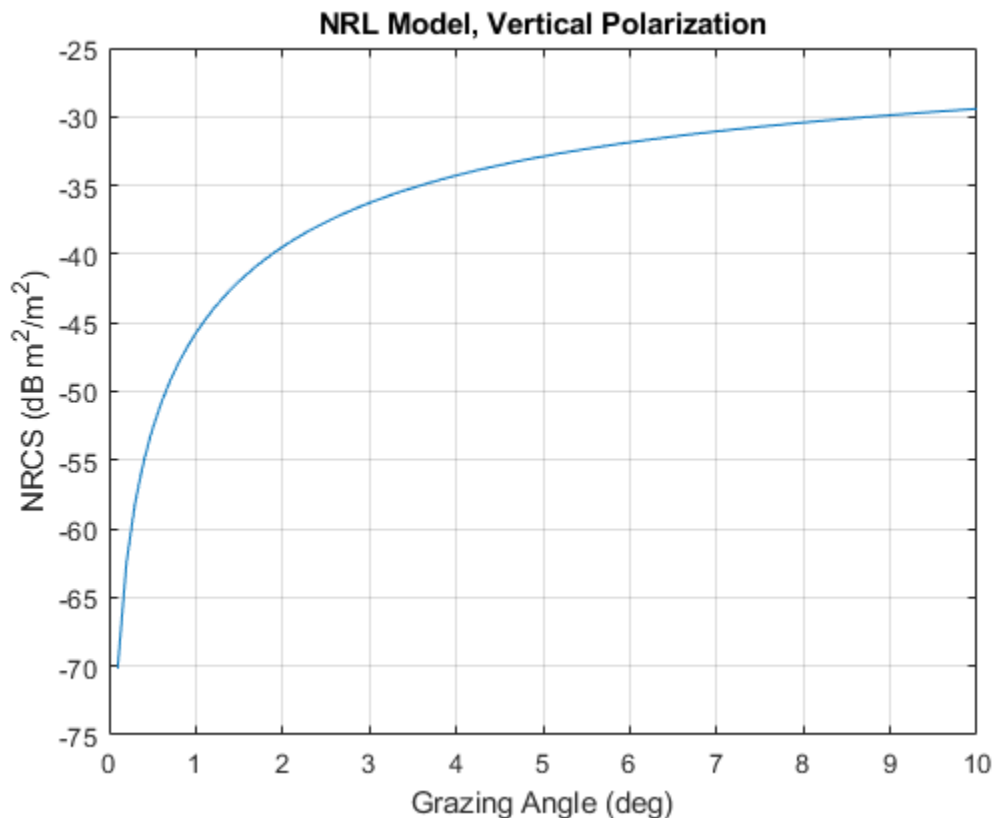
step     Run System object algorithm  
release   Release resources and allow changes to System object property values and input characteristics  
reset     Reset internal states of System object

## Examples

### Create Sea Surface Reflectivity Model

Create a sea surface normalized reflectivity object using the default NRL model and a sea state of 6. Obtain the normalized reflectivity at a frequency of 1 GHz over grazing angles from 0.1 to 10 degrees and assume vertical polarization. Plot the normalized reflectivity as a function of grazing angle.

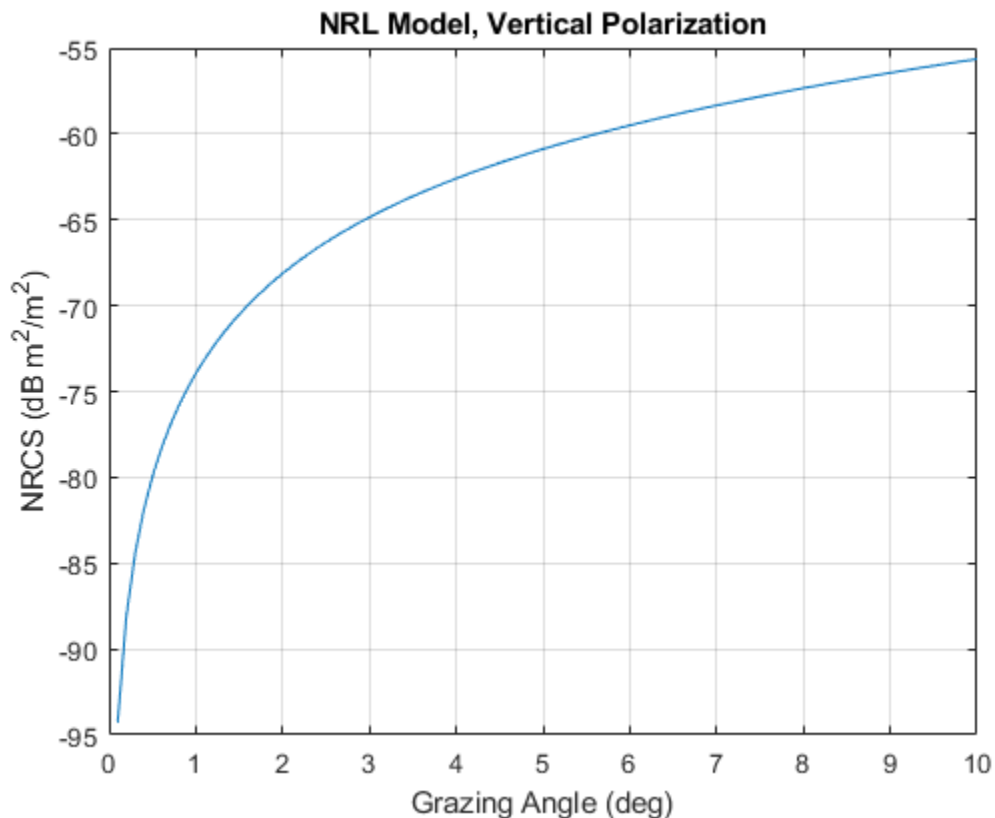
```
grazAng = 0.1:0.1:10;  
freq = 1e9;  
seastate = 6;  
pol = 'V';  
refl = surfaceReflectivitySea(SeaState = seastate,Polarization = pol);  
nrCS = refl(grazAng,freq);  
plot(grazAng,pow2db(nrCS))  
grid on  
xlabel('Grazing Angle (deg)')  
ylabel('NRCS (dB m2/m2)')  
title('NRL Model, Vertical Polarization')
```



### Create Sea Surface Reflectivity Model with Default Values

Create a sea surface normalized reflectivity object using the default model parameters. Obtain the normalized reflectivity at a frequency of 1 GHz over grazing angles from 0.1 to 10 degrees and assume vertical polarization. Plot the normalized reflectivity as a function of grazing angle.

```
grazAng = 0.1:0.1:10;  
freq = 1e9;  
refl = surfaceReflectivitySea  
  
refl =  
    surfaceReflectivitySea with properties:  
  
        Model: 'NRL'  
        SeaState: 1  
        Polarization: 'H'  
        Speckle: 'None'  
  
nrCS = refl(grazAng,freq);  
plot(grazAng,pow2db(nrCS))  
grid on  
xlabel('Grazing Angle (deg)')  
ylabel('NRCS (dB m^2/m^2)')  
title('NRL Model, Vertical Polarization')
```



### Create Reflective Sea Surface in Radar Scenario

Configure a `radarScenario` to simulate a reflective sea surface. Add a sea surface object to define the physical properties of the scenario surface. The surface is a simple 400-by-400 meter rectangle. Use the `surfaceReflectivitySea` function to create a GIT model with a sea state 3. Then, use the scenario `seaSurface` method to add the rectangular sea region and the radar reflectivity model to the scenario. Use a surface reference height of 16 meters.

```
scene = radarScenario(UpdateRate = 0, IsEarthCentered = false);  
refl = surfaceReflectivitySea(Model = "GIT", SeaState = 3, Polarization = "V");  
srf = seaSurface(scene,RadarReflectivity = refl, ...  
    Boundary=[-200 200; -200 200],ReferenceHeight = 16)
```

```
srf =  
    SeaSurface with properties:  
  
        WindSpeed: 10  
        WindDirection: 0  
        Fetch: Inf  
        SpectralModel: []  
        RadarReflectivity: [1x1 surfaceReflectivitySea]  
        ReflectivityMap: 1  
        ReferenceHeight: 16  
        Boundary: [2x2 double]
```



## More About

### Sea Reflectivity Models

Model	Type	Grazing Angles	Frequency Range	Sea State	Settable Properties
'NRL' - Sea Clutter Model due to Gregers-Hansen and Mittal. (Default model) <ul style="list-style-type: none"> <li>• Naval Research Laboratory empirical model for sea reflectivity.</li> <li>• The model does not include variation with azimuth or wind direction.</li> <li>• The model matches experimental results with an absolute deviation of about 2.2 to 2.3 dB for grazing angles from 0.1° to 10°. A deviation of 2.6 dB can be seen for grazing angles above 10° and below 60°.</li> </ul> See [1] and [2].	Empirical	0.1 - 60	0.5 - 35	0 - 6	SeaState Polarization Gamma Speckle SpeckleMean SpeckleStandardDeviation SpeckleScale SpeckleShape

Model	Type	Grazing Angles	Frequency Range	Sea State	Settable Properties
'APL' - <ul style="list-style-type: none"> <li>• John Hopkins Applied Physics Laboratory ADSAM model.</li> <li>• Derived wind velocity from sea state produces less conservative reflectivity values than GIT model at lower sea states.</li> <li>• Takes into account wave height and wave speed.</li> <li>• Differs from the GIT model by deriving wind velocity from sea state.</li> </ul> See [3].	Semi-empirical	0.1 - 10	1 - 100	1 - 6	SeaState Polarization Gamma Speckle SpeckleMean SpeckleStandardDeviation SpeckleScale SpeckleShape

Model	Type	Grazing Angles	Frequency Range	Sea State	Settable Properties
'GIT' - <ul style="list-style-type: none"> <li>Georgia Institute of Technology</li> <li>Semi-empirical model based on multipath, wind speed, and wind direction factor.</li> <li>Takes into account wave height and wave speed.</li> <li>Derived wind velocity from sea state produces less conservative reflectivity values than GIT at lower sea states.</li> </ul> See [4], [5], and [1].	Semi-empirical	0.1 - 10	1 - 100	1 - 6	SeaState Polarization Gamma Speckle SpeckleMean SpeckleStandardDeviation SpeckleScale SpeckleShape

Model	Type	Grazing Angles	Frequency Range	Sea State	Settable Properties
'Hybrid' - <ul style="list-style-type: none"> <li>Hybrid model that mixes work by Barton, Nathanson's tables, and GIT semi-empirical models.</li> <li>May be biased high in the low grazing angle regime.</li> </ul> See [5].	Semi-empirical	0.1 - 30	0.5 - 35	0 - 5	SeaState Polarization Gamma Speckle SpeckleMean SpeckleStandardDeviation SpeckleScale SpeckleShape
'Masuko' - <ul style="list-style-type: none"> <li>Empirical model applicable for medium grazing angles for X and Ka bands.</li> </ul> See [6] and [4].	Empirical	30 - 60	X (8 - 12) Ka (26.5 - 40)	1 - 6	SeaState Polarization Gamma Speckle SpeckleMean SpeckleStandardDeviation SpeckleScale SpeckleShape

Model	Type	Grazing Angles	Frequency Range	Sea State	Settable Properties
'Nathanson' - <ul style="list-style-type: none"> <li>Empirical tables compiled from experimental data that are averages of all wind directions covering UHF to Ka.</li> </ul> See [7].	Empirical	0.1 - 60	UHF (0.3 - 1), L (1 - 2), S(2 - 4), C(4 - 8), X(8 - 12), Ku(12 - 18), Ka(32 - 36)	0 - 6	SeaState Polarization Gamma Speckle SpeckleMean SpeckleStandardDeviation SpeckleScale SpeckleShape
'RRE' - <ul style="list-style-type: none"> <li>Royal Radar Establishment model</li> <li>Averages over all wind directions.</li> <li>Used extensively in the UK for airborne radar performance assessment. See [4].</li> </ul>	Mathematical	< 10	9 - 10	0 - 6	SeaState Polarization Gamma Speckle SpeckleMean SpeckleStandardDeviation SpeckleScale SpeckleShape
'Sittrop' - <ul style="list-style-type: none"> <li>Empirical model for lower grazing angles and higher sea states for X-band.</li> </ul> See [4].	Empirical	0.2 - 10	X (8 - 12)	0 - 7	SeaState Polarization Gamma Speckle SpeckleMean SpeckleStandardDeviation SpeckleScale SpeckleShape

Model	Type	Grazing Angles	Frequency Range	Sea State	Settable Properties
'TSC' - <ul style="list-style-type: none"> <li>• Technology Service Corporation Empirical model.</li> <li>• Based on fit to Nathanson tables.</li> <li>• Similar to the GIT model but with values not falling off as rapidly in range</li> <li>• Recommended for conservative performance prediction or when conditions are unknown.</li> </ul> See [5].	Empirical	0.1 - 90	0.5 - 35	0 - 5	SeaState Polarization Gamma Speckle SpeckleMean SpeckleStandardDeviation SpeckleScale SpeckleShape
'ConstantGamma' - See the "Constant Gamma Model" on page 4-238 for a description.	Mathematical				SeaState Gamma Speckle SpeckleMean SpeckleStandardDeviation SpeckleScale SpeckleShape

**Constant Gamma Model**

The constant-gamma model expresses a simple analytic relationship between the normalized radar cross-section and grazing angle.

$$\sigma = 10^{(\gamma/10)} \sin(\theta)$$

$\gamma$  is defined by the Gamma property and  $\theta$  is the grazing angle input argument `graz` given in degrees. The default value of Gamma is -40, which is representative of sea state 3.

### Speckle Model

Speckle is modeled as an uncorrelated, multiplicative factor  $I = \sigma \cdot n$ , where  $\sigma$  represents the clutter RCS and  $n$  are independent identically distributed (IDD) mean noise samples with unity mean. Because speckle is correlated with underlying terrain RCS, it is usually applied to radar intensity. The speckle noise models include Weibull, Rayleigh, and lognormal.

### References

- [1] Gregers-Hansen, V. and Mittal, R. "An Improved Empirical Model for Radar Sea Clutter Reflectivity." *NRL/MR/5310-12-9346*, Apr. 27, 2012.
- [2] Barton, David Knox. *Radar Equations for Modern Radar*. Artech House, 2013.
- [3] Reilly, J. P., R. L. McDonald, and G. D. Dockery. "RF-Environment Models for the ADSAM Program." Report No. A1A97U-070, Laurel, MD: Johns Hopkins University Applied Physics Laboratory, August 22, 1997.
- [4] Ward, Keith D., Simon Watts, and Robert J. A. Tough. *Sea Clutter: Scattering, the K-Distribution and Radar Performance*. IET Radar, Sonar, Navigation and Avionics Series 20. London: Institution of Engineering and Technology, 2006.
- [5] Antipov, Irina. "Simulation of Sea Clutter Returns." Department of Defence, June 1998.
- [6] Masuko, Harunobu, Ken'ichi Okamoto, Masanobu Shimada, and Shuntaro Niwa. "Measurement of Microwave Backscattering Signatures of the Ocean Surface Using X Band and K a Band Airborne Scatterometers." *Journal of Geophysical Research* 91, no. C11 (1986): 13065. <https://doi.org/10.1029/JC091iC11p13065>.
- [7] Nathanson, Fred E., et al. *Radar Design Principles: Signal Processing and the Environment*. 2. ed., Repr, Scitech Publ, 2004.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`clutterSurfaceRCS` | `landroughness` | `searoughness` | `landreflectivity` | `seareflectivity` | `surfaceReflectivity` | `surfaceReflectivityCustom` | `surfaceReflectivityLand` | `grazingang` | `depressionang` | `radarScenario` | `radarDataGenerator`

### Topics

"Introduction to Radar Scenario Clutter Simulation"  
 "Generate Clutter and Target Returns for MTI Radar"  
 "Simulate Radar Detections of Surface Targets in Clutter"

**Introduced in R2022a**



# surfaceReflectivityCustom

Normalized reflectivity of custom surface

## Description

Normalized reflectivity is the radar cross-section of a unit area of a general surface. Multiplying by the total area of the surface or the illuminated area of a surface gives the total radar cross-section. Normalized reflectivity is also referred to as surface  $\sigma^0$  and is a function of frequency and grazing angle.

To compute the normalized reflectivity:

- 1 Create the `surfaceReflectivityCustom` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
refl = surfaceReflectivityCustom
refl = surfaceReflectivityCustom(Name=Value)
```

### Description

`refl = surfaceReflectivityCustom` creates a normalized reflectivity System object `refl` for a custom surface with default reflectivities. The default custom surface has reflectivity set to  $1 \text{ m}^2/\text{m}^2$  for grazing angles from  $0^\circ$  -  $90^\circ$  for frequencies from 0 -  $1\text{e}20$  Hz.

`refl = surfaceReflectivityCustom(Name=Value)` also creates a normalized reflectivity object for a surface with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as `(Name1=Value1, ..., NameN=ValueN)`.

Example: `refl = surfaceReflectivityCustom(Reflectivity = nrcsTable, Frequency = (0:10)*1e9, GrazingAngle = (0:0.001:2))` creates a custom normalized reflectivity object from a table of reflectivity values `nrcsTable` for frequencies from 0 - 10 GHz and grazing angles from  $0^\circ$  -  $2^\circ$ .

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**Reflectivity — Normalized radar cross section**`ones(91,2)` (default) |  $Q$ -by- $R$  real-valued matrix |  $Q$ -by- $R$ -by- $S$  real-valued array

Normalized radar cross section (NRCS) or reflectivity values, specified as an  $M$ -by- $N$  real-valued matrix. where  $Q$  corresponds to the number of angles set in the `GrazingAngle` property and  $R$  corresponds to the number of frequencies set in the `Frequency` property.

For more than one surface, you can specify an  $Q$ -by- $R$ -by- $S$  real-valued array where  $P$  corresponds to the surface type index.

Units are dimensionless but often expressed in  $\text{m}^2/\text{m}^2$ .

Data Types: `double`

**Frequency — Valid frequencies**`[0 1e20]` (default) | length- $R$  row-vector of real values

Valid frequencies for the normalized reflectivity, specified as a length- $R$  row-vector of real values.  $R$  corresponds to the number of rows in the `Reflectivity` property. Frequency units are in Hz.

Data Types: `double`

**GrazingAngle — Grazing angles**`0:90` (default) | length- $Q$  row-vector of real values

Grazing angles, specified as a length- $Q$  row-vector of real values.  $Q$  corresponds to the number of columns in the `Reflectivity` property. Units are in degrees.

Data Types: `double`

**Speckle — Speckle distribution type**`'None'` (default) | `'Lognormal'` | `'Rayleigh'` | `'Weibull'` | `'Custom'`

Speckle distribution type, specified as `'None'`, `'Lognormal'`, `'Rayleigh'`, `'Weibull'`, or `'Custom'`. Speckle is a multiplicative factor used to make clutter data appear noisier and is especially applicable to imaging applications.

Speckle is correlated with clutter RCS and is applied as  $I = \sigma * n$ , where  $\sigma$  represents the clutter RCS and  $n$  represents random numbers, which are often drawn from an independent identically-distributed unity mean noise statistical distribution.

- `None` - No speckle is applied.
- `Lognormal` - Speckle has a lognormal distribution. Define the distribution using the `SpeckleMean` and `SpeckleStandardDeviation` properties. Default values of these properties create speckle with a normalized mean lognormal distribution.
- `Rayleigh` - Speckle has a Rayleigh distribution. Define the distribution using the `SpeckleScale` property. The default value of this property creates speckle with a unit mean Rayleigh distribution.
- `Weibull` - Speckle has a Weibull distribution. Define the distribution using the `SpeckleScale` and `SpeckleShape` properties. The default values of these properties create speckle with a unit mean Rayleigh distribution.

Data Types: `char` | `string`

**SpeckleMean — Mean of value of lognormal-distributed speckle**`-0.5*log(2)` (default) | scalar

Mean value of lognormal-distributed speckle, specified as a scalar.

#### Dependencies

To enable this property, set the `Speckle` property to 'Lognormal'.

Data Types: double

#### SpeckleStandardDeviation — Standard deviation of lognormal-distributed speckle

$\sqrt{\log(2)}$  (default) | non-negative scalar

Standard deviation of lognormal-distributed speckle, specified as a non-negative scalar.

#### Dependencies

To enable this property, set the `Speckle` property to 'Lognormal'.

Data Types: double

#### SpeckleScale — Scale parameter for Weibull and Rayleigh speckle distribution

$\sqrt{4/\pi}$  (default) | non-negative scalar

Scale parameter for speckle for the Rayleigh and Weibull distributions, specified as a positive scalar.

#### Dependencies

To enable this property, set the `Speckle` property to 'Rayleigh' or 'Weibull'.

Data Types: double

#### SpeckleShape — Shape value for Weibull distribution

2 (default) | positive scalar

Shape value for the Weibull speckle distribution, specified as a positive scalar.

#### Dependencies

To enable this property, set the `Speckle` property to Weibull.

Data Types: double

## Usage

### Syntax

```
nrcs = refl(graz, freq)
nrcs = refl(graz, freq, idx)
[nrcs, speck] = relf(____)
```

### Description

`nrcs = refl(graz, freq)` returns the normalized radar cross section `nrcs` at grazing angle `graz` and frequency `freq`. When either `graz` or `freq` lies outside of the valid region defined by the `GrazingAngle` and `Frequency` properties, the nearest value of the normalized reflectivity is returned.

`nrcs = refl(graz, freq, idx)` also specifies the surface type index `idx` of the surface patch. To enable this syntax, specify the `Reflectivity` property as an  $M$ -by- $N$ -by- $P$  array, where  $M$

corresponds to the number of angles specified in the `GrazingAngle` property,  $N$  corresponds to the number of frequencies in the `Frequency` property, and  $P$  corresponds to the surface type index. MATLAB array.

`[nracs, speck] = relf( ___ )` also returns speckle values `speck`.

### **Input Arguments**

#### **graz — Grazing angle**

nonnegative scalar |  $M$ -length vector of nonnegative values

Grazing angle of the surface relative to the radar, specified as a scalar or a  $M$ -length row vector of nonnegative values. The angles range from  $0^\circ$  to  $90^\circ$ . Units are in degrees.

Example: 10

#### **freq — Transmitted frequencies**

$10e9$  (default) | positive scalar |  $N$ -length vector of positive values

Transmitted frequencies, specified as a positive scalar or  $N$ -length vector of positive values. Units are in Hz.

Example:  $7*10e9$

#### **idx — Surface type index**

scalar | length- $P$  vector of positive values

Surface type index, specified as a scalar or length- $P$  vector of positive values.

Data Types: `double`

### **Output Arguments**

#### **nracs — Normalized surface reflectivity**

real-valued  $N$ -length row vector | real-valued  $M$ -by- $N$  matrix

Normalized surface reflectivity, returned as either a real-valued  $N$ -length row vector or a real-valued  $M$ -by- $N$  matrix. Normalized reflectivity is also called normalized radar cross section.  $M$  is the length of the grazing angle or depression angle vector `graz` and  $N$  is the length of the frequency vector `freq`. `nracs` is dimensionless but often expressed as  $m^2/m^2$ .

#### **speck — Multiplicative speckle values**

length- $N$  vector of real values

Multiple speckle values, returned as a length- $N$  vector of real values.

Data Types: `double`

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

step     Run System object algorithm  
 release   Release resources and allow changes to System object property values and input characteristics  
 reset     Reset internal states of System object

## Examples

### Compute Custom Normalized Reflectivity as Standalone Function

Construct a table of normalized reflectivities of a land surface using the `surfacegamma` function. The table covers frequencies from 1 - 10 GHz and grazing angles from 0 - 7 degrees. Use the `surfaceReflectivityCustom` System object™ directly.

```
freqs = (1:10)*1e9;
angs = 0:.1:7;
gammaFarm = db2pow(surfacegamma('farmland',freqs));
gammaHills = db2pow(surfacegamma('wooded hill',freqs));
```

Create a constant gamma model by multiplying the reflectivity coefficients by the sine of the grazing angle. Add Rayleigh speckle.

```
nrcsTbl = zeros(numel(angs),numel(freqs),2);
nrcsTbl(:,:,1) = gammaFarm.*sind(angs).'; % Farmland
nrcsTbl(:,:,2) = gammaHills.*sind(angs).'; % Wooded hills
refl = surfaceReflectivityCustom(Reflectivity = nrcsTbl, Frequency = freqs, ...
    GrazingAngle = angs, Speckle = 'Rayleigh');
```

Find the normalized reflectivity of farm land (in dB).

```
nrcs = pow2db(refl(6.3, 2.5e9, 1));
disp(nrcs)

-27.2110
```

Find the normalized reflectivity of wooded hills (in dB).

```
nrcs = pow2db(refl(6.3, 2.5e9, 2));
disp(nrcs)

-22.2110
```

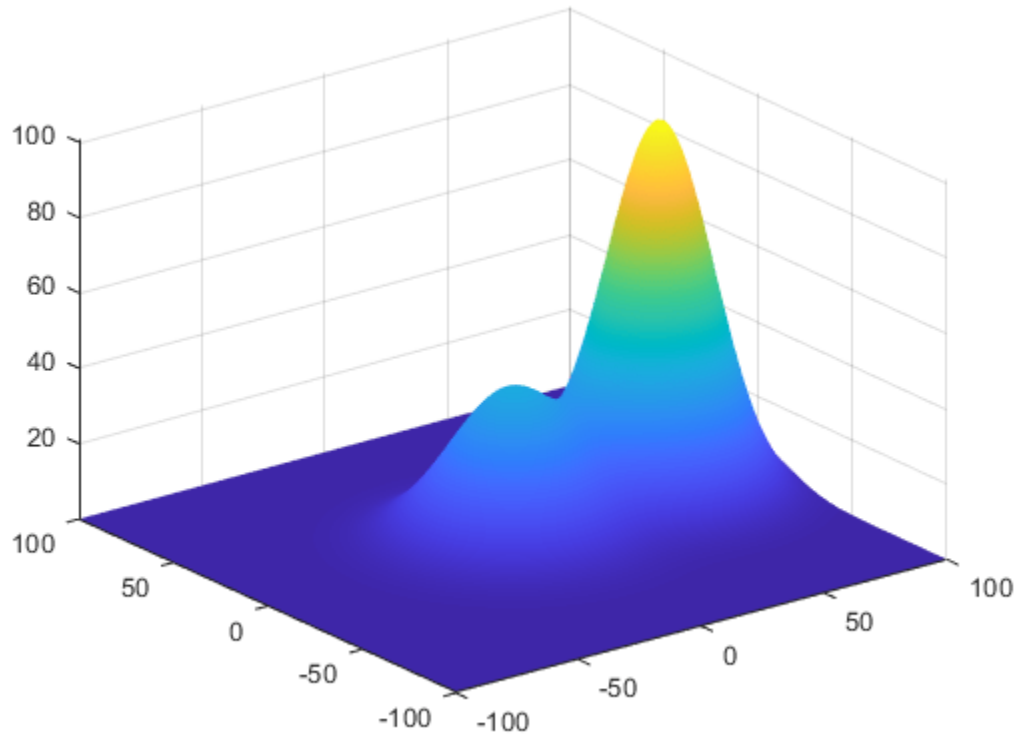
### Create Custom Surface in Radar Scenario

Create a land reflectivity model using the `surfaceReflectivityCustom` object and `radarScenario`.

First create a two hill scenario.

```
[x,y] = meshgrid(linspace(-100,100,201));
ht1 = 40*exp(-(x.^2 + y.^2)/30^2);
ht2 = 100*exp(-((x-60).^2 + y.^2)/25^2);
ht = ht1 + ht2;
```

```
p = surfc(x(1,:),y(:,1),ht);
shading interp
```



Construct a table of normalized reflectivities of a land surface using the `surfacegamma` function. The table covers frequencies from 1 - 10 GHz and grazing angles from 0 - 7 degrees.

```
freqs = (1:10)*1e9;
angs = 0:.1:7;
gammaFarm = db2pow(surfacegamma('farmland',freqs));
gammaHills = db2pow(surfacegamma('wooded hill',freqs));
```

Create a constant gamma model by multiplying the reflectivity coefficients by the sine of the grazing angle. Add Rayleigh speckle.

```
nrcsTbl = zeros(numel(angs),numel(freqs),2);
nrcsTbl(:,:,1) = gammaFarm.*sind(angs).'; % Farmland
nrcsTbl(:,:,2) = gammaHills.*sind(angs).'; % Wooded hills
simTime = 3;
scene = radarScenario(UpdateRate = 1, ...
    IsEarthCentered = false,StopTime = simTime);
refl = surfaceReflectivityCustom(Reflectivity = nrcsTbl, Frequency = freqs, ...
    GrazingAngle = angs, Speckle = 'Rayleigh');
srf = landSurface(scene,RadarReflectivity = refl, ...
    Terrain = ht,Boundary = [-100,100;-100,100]);
```

Find the normalized reflectivity of farm land (in dB).

```
nrcs = pow2db( refl(6.3, 2.5e9, 1) );  
disp(nrcs)
```

```
-27.2110
```

Find the normalized reflectivity of wooded hills (in dB).

```
nrcs = pow2db( refl(6.3, 2.5e9, 2) );  
disp(nrcs)
```

```
-22.2110
```

## More About

### Speckle Model

Speckle is modeled as an uncorrelated, multiplicative factor  $I = \sigma \cdot n$ , where  $\sigma$  represents the clutter RCS and  $n$  are independent identically distributed (IDD) mean noise samples with unity mean. Because speckle is correlated with underlying terrain RCS, it is usually applied to radar intensity. The speckle noise models include Weibull, Rayleigh, and lognormal.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

[clutterSurfaceRCS](#) | [landroughness](#) | [searoughness](#) | [surfacegamma](#) | [landreflectivity](#) | [seareflectivity](#) | [surfaceReflectivity](#) | [surfaceReflectivityLand](#) | [surfaceReflectivitySea](#) | [radarScenario](#) | [radarDataGenerator](#)

### Topics

“Introduction to Radar Scenario Clutter Simulation”  
“Generate Clutter and Target Returns for MTI Radar”  
“Simulate Radar Detections of Surface Targets in Clutter”

### Introduced in R2022a

# surfaceReflectivity

Normalized reflectivity of surface

## Description

The `surfaceReflectivity` System object creates a common interface for the `surfaceReflectivityLand`, `surfaceReflectivityCustom`, and `surfaceReflectivitySea` System objects.

To compute the normalized reflectivity:

- 1 Create the `surfaceReflectivity` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
refl = surfaceReflectivity
refl = surfaceReflectivity(surfacetype)
refl = surfaceReflectivity(surfacetype,Name=Value)
```

### Description

`refl = surfaceReflectivity` creates a normalized reflectivity object, `refl`, for a land surface. Use this object to generate a normalized radar cross section (NRCS) (also called surface  $\sigma^0$ ) as a function of frequency and grazing angle. This syntax creates a constant gamma land reflectivity model with a constant gamma value of -20 dB, which is representative of flatland.

`refl = surfaceReflectivity(surfacetype)` creates a normalized reflectivity object, `refl`, for a `surfacetype` specified as one of 'Land', 'Sea', or 'Custom'. Use this object to generate a normalized radar cross section (NRCS) (also called surface  $\sigma^0$ ) as a function of frequency and grazing angle.

`refl = surfaceReflectivity(surfacetype,Name=Value)` also creates a normalized reflectivity object for surface type `surfacetype` with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as `(Name1=Value1,...,NameN=ValueN)`.

Example: `refl = surfaceReflectivity('Land',Model="GIT",LandType="Soil",SurfaceHeightStandardDeviation=1)` creates a normalized reflectivity object for land using the GIT model with a `LandType` of `Soil` and a `SurfaceHeightStandardDeviation` of 1.



## Usage

### Syntax

```
nrcs = refl(graz, freq)
```

### Description

`nrcs = refl(graz, freq)` returns the normalized radar cross section `nrcs` at grazing angle `graz` and frequency `freq`.

### Input Arguments

#### **graz** — Grazing or depression angle

scalar |  $M$ -length vector of real values

Grazing or depression angle of a surface relative to the radar, specified as a scalar or an  $M$ -length row vector of real values. When the land `Model` property is set to 'Billingsley', the angle is interpreted as a depression angle `depressionang` between  $-90^\circ$  and  $90^\circ$ . For all other models, the angle is interpreted as a grazing angle `grazingang` ranging from  $0^\circ$  to  $90^\circ$ . Units are in degrees.

#### **freq** — Transmitted frequencies

$10e9$  (default) | positive scalar |  $N$ -length vector of positive values

Transmitted frequencies, specified as a positive scalar or  $N$ -length vector of positive values. Units are in Hz.

Example: `freq = 7*10e9`

### Output Arguments

#### **nrcs** — Normalized surface reflectivity

real-valued  $N$ -length row vector | real-valued  $M$ -by- $N$  matrix

Normalized surface reflectivity, returned as either a real-valued  $N$ -length row vector or a real-valued  $M$ -by- $N$  matrix. Normalized reflectivity is also called normalized radar cross section.  $M$  is the length of the grazing angle or depression angle vector `graz` and  $N$  is the length of the frequency vector `freq`. `nrcs` is dimensionless but often expressed as  $m^2/m^2$ .

## Object Functions

To use an object function, specify the `System` object as the first input argument. For example, to release system resources of a `System` object named `obj`, use this syntax:

```
release(obj)
```

## Common to All System Objects

<code>step</code>	Run <code>System</code> object algorithm
<code>release</code>	Release resources and allow changes to <code>System</code> object property values and input characteristics
<code>reset</code>	Reset internal states of <code>System</code> object

## Examples

### Constant Gamma Normalized Reflectivity

Obtain the constant gamma normalized reflectivity for using the default gamma value of -20 dB at a frequency of 10 GHz and a grazing angle of 10 degrees.

```
grazAng = 10;
freq = 10e9;
refl = surfaceReflectivity

refl =
    surfaceReflectivityLand with properties:

        Model: 'ConstantGamma'
        Gamma: -20
        Speckle: 'None'

nracs = refl(grazAng,freq)

nracs = 0.0017
```

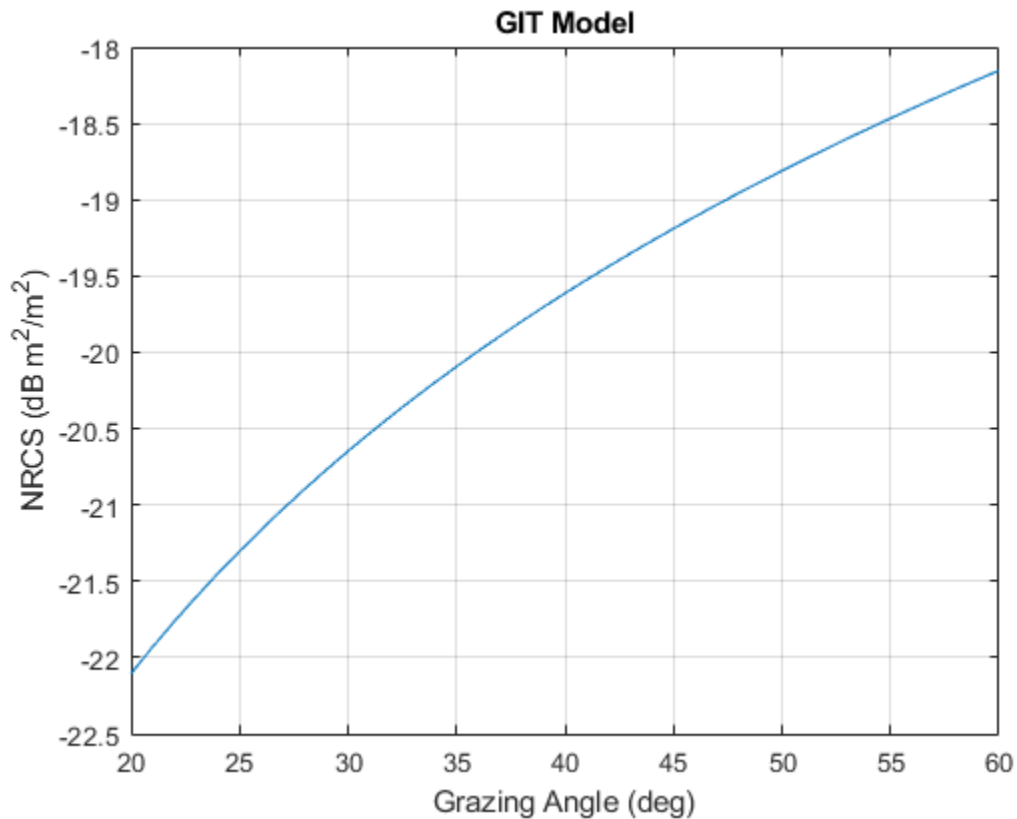
### Obtain Normalized Radar Cross-Section From GIT Model

Create a normalized reflectivity cross-section object for a land surface using the GIT model and a soil land type. Obtain the NRCS at a frequency of 10 GHz over grazing angles from 20 to 60 degrees. Assume a standard deviation of surface height of 1 m.

```
grazAng = 20:60;
freq = 10e9;
refl = surfaceReflectivity('Land','Model','GIT','LandType','Soil', ...
    'SurfaceHeightStandardDeviation',1);
nracs = refl(grazAng,freq);
```

Plot normalized reflectivities for grazing angles from 20 to 60 degrees.

```
plot(grazAng,pow2db(nracs))
grid on
xlabel('Grazing Angle (deg)')
ylabel('NRCS (dB m^2/m^2)')
title('GIT Model')
```



### Sea Surface Normalized Reflectivity

Create a sea normalized reflectivity object using the default NRL model and a sea state of 2. Obtain the normalized reflectivity at 10 GHz over grazing angles from 0.1 to 10 degrees. Assume horizontal polarization.

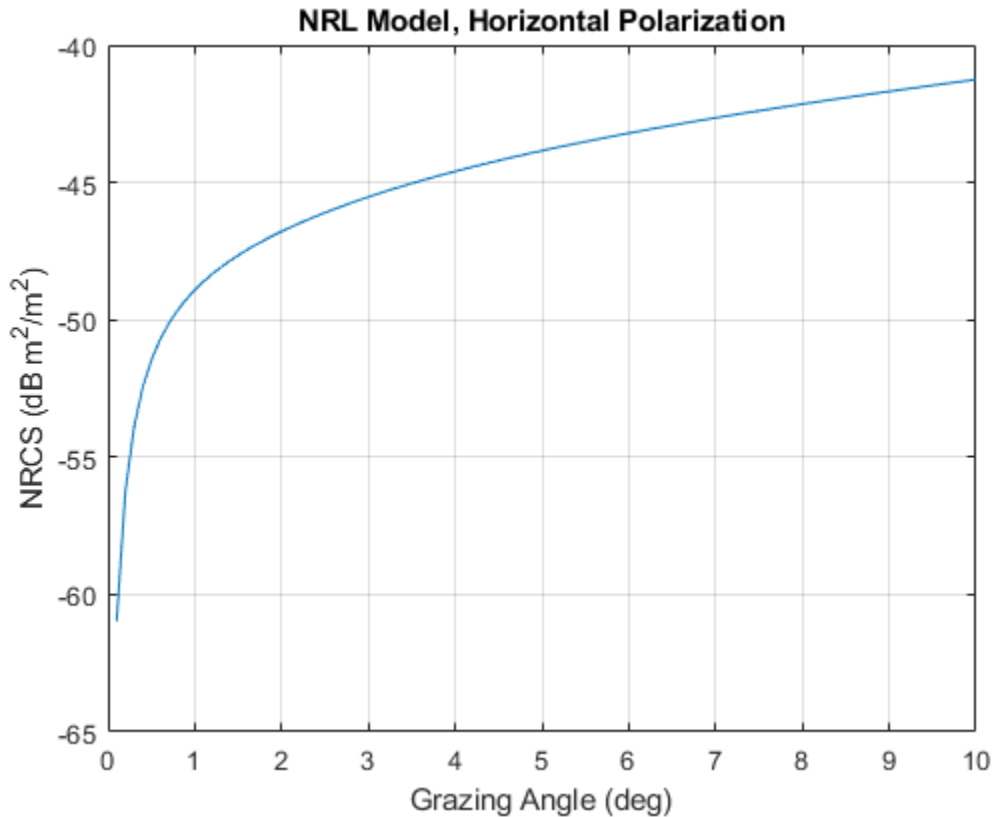
```
grazAng = 0.1:0.1:10;
freq = 10e9;
ss = 2;
pol = 'H';
```

Use the `surfaceReflectivity` object to obtain the normalized reflectivity.

```
refl = surfaceReflectivity('Sea',SeaState = ss,Polarization = pol);
nrCS = refl(grazAng,freq);
```

Plot the reflectivity as a function of grazing angle.

```
plot(grazAng,pow2db(nrCS))
grid on
xlabel('Grazing Angle (deg)')
ylabel('NRCS (dB m^2/m^2)')
title('NRL Model, Horizontal Polarization')
```



### Nathanson Reflectivity Model

Define a custom NRCS table using Nathanson reflectivity values for farmlands. Assume Rayleigh speckle. Next, calculate the RCS of a clutter patch and estimate the clutter-to-noise ratio at the receiver. Assume that the patch is 1000 meters away from the radar system. The azimuth and elevation beamwidths are 1 degree and 3 degrees, respectively. The grazing angle is 10 degrees. The pulse width is 10 microseconds. The radar operates at an L-band frequency of 1.5 GHz with a peak power of 5 kw. Use the general `surfaceReflectivity` object.

```
rng      = 1000;
bwAz     = 1;
bwEl     = 3;
graz     = 10;
tau      = 10e-6;
freq     = 1.5e9;
ppow     = 5000;
```

Configure a custom surface.

```
nathansonNRCS = db2pow([-35 -33 -32; -31 -30 -29; -29 -27 -25; ...
    -19 -17 -15; -14 -15 -14]);
nathansonFreq  = [1.5 3 6]*1e9;
nathansonGrazAng = [1.5 3 10 30 60];
refl = surfaceReflectivity('Custom',Reflectivity = nathansonNRCS, ...
```

```

    Frequency = nathansonFreq,GrazingAngle = nathansonGrazAng, ...
    Speckle = "Rayleigh")

refl =
    surfaceReflectivityCustom with properties:

        Reflectivity: [5x3 double]
        Frequency: [1.5000e+09 3.0000e+09 6.0000e+09]
        GrazingAngle: [1.5000 3 10 30 60]
        Speckle: 'Rayleigh'
        SpeckleScale: 1.1284

[nrcs,n] = refl(graz,freq)

nrcs = 0.0013

n = 1.6210

Calculate the clutter RCS and apply multiplicative speckle

sigma = clutterSurfaceRCS(nrcs,rng,bwAz,bwEl,graz,tau)

sigma = 6.6253

rcs = sigma.*n

rcs = 10.7394

Calculate clutter-to-noise ratio.

lambda = freq2wavelen(freq);
cnr = radareqsnr(lambda,rng,ppow,tau,'rcs',rcs)

cnr = 74.3130

```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

clutterSurfaceRCS | landroughness | searoughness | landreflectivity | seareflectivity | surfaceReflectivityLand | surfaceReflectivityCustom | surfaceReflectivitySea | grazingang | depressionang | radarScenario | radarDataGenerator

### Topics

“Introduction to Radar Scenario Clutter Simulation”  
 “Generate Clutter and Target Returns for MTI Radar”  
 “Simulate Radar Detections of Surface Targets in Clutter”

**Introduced in R2022a**

# LandSurface

Land surface belonging to radar scenario

## Description

LandSurface defines a land surface object belonging to a radarScenario object. You can use the LandSurface object to determine land heights in a scenario and surface reflectivity, test for occlusion along the line-of-sight between two points in the scenario and give surface height at a point.

## Creation

Create LandSurface objects using the `landSurface` object function of the radarScenario object.

## Properties

### RadarReflectivity — Radar reflectivity object of surface

`surfaceReflectivityLand` object (default) | `surfaceReflectivityCustom` object

Radar reflectivity object of surface, specified as a `surfaceReflectivityLand` or `surfaceReflectivityCustom` System object for the normalized radar cross section (NRCS) of the surface. Defaults to a `surfaceReflectivityLand` object using a Barton land model and flatland land type.

Example:

```
surfaceReflectivityLand(Model="GIT",LandType="Soil",SurfaceHeightStandardDeviation=1)
```

### ReflectivityMap — Map of reflectivity type indices over the surface

1 (default) | scalar | real-valued matrix

This property contains a grid of reflectivity type values corresponding to vertices of the surface height data. If any terrain or a spectral model is present, `ReflectivityMap` must be a matrix of the same size as the domain of that data. Otherwise it must be scalar. Each element is an index into the third dimension of the `Reflectivity` property of the `surfaceReflectivityCustom` object.

### Dependencies

To enable this property, set the `RadarReflectivity` property to a `surfaceReflectivityCustom` object.

Data Types: double

### ReferenceHeight — Surface reference height

0 (default) | scalar

Reference height of surface height data, specified as a scalar. Surface heights are relative to the reference height. For surfaces with no height data, this is the assumed height of the entire surface. Units are in meters.

Data Types: double

**Boundary — Bounding rectangle of surface**

[-Inf Inf; -Inf Inf] (default) | 2-by-2 matrix of real values

Bounding rectangle of the surface, specified as a 2-by-2 matrix of real values. The bounding rectangle is defined by two two-dimensional points in either Cartesian or geodetic scenario coordinates. When the `IsEarthCentered` property of the `radarScenario` object is specified as:

- `false` — Scenario coordinates are Cartesian. Specify the bounding rectangle [`MinX`, `MaxX`, `MinY`, `MaxY`], as Cartesian coordinates in the reference frame of the scenario, where `MinX` < `MaxX`, and `MinY` < `MaxY`.
- `true` — Scenario coordinates are geodetic. Specify the bounding rectangle as [`StartLat`, `EndLat`, `StartLon`, `EndLon`] where `StartLat` and `EndLat` are the minimum and maximum latitudes of the geodetic frames. `StartLat` and `EndLat` must lie in the interval [-90,90] where `StartLat` < `EndLat`.

Specify `StartLon` and `EndLon` as the minimum and maximum longitudes of the geodetic frame. `StartLon` and `EndLon` must lie in the interval [-180,180]. If `EndLon` < `StartLon`, the object wraps `EndLon` to `StartLon` + 360°. Units are in degrees.

Data Types: double

**Terrain — Terrain data for surface**[] (default) | *M*-by-*N* real-valued matrix | string | char array

Terrain data of the surface, specified as an *M*-by-*N* real-valued matrix or a string containing a Digital Terrain Elevation Data (DTED) file name. Terrain data consists of land height as a function of position.

- *M*-by-*N* real-valued matrix — The matrix values represent the height data of an area defined by the `Boundary` property of the surface object. The domain can be a global Cartesian frame in meters or a geodetic grid with units of degrees. The object extends the height data in the matrix to the area. The object automatically fills heights of unspecified points using linear interpolation. *M* or *N* must be greater than or equal to 3.
- DTED file name — To use this option, you must specify the `IsEarthCentered` property of the radar scenario as `true`. In this case, the function uses the DTED file to specify the terrain heights for an area defined by the `Boundary` property of the ground surface object. Also, the object automatically fills unspecified data in the DTED file using linear interpolation.

Height values here are relative to the `ReferenceHeight` property.

Data Types: double | string | char

**Object Functions**

`height`      Height of point on surface  
`occlusion`    Test for occlusion of point by a surface

**Examples**

### Create Land Surface in Radar Scenario

Create a surface with two hills. Plot the surface on a 200-by-200 meter grid with grid points one meter apart. Add the surface to a radar scenario. Assume the surface has a radar reflectivity defined by a constant gamma model.

```
[x,y] = meshgrid(linspace(-100,100,201));
ht1 = 40*exp(-(x.^2 + y.^2)/30^2);
ht2 = 100*exp(-((x-60).^2 + y.^2)/25^2);
ht = ht1 + ht2;
p = surfc(x(1,:),y(:,1),ht);
axis equal
axis tight
shading interp
simTime = 3;
scene = radarScenario(UpdateRate = 1, ...
    IsEarthCentered = false,StopTime = simTime);
gammaDB = surfaceGamma('Flatland');
refl = surfaceReflectivityLand(Model = 'ConstantGamma',Gamma = gammaDB);
srf = landSurface(scene,RadarReflectivity = refl, ...
    Terrain = ht,Boundary = [-100,100;-100,100]);
```

Use surface manager to identify the surface.

```
scene.SurfaceManager

ans =
    SurfaceManager with properties:

        UseOcclusion: 1
        Surfaces: [1x1 radar.scenario.LandSurface]
```

```
scene.SurfaceManager.Surfaces

ans =
    LandSurface with properties:

        RadarReflectivity: [1x1 surfaceReflectivityLand]
        ReflectivityMap: 1
        ReferenceHeight: 0
        Boundary: [2x2 double]
        Terrain: [201x201 double]
```

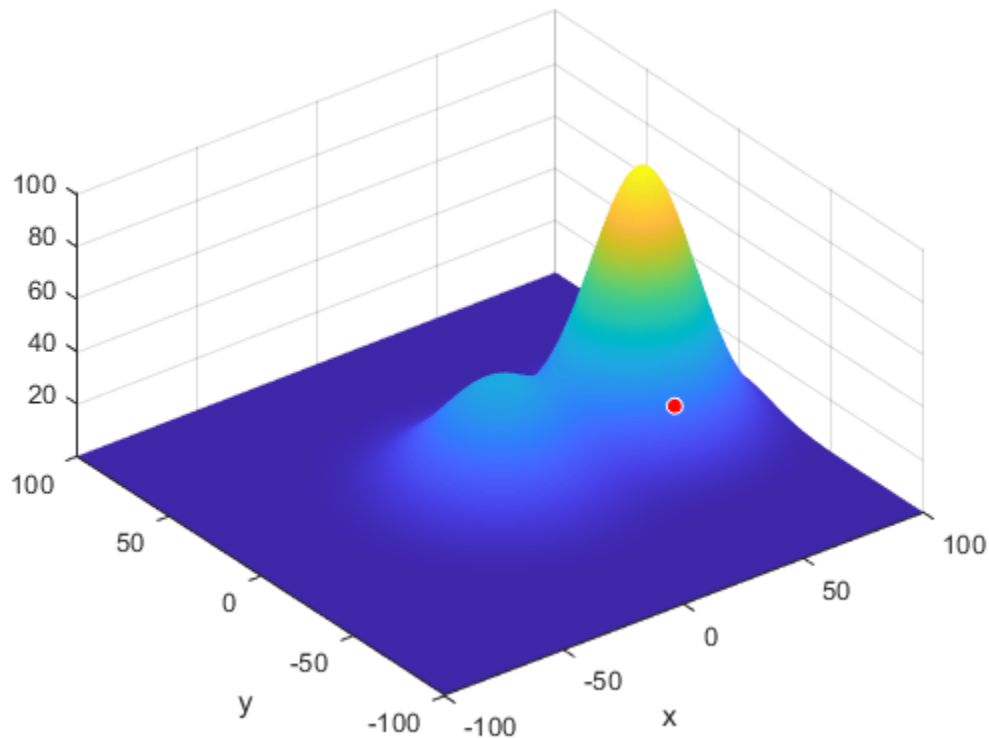
Obtain and plot the height of the surface at the point (50,-30).

```
xt = 50;
yt = -30;
htx = height(srf,[xt,yt])

htx = 21.1046

hold on
plot3(xt,yt,htx+5,'ow','MarkerFaceColor','r')
xlabel('x')
ylabel('y')
hold off
```





### Create Land Surface from DTED File

Create a radar scenario and specify its `IsEarthCentered` property as `true` to use DTED file.

```
scene = radarScenario(IsEarthCentered = true);
```

Model the reflectivity as a constant gamma surface.

```
refl = surfaceReflectivityLand(Model = 'ConstantGamma',Gamma = -20);
```

Add a 0.1-by-0.1 degree land surface derived from a DTED file.

```
bdry = [39.5 39.6;-105.51 -105.41];
srf = landSurface(scene,Terrain = 'n39_w106_3arc_v2.dt1', ...
    Boundary = bdry,RadarReflectivity = refl)
```

```
srf =
    LandSurface with properties:
```

```
    RadarReflectivity: [1x1 surfaceReflectivityLand]
    ReflectivityMap: 1
    ReferenceHeight: 0
    Boundary: [2x2 double]
    Terrain: 'n39_w106_3arc_v2.dt1'
```

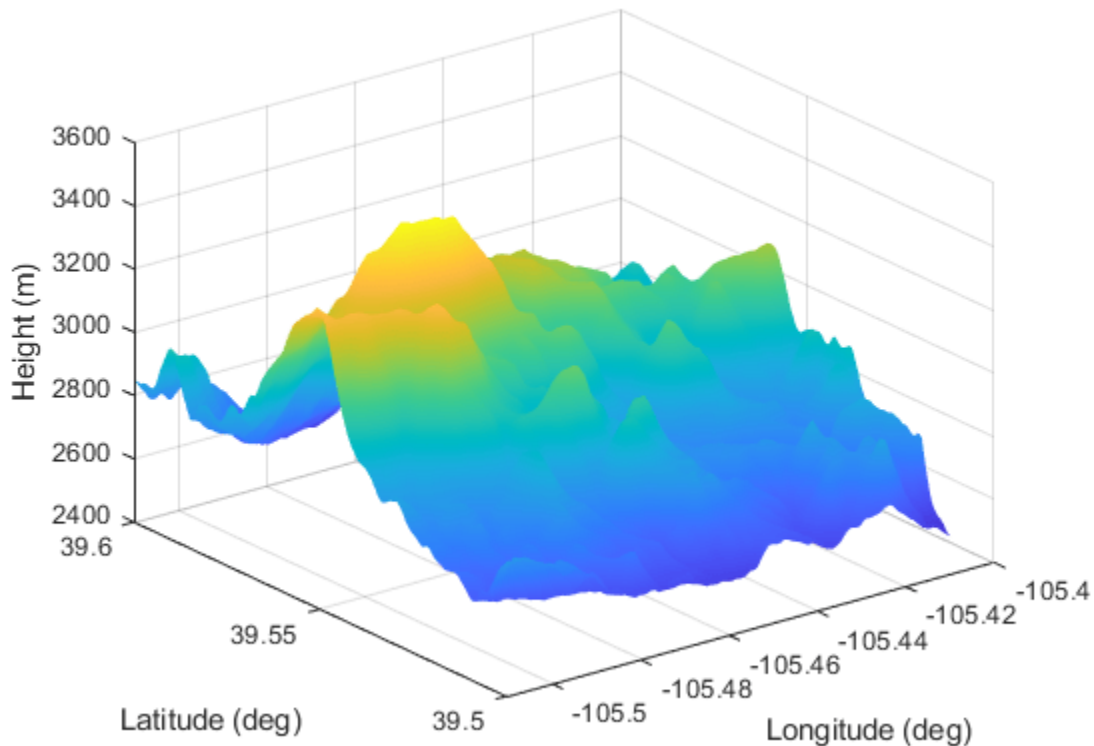
```
mgr = scene.SurfaceManager

mgr =
  SurfaceManager with properties:

    UseOcclusion: 1
    Surfaces: [1x1 radar.scenario.LandSurface]
```

Plot the surface height.

```
x = linspace(srf.Boundary(2,1),srf.Boundary(2,2),201);
y = linspace(srf.Boundary(1,1),srf.Boundary(1,2),201);
[X,Y] = meshgrid(x,y);
X1 = X(:)';
Y1 = Y(:)';
H = height(srf,[Y1;X1]);
H = reshape(H,length(x),length(y));
surf(x,y,H)
shading interp
ylabel('Latitude (deg)')
xlabel('Longitude (deg)')
zlabel('Height (m)')
```



### Test for Occlusion Between Two Points on Land Surface

Create a radar scenario and specify set the `IsEarthCentered` property as true to obtain the terrain from a DTED file.

```
scene = radarScenario(IsEarthCentered = true);
```

Model the reflectivity as a constant gamma surface.

```
refl = surfaceReflectivityLand(Model = 'ConstantGamma',Gamma = -20);
```

Add a 0.1-by-0.1 degree land surface derived from a DTED file.

```
bdry = [39.5 39.6;-105.51 -105.41];
srf = landSurface(scene,Terrain = 'n39_w106_3arc_v2.dtl', ...
    Boundary = bdry,RadarReflectivity = refl);
```

Verify that occlusion is turned on.

```
mgr = scene.SurfaceManager
```

```
mgr =
```

```
SurfaceManager with properties:
```

```
UseOcclusion: 1
Surfaces: [1x1 radar.scenario.LandSurface]
```

Plot the surface height.

```
x = linspace(srf.Boundary(2,1),srf.Boundary(2,2),201);
y = linspace(srf.Boundary(1,1),srf.Boundary(1,2),201);
[X,Y] = meshgrid(x,y);
X1 = X(:)';
Y1 = Y(:)';
H = height(srf,[Y1;X1]);
H = reshape(H,length(x),length(y));
surf(x,y,H)
shading interp
ylabel('Latitude (deg)')
xlabel('Longitude (deg)')
zlabel('Height (m)')
hold on
```

Test for occlusion.

```
ht1 = height(srf,[39.59 -105.5])
```

```
ht1 = 2810
```

```
ht2 = height(srf,[39.51 -105.41])
```

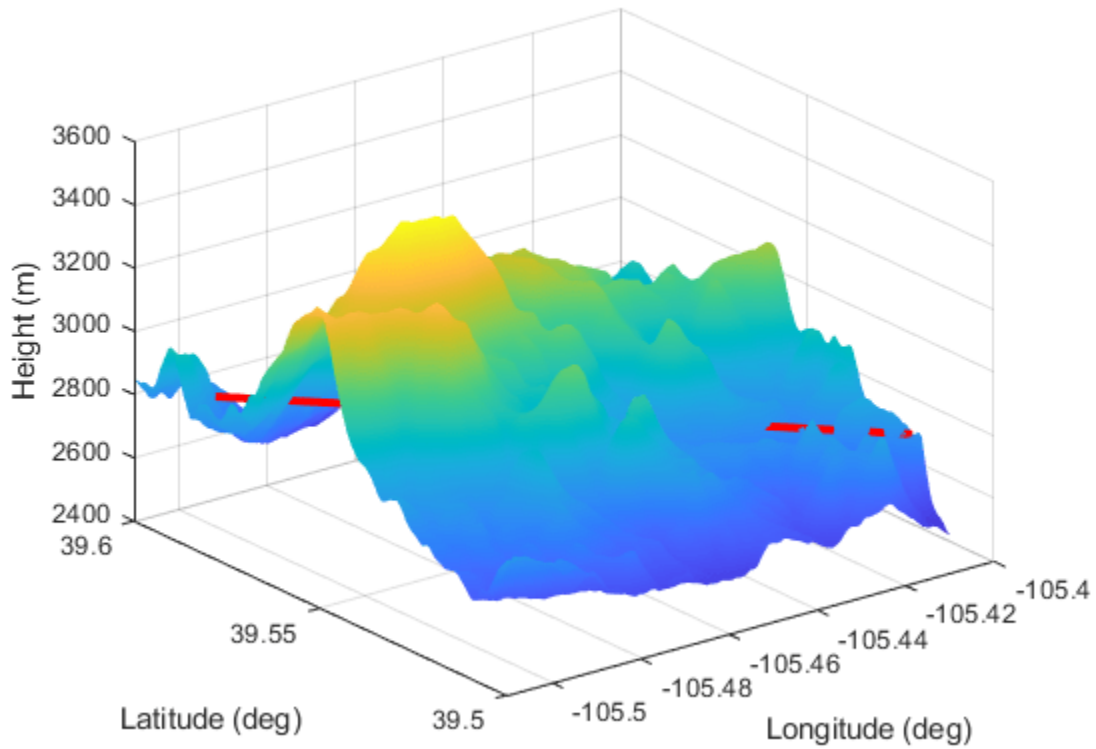
```
ht2 = 2786
```

```
occlusion(srf,[39.59 -105.5 ht1],[39.51 -105.41 ht2])
```

```
ans = logical
     1
```

The points are occluded. The line between the two points passes through the surface as shown.

```
plot3([-105.5 -105.41],[39.59 39.51], [ht1 ht2], 'r', 'LineWidth', 3)
```



### See Also

[height](#) | [occlusion](#) | [radarScenario](#) | [SurfaceManager](#)

### Topics

“Introduction to Radar Scenario Clutter Simulation”

**Introduced in R2022a**

# landSurface

Add land surface to radar scenario

## Syntax

```
srf = landSurface(scene)
srf = landSurface(scene,Name=Value)
```

## Description

`srf = landSurface(scene)` adds a `LandSurface` object `srf` to the radar scenario `radarScenario` `scene`.

`srf = landSurface(scene,Name=Value)` adds a land surface object with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as `(Name1=Value1,...,NameN=ValueN)`.

## Examples

### Create Land Surface in Radar Scenario

Create a surface with two hills. Plot the surface on a 200-by-200 meter grid with grid points one meter apart. Add the surface to a radar scenario. Assume the surface has a radar reflectivity defined by a constant gamma model.

```
[x,y] = meshgrid(linspace(-100,100,201));
ht1 = 40*exp(-(x.^2 + y.^2)/30^2);
ht2 = 100*exp(-((x-60).^2 + y.^2)/25^2);
ht = ht1 + ht2;
p = surfc(x(1,:),y(:,1),ht);
axis equal
axis tight
shading interp
simTime = 3;
scene = radarScenario(UpdateRate = 1, ...
    IsEarthCentered = false,StopTime = simTime);
gammaDB = surfacegamma('Flatland');
refl = surfaceReflectivityLand(Model = 'ConstantGamma',Gamma = gammaDB);
srf = landSurface(scene,RadarReflectivity = refl, ...
    Terrain = ht,Boundary = [-100,100;-100,100]);
```

Use surface manager to identify the surface.

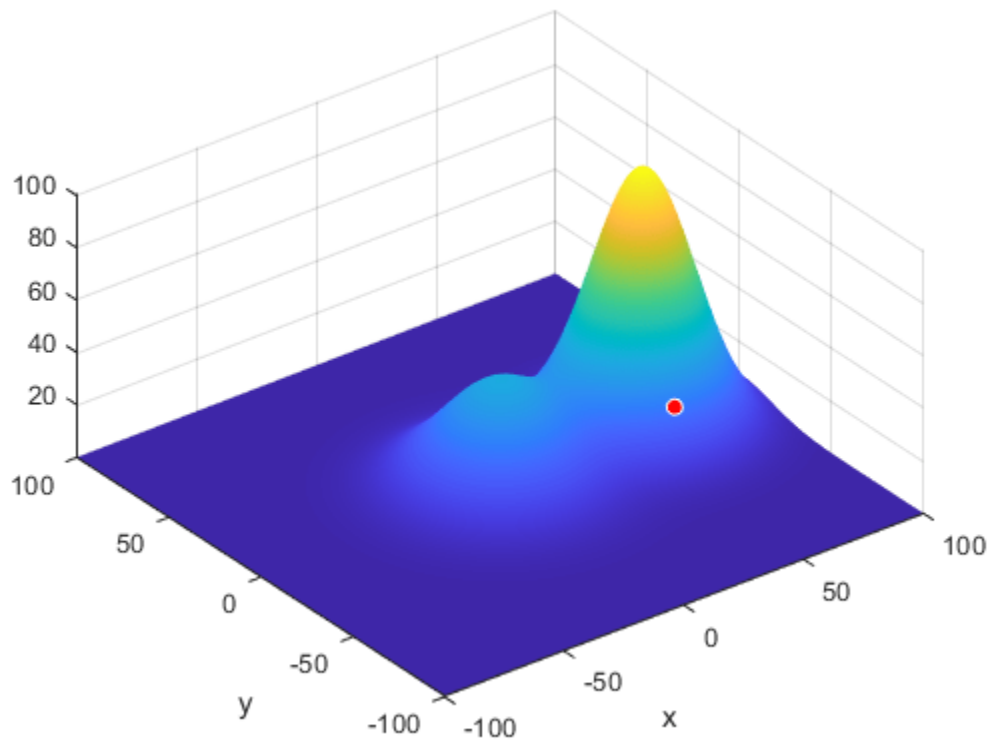
```
scene.SurfaceManager
ans =
    SurfaceManager with properties:
        UseOcclusion: 1
        Surfaces: [1x1 radar.scenario.LandSurface]
```

```
scene.SurfaceManager.Surfaces
```

```
ans =  
LandSurface with properties:  
    RadarReflectivity: [1x1 surfaceReflectivityLand]  
    ReflectivityMap: 1  
    ReferenceHeight: 0  
    Boundary: [2x2 double]  
    Terrain: [201x201 double]
```

Obtain and plot the height of the surface at the point (50,-30).

```
xt = 50;  
yt = -30;  
htx = height(srf,[xt,yt])  
  
htx = 21.1046  
  
hold on  
plot3(xt,yt,htx+5,'ow','MarkerFaceColor','r')  
xlabel('x')  
ylabel('y')  
hold off
```



### Create Land Surface from DTED File

Create a radar scenario and specify its `IsEarthCentered` property as `true` to use DTED file.

```
scene = radarScenario(IsEarthCentered = true);
```

Model the reflectivity as a constant gamma surface.

```
refl = surfaceReflectivityLand(Model = 'ConstantGamma',Gamma = -20);
```

Add a 0.1-by-0.1 degree land surface derived from a DTED file.

```
bdry = [39.5 39.6;-105.51 -105.41];
srf = landSurface(scene,Terrain = 'n39_w106_3arc_v2.dt1', ...
    Boundary = bdry,RadarReflectivity = refl)
```

```
srf =
    LandSurface with properties:

        RadarReflectivity: [1x1 surfaceReflectivityLand]
        ReflectivityMap: 1
        ReferenceHeight: 0
        Boundary: [2x2 double]
        Terrain: 'n39_w106_3arc_v2.dt1'
```

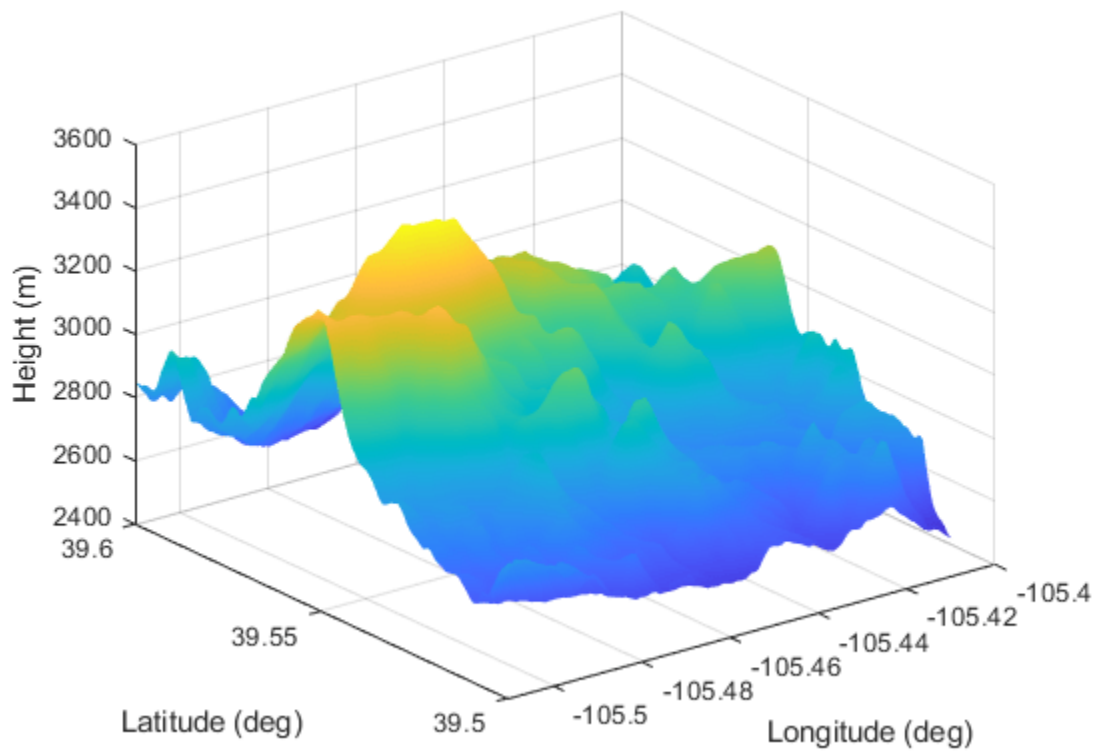
```
mgr = scene.SurfaceManager
```

```
mgr =
    SurfaceManager with properties:

        UseOcclusion: 1
        Surfaces: [1x1 radar.scenario.LandSurface]
```

Plot the surface height.

```
x = linspace(srf.Boundary(2,1),srf.Boundary(2,2),201);
y = linspace(srf.Boundary(1,1),srf.Boundary(1,2),201);
[X,Y] = meshgrid(x,y);
X1 = X(:)';
Y1 = Y(:)';
H = height(srf,[Y1;X1]);
H = reshape(H,length(x),length(y));
surf(x,y,H)
shading interp
ylabel('Latitude (deg)')
xlabel('Longitude (deg)')
zlabel('Height (m)')
```



### Test for Occlusion Between Two Points on Land Surface

Create a radar scenario and specify set the `IsEarthCentered` property as true to obtain the terrain from a DTED file.

```
scene = radarScenario(IsEarthCentered = true);
```

Model the reflectivity as a constant gamma surface.

```
refl = surfaceReflectivityLand(Model = 'ConstantGamma',Gamma = -20);
```

Add a 0.1-by-0.1 degree land surface derived from a DTED file.

```
bdry = [39.5 39.6;-105.51 -105.41];
srf = landSurface(scene,Terrain = 'n39_w106_3arc_v2.dt1', ...
    Boundary = bdry,RadarReflectivity = refl);
```

Verify that occlusion is turned on.

```
mgr = scene.SurfaceManager
```

```
mgr =
    SurfaceManager with properties:
```

```
    UseOcclusion: 1
```



Surfaces: [1x1 radar.scenario.LandSurface]

Plot the surface height.

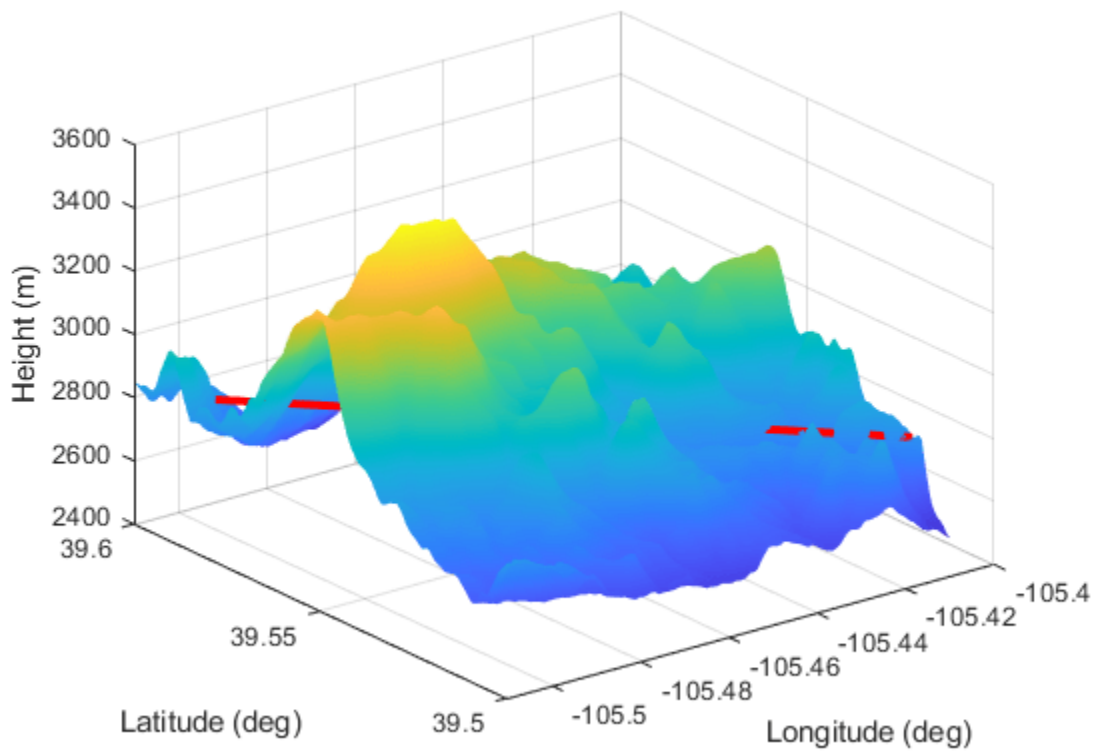
```
x = linspace(srf.Boundary(2,1),srf.Boundary(2,2),201);
y = linspace(srf.Boundary(1,1),srf.Boundary(1,2),201);
[X,Y] = meshgrid(x,y);
X1 = X(:)';
Y1 = Y(:)';
H = height(srf,[Y1;X1]);
H = reshape(H,length(x),length(y));
surf(x,y,H)
shading interp
ylabel('Latitude (deg)')
xlabel('Longitude (deg)')
zlabel('Height (m)')
hold on
```

Test for occlusion.

```
ht1 = height(srf,[39.59 -105.5])
ht1 = 2810
ht2 = height(srf,[39.51 -105.41])
ht2 = 2786
occlusion(srf,[39.59 -105.5 ht1],[39.51 -105.41 ht2])
ans = logical
     1
```

The points are occluded. The line between the two points passes through the surface as shown.

```
plot3([-105.5 -105.41],[39.59 39.51], [ht1 ht2], 'r', 'LineWidth', 3)
```



## Input Arguments

### scene — Radar scenario

radarScenario object

Radar scenario, specified as a radarScenario object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `surface = landSurface(scene, Boundary=[-100,100,-100,100])`

### RadarReflectivity — Radar reflectivity object of surface

surfaceReflectivityLand object (default) | surfaceReflectivityCustom object

Radar reflectivity object of surface, specified as a surfaceReflectivityLand or surfaceReflectivityCustom System object for the normalized radar cross section (NRCS) of the surface. Defaults to a surfaceReflectivityLand object using a Barton land model and flatland land type.

Example:

```
surfaceReflectivityLand(Model="GIT",LandType="Soil",SurfaceHeightStandardDeviation=1)
```

### ReflectivityMap — Map of reflectivity type indices over the surface

1 (default) | scalar | real-valued matrix

This property contains a grid of reflectivity type values corresponding to vertices of the surface height data. If any terrain or a spectral model is present, `ReflectivityMap` must be a matrix of the same size as the domain of that data. Otherwise it must be scalar. Each element is an index into the third dimension of the `Reflectivity` property of the `surfaceReflectivityCustom` object.

#### Dependencies

To enable this property, set the `RadarReflectivity` property to a `surfaceReflectivityCustom` object.

Data Types: double

### ReferenceHeight — Surface reference height

0 (default) | scalar

Reference height of surface height data, specified as a scalar. Surface heights are relative to the reference height. For surfaces with no height data, this is the assumed height of the entire surface. Units are in meters.

Data Types: double

### Boundary — Bounding rectangle of surface

[-Inf Inf; -Inf Inf] (default) | 2-by-2 matrix of real values

Bounding rectangle of the surface, specified as a 2-by-2 matrix of real values. The bounding rectangle is defined by two two-dimensional points in either Cartesian or geodetic scenario coordinates. When the `IsEarthCentered` property of the `radarScenario` object is specified as:

- `false` — Scenario coordinates are Cartesian. Specify the bounding rectangle [`MinX`, `MaxX`, `MinY`, `MaxY`], as Cartesian coordinates in the reference frame of the scenario, where `MinX` < `MaxX`, and `MinY` < `MaxY`.
- `true` — Scenario coordinates are geodetic. Specify the bounding rectangle as [`StartLat`, `EndLat`, `StartLon`, `EndLon`] where `StartLat` and `EndLat` are the minimum and maximum latitudes of the geodetic frames. `StartLat` and `EndLat` must lie in the interval `[-90,90]` where `StartLat` < `EndLat`.

Specify `StartLon` and `EndLon` as the minimum and maximum longitudes of the geodetic frame. `StartLon` and `EndLon` must lie in the interval `[-180,180]`. If `EndLon` < `StartLon`, the object wraps `EndLon` to `StartLon + 360°`. Units are in degrees.

Data Types: double

### Terrain — Terrain data for surface

[] (default) | *M*-by-*N* real-valued matrix | string | char array

Terrain data of the surface, specified as an *M*-by-*N* real-valued matrix or a string containing a Digital Terrain Elevation Data (DTED) file name. Terrain data consists of land height as a function of geolocation.

- *M*-by-*N* real-valued matrix — The matrix values represent the height data of an area defined by the `Boundary` property of the surface object. The domain can be a global Cartesian frame in meters or a geodetic grid with units of degrees. The object extends the height data in the matrix to the area. The object automatically fills heights of unspecified points using linear interpolation. *M* or *N* must be greater than or equal to 3.
- DTED file name — To use this option, you must specify the `IsEarthCentered` property of the radar scenario as `true`. In this case, the function uses the DTED file to specify the terrain heights for an area defined by the `Boundary` property of the ground surface object. Also, the object automatically fills unspecified data in the DTED file using linear interpolation.

Height values here are relative to the `ReferenceHeight` property.

Data Types: `double` | `string` | `char`

## Output Arguments

### **srf** — Land surface

LandSurface object

Land surface, returned as a LandSurface object.

## See Also

`radarScenario` | `LandSurface` | `SurfaceManager` | `height` | `occlusion`

## Topics

“Introduction to Radar Scenario Clutter Simulation”

## Introduced in R2022a

# SeaSurface

Sea surface belonging to radar scenario

## Description

SeaSurface defines a sea surface object belonging to a radarScenario. The object describes the size, surface reflectivity, motion, and wind speed of the surface. You can use the SeaSurface object to determine sea heights in a scenario and surface reflectivity, test for occlusion along the line-of-sight between two points in the scenario and give surface height at a point.

## Creation

Create SeaSurface objects using the seaSurface object function of the radarScenario object.

## Properties

### SpectralModel — Sea surface omnidirectional motion spectrum

seaSpectrum object

Sea surface omnidirectional motion spectrum, specified as a seaSpectrum object. This object models surface heights over time.

### Dependencies

To enable this property, set the radarScenario property IsEarthCentered to false.

### Boundary — Bounding rectangle of surface

`[-Inf Inf; -Inf Inf]` (default) | 2-by-2 matrix of real values

Bounding rectangle of the surface, specified as a 2-by-2 matrix of real values. The bounding rectangle is defined by two two-dimensional points in either Cartesian or geodetic scenario coordinates. When the IsEarthCentered property of the radarScenario object is specified as:

- `false` — Scenario coordinates are Cartesian. Specify the bounding rectangle `[MinX, MaxX, MinY, MaxY]`, as Cartesian coordinates in the reference frame of the scenario, where `MinX < MaxX`, and `MinY < MaxY`.
- `true` — Scenario coordinates are geodetic. Specify the bounding rectangle as `[StartLat, EndLat, StartLon, EndLon]` where `StartLat` and `EndLat` are the minimum and maximum latitudes of the geodetic frames. `StartLat` and `EndLat` must lie in the interval `[-90,90]` where `StartLat < EndLat`.

Specify `StartLon` and `EndLon` as the minimum and maximum longitudes of the geodetic frame. `StartLon` and `EndLon` must lie in the interval `[-180,180]`. If `EndLon < StartLon`, the object wraps `EndLon` to `StartLon + 360°`. Units are in degrees.

Data Types: double

### RadarReflectivity — Radar reflectivity object of surface

surfaceReflectivitySea object (default) | surfaceReflectivityCustom object

Radar reflectivity object of surface, specified as a `surfaceReflectivitySea` or `surfaceReflectivityCustom` for the normalized radar cross section (NRCS) of the surface. The default object is a `surfaceReflectivitySea` object with default properties.

**ReflectivityMap — Map of reflectivity type indices over the surface**

1 (default) | matrix of real values

This property contains a grid of reflectivity type values corresponding to vertices of the surface height data. If terrain or spectral model is present, `ReflectivityMap` must be a matrix of the same size as the domain of that data. Otherwise it must be scalar. Each element is an index into the third dimension of the `Reflectivity` property of the `surfaceReflectivityCustom` object.

**Dependencies**

To enable this property, set the `RadarReflectivity` property to a `surfaceReflectivityCustom` object.

Data Types: `double`

**ReferenceHeight — Surface reference height**

0 (default) | scalar

Reference height of surface height data, specified as a scalar. Surface heights are relative to the reference height. For surfaces with no height data, this is the assumed height of the entire surface. Units are in meters.

Data Types: `double`

**WindSpeed — Wind speed**

10 (default) | nonnegative scalar

Wind speed over sea surface, specified as a nonnegative scalar. Wind speed is defined at a height of 10 meters from the water surface. Wind speed is used as a parameter for the associated spectral model. Units are m/s.

Data Types: `double`

**WindDirection — Wind direction**

0 (default) | scalar

Wind direction over the sea surface, specified as a scalar in the range 0° to 180°. A standard right-handed Cartesian coordinate system is used. When the `IsEarthCentered` property of `radarScenario` is `true`, wind direction is a positive angle defined counterclockwise from the positive x-axis. Otherwise, wind direction is defined as clockwise from the North direction. This property is used to determine surface reflectivity and is used as a parameter for the associated spectral model. Units are in degrees.

Data Types: `double`

**Fetch — Wave fetch**

Inf (default) | positive scalar

Fetch, specified as a positive scalar. Fetch is the distance over a sea surface in which the wind blows in a single direction without obstruction. The fetch is used as a parameter for the associated spectral model. Units are in meters.

Data Types: `double`

## Object Functions

height     Height of point on surface  
 occlusion   Test for occlusion of point by a surface

## Examples

### Find Height of Sea Surface

Create a square sea surface area using the `seaSurface` object. Assume a moderate sea state with a wind speed of about 10 m/s, a fetch of 250 km and a length of 1.0 km. Add an Elfouhaily spectrum to the sea surface. Use the height function to determine the heights of 2 points on the map.

Create a radar scenario.

```
scene = radarScenario(IsEarthCentered = false);
rng('default')
```

Add a sea surface to the scene with an Elfouhaily spectrum.

```
spec = seaSpectrum(Resolution = 20);
srf = seaSurface(scene,Boundary = [-500 500; -500 500], ...
    WindSpeed = 10,Fetch = 250000,SpectralModel = spec);
```

Find the height at two points.

```
P1 = [0;0];
P2 = [30;-70];
H = height(srf,[P1 P2])
```

```
H = 1x2
    -0.9394    -0.2682
```

Display the sea surface properties in the surface manager.

```
mgr = scene.SurfaceManager

mgr =
  SurfaceManager with properties:
    UseOcclusion: 1
    Surfaces: [1x1 radar.scenario.SeaSurface]

mgr.Surfaces

ans =
  SeaSurface with properties:
    WindSpeed: 10
    WindDirection: 0
    Fetch: 250000
    SpectralModel: [1x1 seaSpectrum]
    RadarReflectivity: [1x1 surfaceReflectivitySea]
    ReflectivityMap: 1
```

```
ReferenceHeight: 0  
Boundary: [2x2 double]
```

### Test for Occlusion Between Two Points on Sea Surface

Create a square sea surface assuming a moderate sea state with a wind speed of about 12 knots (6.17 m/s), a fetch of 120 nmi (222.24 km), and a length of 1.024 km. Add an Elfouhaily spectrum to the sea surface. Use the `occlusion` object function to determine if the path from point 1 to point 2 is occluded.

Start by creating a radar scenario;

```
scene = radarScenario;
```

Add a sea surface with an Elfouhaily spectrum.

```
rng('default');  
spec = seaSpectrum('Resolution',16);
```

Create the sea surface.

```
bnds = [0 1024; 0 1024];  
srf = seaSurface(scene,'Boundary',bnds, ...  
    'WindSpeed',6.17,'Fetch',222.24e3, ...  
    'SpectralModel',spec);
```

Set two points for testing occlusion.

```
p1 = [1016; 368; -0.082];  
p2 = [10; 100; 0.13];
```

Determine if the path from p1 to p2 is occluded

```
tf1 = occlusion(srf,p1,p2)
```

```
tf1 = logical  
    1
```

### Create Surface from Sea Spectrum

Create a 1024-by-1024 m square sea surface. Assume an NRL reflectivity model for a high sea state 6 with a wind speed of about 20 m/s and a fetch of 250 km. Set `UseOcclusion` in the `SurfaceManager` to `false`.

Create a radar scenario.

```
scene = radarScenario;
```

Model the reflectivity using the NRL model.



```
refl = surfaceReflectivitySea(Model = 'NRL',SeaState = 6, ...
    Polarization = 'V')

refl =
    surfaceReflectivitySea with properties:

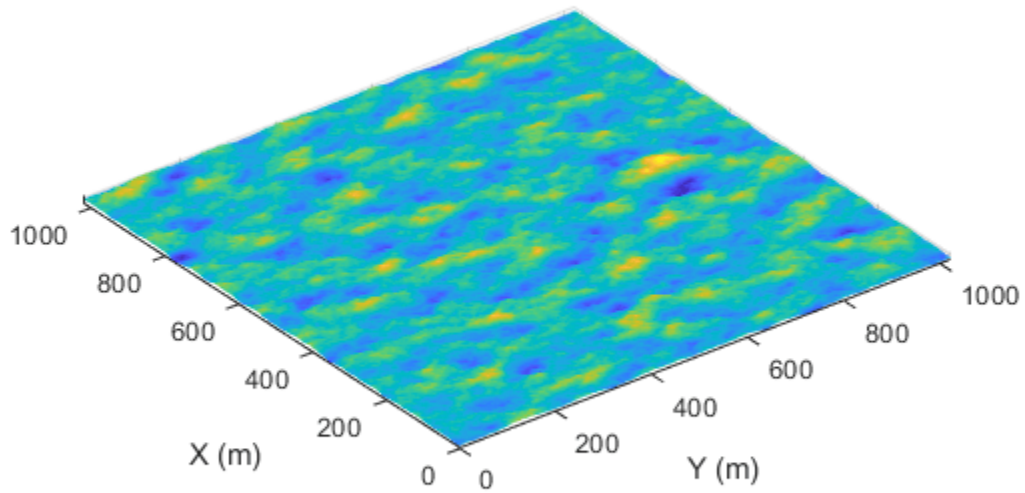
        Model: 'NRL'
        SeaState: 6
        Polarization: 'V'
        Speckle: 'None'

rng(2033)
spec = seaSpectrum(Resolution = 2);
bnds = [0 1024; 0 1024];
srf = seaSurface(scene,Boundary = bnds, ...
    WindSpeed = 20,Fetch = 250e3, ...
    SpectralModel = spec);
mgr = scene.SurfaceManager;
mgr.UseOcclusion = false

mgr =
    SurfaceManager with properties:

        UseOcclusion: 0
        Surfaces: [1x1 radar.scenario.SeaSurface]

x = linspace(srf.Boundary(1,1),srf.Boundary(1,2),1000);
y = linspace(srf.Boundary(2,1),srf.Boundary(2,2),1000);
[X,Y] = meshgrid(x,y);
X1 = X(:)';
Y1 = Y(:)';
hts = height(srf,[Y1;X1]);
hts = reshape(hts,length(x),length(y));
surf(x,y,hts)
axis equal
shading interp
ylabel('X (m)')
xlabel('Y (m)')
zlabel('Height (m)')
```



**See Also**

`rum` | `SurfaceManager` | `radarScenario` | `landSurface` | `seaSpectrum`

**Topics**

“Simulating Radar Returns from Moving Sea Surfaces”

**Introduced in R2022a**

# seaSurface

Add sea surface to radar scenario

## Syntax

```
srf = seaSurface(scene)
srf = seaSurface(scene,Name=Value)
```

## Description

`srf = seaSurface(scene)` adds a `SeaSurface` object `srf` to the `radarScenario` object `scene`.

`srf = seaSurface(scene,Name=Value)` adds a sea surface object with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as `(Name1=Value1,...,NameN=ValueN)`.

## Examples

### Find Height of Sea Surface

Create a square sea surface area using the `seaSurface` object. Assume a moderate sea state with a wind speed of about 10 m/s, a fetch of 250 km and a length of 1.0 km. Add an Elfouhaily spectrum to the sea surface. Use the height function to determine the heights of 2 points on the map.

Create a radar scenario.

```
scene = radarScenario(IsEarthCentered = false);
rng('default')
```

Add a sea surface to the scene with an Elfouhaily spectrum.

```
spec = seaSpectrum(Resolution = 20);
srf = seaSurface(scene,Boundary = [-500 500; -500 500], ...
    WindSpeed = 10,Fetch = 250000,SpectralModel = spec);
```

Find the height at two points.

```
P1 = [0;0];
P2 = [30;-70];
H = height(srf,[P1 P2])
```

```
H = 1×2
    -0.9394    -0.2682
```

Display the sea surface properties in the surface manager.

```
mgr = scene.SurfaceManager
mgr =
    SurfaceManager with properties:
```

```
UseOcclusion: 1
Surfaces: [1x1 radar.scenario.SeaSurface]
```

`mgr.Surfaces`

```
ans =
  SeaSurface with properties:

    WindSpeed: 10
    WindDirection: 0
    Fetch: 250000
    SpectralModel: [1x1 seaSpectrum]
    RadarReflectivity: [1x1 surfaceReflectivitySea]
    ReflectivityMap: 1
    ReferenceHeight: 0
    Boundary: [2x2 double]
```

### Test for Occlusion Between Two Points on Sea Surface

Create a square sea surface assuming a moderate sea state with a wind speed of about 12 knots (6.17 m/s), a fetch of 120 nmi (222.24 km), and a length of 1.024 km. Add an Elfouhaily spectrum to the sea surface. Use the occlusion object function to determine if the path from point 1 to point 2 is occluded.

Start by creating a radar scenario;

```
scene = radarScenario;
```

Add a sea surface with an Elfouhaily spectrum.

```
rng('default');
spec = seaSpectrum('Resolution',16);
```

Create the sea surface.

```
bnds = [0 1024; 0 1024];
srf = seaSurface(scene,'Boundary',bnds, ...
    'WindSpeed',6.17,'Fetch',222.24e3, ...
    'SpectralModel',spec);
```

Set two points for testing occlusion.

```
p1 = [1016; 368; -0.082];
p2 = [10; 100; 0.13];
```

Determine if the path from p1 to p2 is occluded

```
tf1 = occlusion(srf,p1,p2)
```

```
tf1 = logical
      1
```

## Create Surface from Sea Spectrum

Create a 1024-by-1024 m square sea surface. Assume an NRL reflectivity model for a high sea state 6 with a wind speed of about 20 m/s and a fetch of 250 km. Set `UseOcclusion` in the `SurfaceManager` to `false`.

Create a radar scenario.

```
scene = radarScenario;
```

Model the reflectivity using the NRL model.

```
refl = surfaceReflectivitySea(Model = 'NRL',SeaState = 6, ...
    Polarization = 'V')
```

```
refl =
    surfaceReflectivitySea with properties:
```

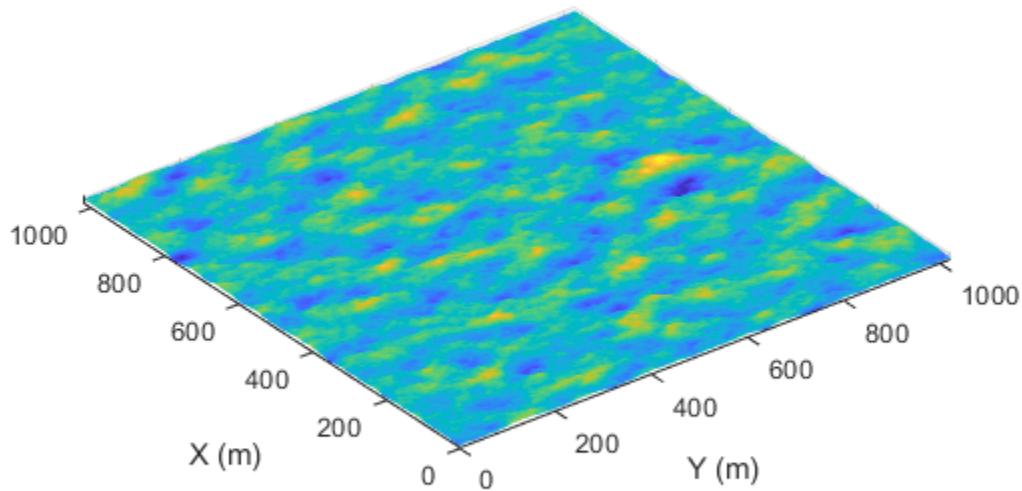
```
        Model: 'NRL'
        SeaState: 6
        Polarization: 'V'
        Speckle: 'None'
```

```
rng(2033)
spec = seaSpectrum(Resolution = 2);
bnds = [0 1024; 0 1024];
srf = seaSurface(scene,Boundary = bnds, ...
    WindSpeed = 20,Fetch = 250e3, ...
    SpectralModel = spec);
mgr = scene.SurfaceManager;
mgr.UseOcclusion = false
```

```
mgr =
    SurfaceManager with properties:

        UseOcclusion: 0
        Surfaces: [1x1 radar.scenario.SeaSurface]
```

```
x = linspace(srf.Boundary(1,1),srf.Boundary(1,2),1000);
y = linspace(srf.Boundary(2,1),srf.Boundary(2,2),1000);
[X,Y] = meshgrid(x,y);
X1 = X(:)';
Y1 = Y(:)';
hts = height(srf,[Y1;X1]);
hts = reshape(hts,length(x),length(y));
surf(x,y,hts)
axis equal
shading interp
ylabel('X (m)')
xlabel('Y (m)')
zlabel('Height (m)')
```



## Input Arguments

### scene — Radar scenario

radarScenario object

Radar scenario, specified as a radarScenario object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1, ..., NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

```
Example: srf = seaSurface(scene,'Boundary',[-500 500; -500
500], 'WindSpeed',10, 'Fetch',250000, 'SpectralModel',spec);
```

### SpectralModel — Sea surface omnidirectional motion spectrum

seaSpectrum object

Sea surface omnidirectional motion spectrum, specified as a seaSpectrum object. This object models surface heights over time.

**Dependencies**

To enable this property, set the `radarScenario` property `IsEarthCentered` to `false`.

**Boundary — Bounding rectangle of surface**

`[-Inf Inf; -Inf Inf]` (default) | 2-by-2 matrix of real values

Bounding rectangle of the surface, specified as a 2-by-2 matrix of real values. The bounding rectangle is defined by two two-dimensional points in either Cartesian or geodetic scenario coordinates. When the `IsEarthCentered` property of the `radarScenario` object is specified as:

- `false` — Scenario coordinates are Cartesian. Specify the bounding rectangle `[MinX, MaxX, MinY, MaxY]`, as Cartesian coordinates in the reference frame of the scenario, where `MinX < MaxX`, and `MinY < MaxY`.
- `true` — Scenario coordinates are geodetic. Specify the bounding rectangle as `[StartLat, EndLat, StartLon, EndLon]` where `StartLat` and `EndLat` are the minimum and maximum latitudes of the geodetic frames. `StartLat` and `EndLat` must lie in the interval `[-90,90]` where `StartLat < EndLat`.

Specify `StartLon` and `EndLon` as the minimum and maximum longitudes of the geodetic frame. `StartLon` and `EndLon` must lie in the interval `[-180,180]`. If `EndLon < StartLon`, the object wraps `EndLon` to `StartLon + 360°`. Units are in degrees.

Data Types: `double`

**RadarReflectivity — Radar reflectivity object of surface**

`surfaceReflectivitySea` object (default) | `surfaceReflectivityCustom` object

Radar reflectivity object of surface, specified as a `surfaceReflectivitySea` or `surfaceReflectivityCustom` for the normalized radar cross section (NRCS) of the surface. Defaults to a `surfaceReflectivitySea` object using a Barton land model and flatland land type.

**ReflectivityMap — Map of reflectivity type indices over the surface**

`1` (default) | matrix of real values

This property contains a grid of reflectivity type values corresponding to vertices of the surface height data. If terrain or spectral model is present, `ReflectivityMap` must be a matrix of the same size as the domain of that data. Otherwise it must be scalar. Each element is an index into the third dimension of the `Reflectivity` property of the `surfaceReflectivityCustom` object.

**Dependencies**

To enable this property, set the `RadarReflectivity` property to a `surfaceReflectivityCustom` object.

Data Types: `double`

**ReferenceHeight — Surface reference height**

`0` (default) | scalar

Reference height of surface height data, specified as a scalar. Surface heights are relative to the reference height. For surfaces with no height data, this is the assumed height of the entire surface. Units are in meters.

Data Types: `double`

**WindSpeed — Wind speed**

10 (default) | nonnegative scalar

Wind speed over sea surface, specified as a nonnegative scalar. Wind speed is defined at a height of 10 meters from the water surface. Wind speed is used as a parameter for the associated spectral model. Units are m/s.

Data Types: double

**WindDirection — Wind direction**

0 (default) | scalar

Wind direction over the sea surface, specified as a scalar in the range 0° to 180°. A standard right-handed Cartesian coordinate system is used. When the `IsEarthCentered` property of `radarScenario` is `true`, wind direction is a positive angle defined counterclockwise from the positive x-axis. Otherwise, wind direction is defined as clockwise from the North direction. This property is used to determine surface reflectivity and is used as a parameter for the associated spectral model. Units are in degrees.

Data Types: double

**Fetch — Wave fetch**

Inf (default) | positive scalar

Fetch, specified as a positive scalar. Fetch is the distance over a sea surface in which the wind blows in a single direction without obstruction. The fetch is used as a parameter for the associated spectral model. Units are in meters.

Data Types: double

**Output Arguments****srf — Sea surface**

SeaSurface object

Sea surface, returned as a SeaSurface object.

**See Also**

SurfaceManager | radarScenario | landSurface | height | occlusion | seaSpectrum | SeaSurface | surfaceReflectivitySea

**Topics**

“Simulating Radar Returns from Moving Sea Surfaces”

**Introduced in R2022a**



# occlusion

Test for occlusion of point by a surface

## Syntax

```
status = occlusion(surface,point1,point2)
```

## Description

`status = occlusion(surface,point1,point2)` returns `status` as `true` if the line-of-sight between `point1` and `point2` on a surface is occluded by parts of the surface.

## Examples

### Test for Occlusion Between Two Points on Land Surface

Create a radar scenario and specify set the `IsEarthCentered` property as `true` to obtain the terrain from a DTED file.

```
scene = radarScenario(IsEarthCentered = true);
```

Model the reflectivity as a constant gamma surface.

```
refl = surfaceReflectivityLand(Model = 'ConstantGamma',Gamma = -20);
```

Add a 0.1-by-0.1 degree land surface derived from a DTED file.

```
bdry = [39.5 39.6;-105.51 -105.41];
srf = landSurface(scene,Terrain = 'n39_w106_3arc_v2.dt1', ...
    Boundary = bdry,RadarReflectivity = refl);
```

Verify that occlusion is turned on.

```
mgr = scene.SurfaceManager
```

```
mgr =
  SurfaceManager with properties:
    UseOcclusion: 1
    Surfaces: [1x1 radar.scenario.LandSurface]
```

Plot the surface height.

```
x = linspace(srf.Boundary(2,1),srf.Boundary(2,2),201);
y = linspace(srf.Boundary(1,1),srf.Boundary(1,2),201);
[X,Y] = meshgrid(x,y);
X1 = X(:)';
Y1 = Y(:)';
H = height(srf,[Y1;X1]);
H = reshape(H,length(x),length(y));
surf(x,y,H)
```

```
shading interp
ylabel('Latitude (deg)')
xlabel('Longitude (deg)')
zlabel('Height (m)')
hold on
```

Test for occlusion.

```
ht1 = height(srf,[39.59 -105.5])
```

```
ht1 = 2810
```

```
ht2 = height(srf,[39.51 -105.41])
```

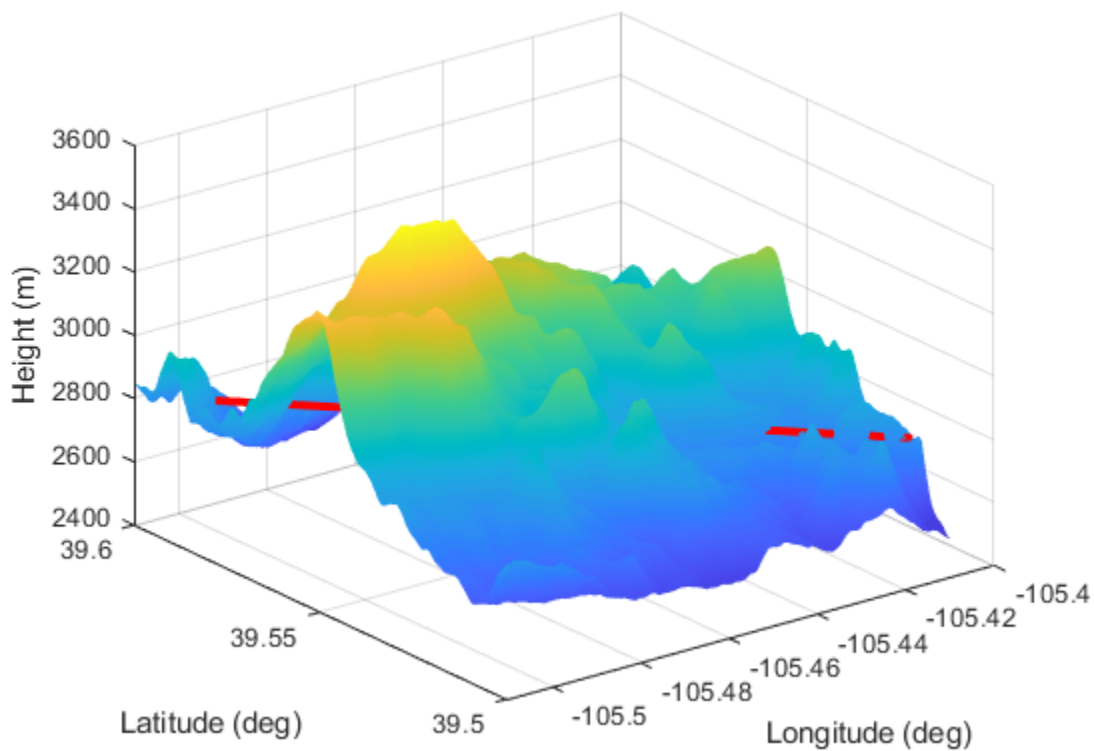
```
ht2 = 2786
```

```
occlusion(srf,[39.59 -105.5 ht1],[39.51 -105.41 ht2])
```

```
ans = logical
      1
```

The points are occluded. The line between the two points passes through the surface as shown.

```
plot3([-105.5 -105.41],[39.59 39.51], [ht1 ht2], 'r', 'LineWidth', 3)
```



## Test for Occlusion Between Two Points on Sea Surface

Create a square sea surface assuming a moderate sea state with a wind speed of about 12 knots (6.17 m/s), a fetch of 120 nmi (222.24 km), and a length of 1.024 km. Add an Elfouhaily spectrum to the sea surface. Use the `occlusion` object function to determine if the path from point 1 to point 2 is occluded.

Start by creating a radar scenario;

```
scene = radarScenario;
```

Add a sea surface with an Elfouhaily spectrum.

```
rng('default');
spec = seaSpectrum('Resolution',16);
```

Create the sea surface.

```
bnds = [0 1024; 0 1024];
srf = seaSurface(scene,'Boundary',bnds, ...
    'WindSpeed',6.17,'Fetch',222.24e3, ...
    'SpectralModel',spec);
```

Set two points for testing occlusion.

```
p1 = [1016; 368; -0.082];
p2 = [10; 100; 0.13];
```

Determine if the path from p1 to p2 is occluded

```
tf1 = occlusion(srf,p1,p2)
```

```
tf1 = logical
      1
```

## Input Arguments

### surface — Land or sea surface

LandSurface object | SeaSurface object

Surface, specified as a LandSurface object or SeaSurface object.

### point1 — Position of first point

length-3 real-valued vector

Position of the first point on a surface, specified as a length-3 real-valued vector.

If the `IsEarthCentered` property of the `radarScenario` object is specified as:

- `false` — Specify the three elements as the x-, y-, and z-coordinates in the reference frame of the tracking scenario. Units are in meters,
- `true` — Specify the three elements as the latitude in degrees, longitude in degrees, and altitude in meters, in the geodetic frame.

Data Types: double

**point2 — Position of second point**

length-3 real-valued vector

Position of the second point on a surface, specified as a length-3 real-valued vector.

If the `IsEarthCentered` property of the `radarScenario` object is specified as:

- `false` — Specify the three elements as the x-, y-, and z-coordinates in the reference frame of the tracking scenario. Units are in meters,
- `true` — Specify the three elements as the latitude in degrees, longitude in degrees, and altitude in meters, in the geodetic frame.

Data Types: `double`

**Output Arguments****status — Occlusion status**`true or 1 | false or 0`

Occlusion status, returned as a logical 1 (`true`) representing occluded, or 0 (`false`) representing not occluded.

**See Also**`LandSurface | SeaSurface | height`**Introduced in R2022a**

# height

Height of point on surface

## Syntax

```
hgt = height(surface,pt)
hgt = height(surface,pt,t)
```

## Description

`hgt = height(surface,pt)` returns the height `hgt` of the point `pt` on the surface. This syntax applies when the surface is a `SeaSurface` object or a `LandSurface` object.

`hgt = height(surface,pt,t)` returns the height `hgt` of the point `pt` on the surface at the time `t`. This syntax only applies when the surface is a `SeaSurface` object.

## Examples

### Find Height of Sea Surface

Create a square sea surface area using the `seaSurface` object. Assume a moderate sea state with a wind speed of about 10 m/s, a fetch of 250 km and a length of 1.0 km. Add an Elfouhaily spectrum to the sea surface. Use the height function to determine the heights of 2 points on the map.

Create a radar scenario.

```
scene = radarScenario(IsEarthCentered = false);
rng('default')
```

Add a sea surface to the scene with an Elfouhaily spectrum.

```
spec = seaSpectrum(Resolution = 20);
srf = seaSurface(scene,Boundary = [-500 500; -500 500], ...
    WindSpeed = 10,Fetch = 250000,SpectralModel = spec);
```

Find the height at two points.

```
P1 = [0;0];
P2 = [30;-70];
H = height(srf,[P1 P2])
```

```
H = 1x2
    -0.9394    -0.2682
```

Display the sea surface properties in the surface manager.

```
mgr = scene.SurfaceManager
mgr =
    SurfaceManager with properties:
```

```
UseOcclusion: 1
Surfaces: [1x1 radar.scenario.SeaSurface]
```

`mgr.Surfaces`

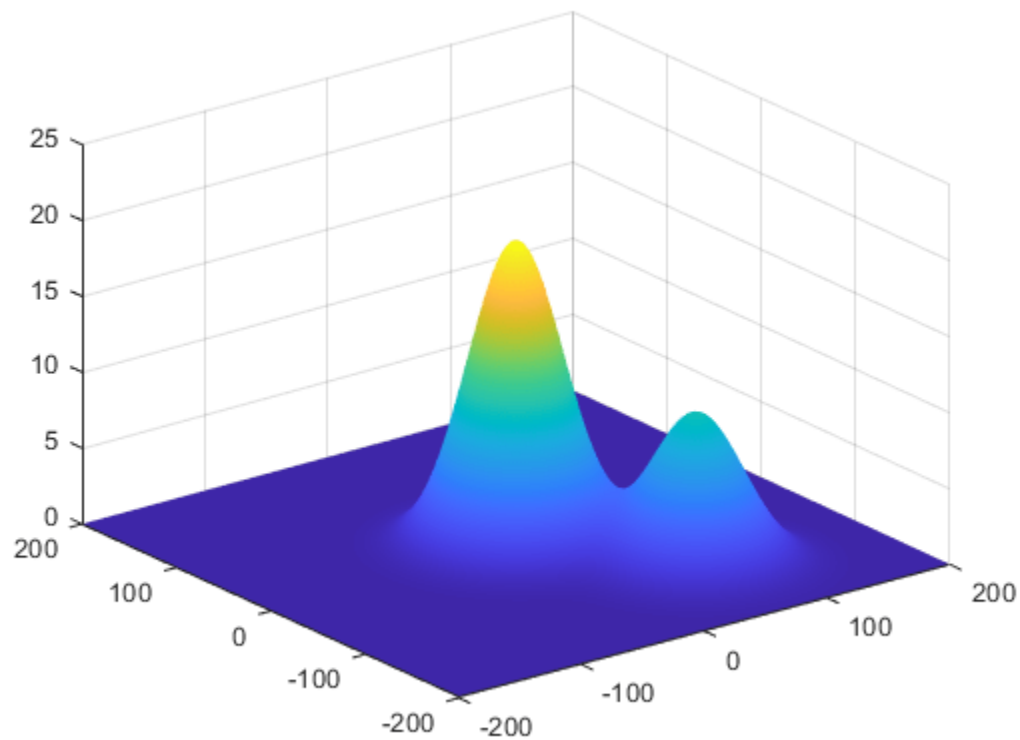
```
ans =
SeaSurface with properties:

    WindSpeed: 10
    WindDirection: 0
        Fetch: 250000
    SpectralModel: [1x1 seaSpectrum]
    RadarReflectivity: [1x1 surfaceReflectivitySea]
    ReflectivityMap: 1
    ReferenceHeight: 0
        Boundary: [2x2 double]
```

### Find Height of Land Surface

Create a radar scenario. Add a 400-by-400 m area to the scenario with two simulated hills. Find the height of two points.

```
scene = radarScenario('IsEarthCentered',false);
bnds = [-200 200; -200, 200];
x = -200:200;
y = -200:200;
[X,Y] = meshgrid(x,y);
htmap = 20*exp(-X.^2/2000 - Y.^2/2000) + 10*exp(-(X-70).^2/2000 - (Y+100).^2/2000);
surf(X,Y,htmap)
shading interp
```



Find the height of the surface at two points.

```
P1 = [0.0; 0.0]; % Point 1
P2 = [28.0; -40.0]; % Point 2
srf = landSurface(scene, 'Terrain', htm, 'Boundary', bnds)
```

```
srf =
  LandSurface with properties:

    RadarReflectivity: [1x1 surfaceReflectivityLand]
    ReflectivityMap: 1
    ReferenceHeight: 0
    Boundary: [2x2 double]
    Terrain: [401x401 double]
```

```
H = height(srf, [P1 P2])
```

```
H = 1x2

    20.0058    6.7565
```

Use the surface manager find the surfaces in the scenario.

```
mgr = scene.SurfaceManager

mgr =
  SurfaceManager with properties:
```

```
UseOcclusion: 1
Surfaces: [1x1 radar.scenario.LandSurface]
```

`mgr.Surfaces`

```
ans =
  LandSurface with properties:

    RadarReflectivity: [1x1 surfaceReflectivityLand]
    ReflectivityMap: 1
    ReferenceHeight: 0
    Boundary: [2x2 double]
    Terrain: [401x401 double]
```

## Input Arguments

### **surface** — Land or Sea surface

LandSurface object | SeaSurface object

Surface, specified as a LandSurface or SeaSurface object.

### **pt** — Point on surface

2-by-*N* matrix of real values | 3-by-*N* matrix of real values

Points on surface, specified as 2-by-*N* matrix of real values or a 3-by-*N* matrix of real values, where *N* is the number of points.

The coordinate system of the point depends on the value of the `IsEarthCentered` property of the radar scenario object::

- `false` — Each column of the 2-by-*N* matrix represents the x- and y-coordinates of points in meters. Each column of the 3-by-*N* matrix represents the x-, y-, and z-coordinates in meters. Note that the z-coordinate is irrelevant for surface height querying.
- `true` — Each column of the 2-by-*N* matrix is the latitude in degrees and longitude in degrees, in the geodetic frame. Each column of the 3-by-*N* matrix is the latitude in degrees, longitude in degrees, and altitude in degrees, in geodetic coordinates. Note that the latitude is irrelevant for surface height querying.

### **t** — Simulation time

scalar

Simulation time, specified as a scalar.

### Dependencies

To enable this argument, select `surface` as a SeaSurface object.

Data Types: `double`



## Output Arguments

### **hgt — Height of surface point**

scalar |  $N$ -element vector of real values

Height of point, returned as a scalar or an  $N$ -element vector of real values, where  $N$  is the number of queried positions. Units are in meters.

### **See Also**

LandSurface | SeaSurface | occlusion | SurfaceManager

**Introduced in R2022a**

# seaSpectrum

Sea surface omnidirectional motion spectrum model

## Description

The `seaSpectrum` object creates a spectrum model for use in the `SpectralModel` property of the `SeaSurface` object.

## Creation

### Syntax

```
seaspect = seaSpectrum  
seaspect = seaSpectrum(Name = Value)
```

### Description

`seaspect = seaSpectrum` creates a `seaSpectrum` object `seaspect` with default property values. The default sea surface spectrum and spreading function are based on the Elfouhaily model. The Elfouhaily model is an omnidirectional and wind-dependent spectrum. The wave spectrum consists of the frequency spectrum and angular spreading function. The spreading function is symmetric about the wind direction and has both wave number and wind speed dependence.

`seaspect = seaSpectrum(Name = Value)` creates a `seaSpectrum` object with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as (`Name1 = Value1, ..., NameN = ValueN`).

## Properties

### **SpectrumSource** — Source of omnidirectional spectrum

'Auto' (default) | 'Custom'

Source of the omnidirectional wave spectrum, specified as 'Auto' or 'Custom'.

- Choosing 'Auto' creates an Elfouhaily sea spectrum. The object offers properties that tailor the Elfouhaily sea spectrum.
- Choosing 'Custom' allows you to specify the omnidirectional wave spectrum using the `CustomSpectrum` and `WaveVectorSpacing` properties.

Data Types: `char` | `string`

### **Resolution** — Sea surface resolution

8 (default) | positive scalar | 1-by-2 vector of positive values

Sea surface resolution, specified as a positive scalar or as a 1-by-2 vector of positive values. The resolution vector takes the form `[resolutionX resolutionY]` where `resolutionX` and `resolutionY` denote the resolution in the *x*- and *y*- directions, respectively. If the resolution is a scalar, the *x*-resolution and *y*-resolution is the same.

The sea surface physical length is calculated as the difference of the limits of the `Boundary` property of the `SeaSurface` object in the `x` and `y` dimensions. The sea surface length has samples spaced by the `Resolution` property. The length should not be set below 0.02 m as the wave motion below 0.02 m is minimal. Units are in meters.

Data Types: `double`

### CustomSpectrum — Omnidirectional wave spectrum

*M*-by-*N* matrix

Omnidirectional wave spectrum, specified as an *M*-by-*N* matrix. *M* and *N* dictate the inverse fast Fourier transfer (IFFT) length when returning the elevation data of the sea surface in the `x` and `y` dimensions, respectively.

The resolution for the custom case is calculated as  $Resolution = surface\ length./size(CustomSpectrum)$ , where *surface length* is the sea surface physical length, calculated as the difference of the limits of the `Boundary` property of the `seaSurface` in the `x` and `y` dimensions.

#### Dependencies

To enable this property, set the value of the `SpectrumSource` property to `'Custom'`.

Data Types: `double`

### WaveVectorSpacing — Wave vector domain spacing

$2\pi/1024$  (default) | positive scalar | 1-by-2 real-valued vector

Positive wavevector domain spacing, specified as a scalar or a 1-by-2 dimension vector, [`kx ky`]. This property represents the wavevector spacing in the `x` and `y` dimensions, respectively. Units are radians/meter. This value is typically set as  $WaveVectorSpacing \leq 2\pi/surface\ length$  where *surface length* is the sea surface physical length, calculated as the difference of the limits of the `Boundary` property on theseaSurface in the `x` and `y` dimensions.

#### Dependencies

To enable this property, set the `SpectrumSource` property to `'Custom'`.

Data Types: `double`

## Examples

### Create Surface from Sea Spectrum

Create a 1024-by-1024 m square sea surface. Assume an NRL reflectivity model for a high sea state 6 with a wind speed of about 20 m/s and a fetch of 250 km. Set `UseOcclusion` in the `SurfaceManager` to `false`.

Create a radar scenario.

```
scene = radarScenario;
```

Model the reflectivity using the NRL model.

```
refl = surfaceReflectivitySea(Model = 'NRL',SeaState = 6, ...
    Polarization = 'V')
```

```
refl =
  surfaceReflectivitySea with properties:

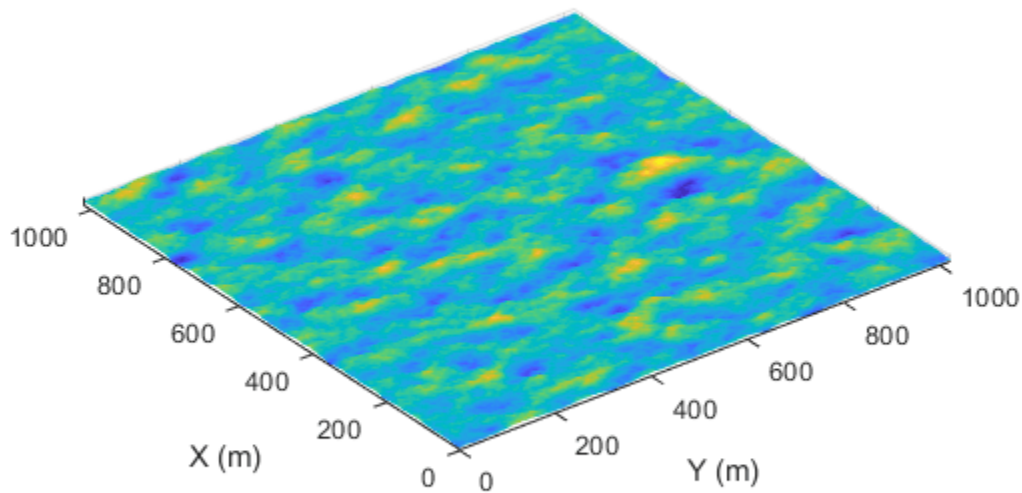
    Model: 'NRL'
    SeaState: 6
    Polarization: 'V'
    Speckle: 'None'

rng(2033)
spec = seaSpectrum(Resolution = 2);
bnds = [0 1024; 0 1024];
srf = seaSurface(scene,Boundary = bnds, ...
    WindSpeed = 20,Fetch = 250e3, ...
    SpectralModel = spec);
mgr = scene.SurfaceManager;
mgr.UseOcclusion = false

mgr =
  SurfaceManager with properties:

    UseOcclusion: 0
    Surfaces: [1x1 radar.scenario.SeaSurface]

x = linspace(srf.Boundary(1,1),srf.Boundary(1,2),1000);
y = linspace(srf.Boundary(2,1),srf.Boundary(2,2),1000);
[X,Y] = meshgrid(x,y);
X1 = X(:)';
Y1 = Y(:)';
hts = height(srf,[Y1;X1]);
hts = reshape(hts,length(x),length(y));
surf(x,y,hts)
axis equal
shading interp
ylabel('X (m)')
xlabel('Y (m)')
zlabel('Height (m)')
```



### Sea Surface with Custom Wave Spectrum

Create a square sea surface with a custom wave spectrum. Import a JONSWAP wave spectrum with a moderate sea state. The spectrum is a 64-by-64 matrix with a physical length of 512 m and a resolution of 2 m.

```
scene = radarScenario(IsEarthCentered = false);
```

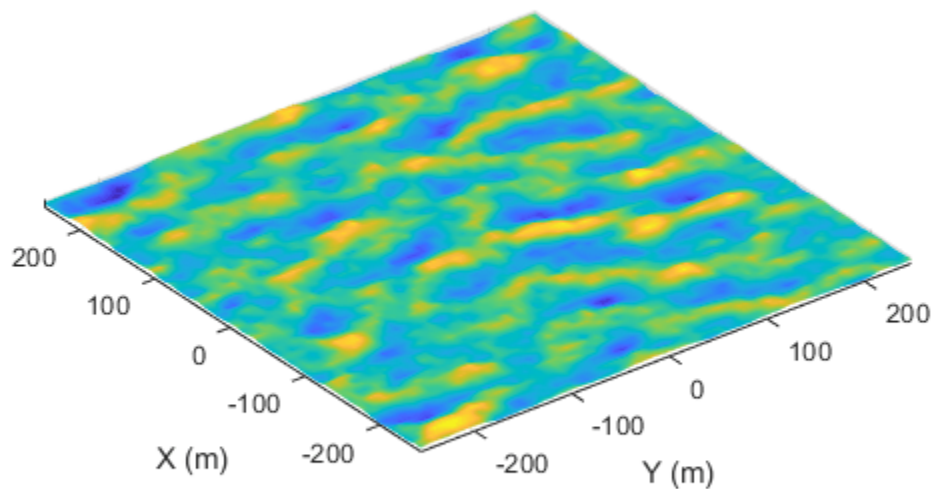
Add a sea surface to the scene with a custom spectrum

```
load('jonswap.mat');
spec = seaSpectrum(SpectrumSource = 'Custom', Resolution = 2, ...
    CustomSpectrum = Psi, WaveVectorSpacing = 2*pi/512)
```

```
spec =
    seaSpectrum with properties:
        SpectrumSource: 'Custom'
        CustomSpectrum: [64x64 double]
        WaveVectorSpacing: 0.0123
```

```
srf = seaSurface(scene, Boundary = [-256 256; -256 256], ...
    SpectralModel = spec)
```

```
srf =  
  SeaSurface with properties:  
  
      WindSpeed: 10  
      WindDirection: 0  
      Fetch: Inf  
      SpectralModel: [1x1 seaSpectrum]  
      RadarReflectivity: [1x1 surfaceReflectivitySea]  
      ReflectivityMap: 1  
      ReferenceHeight: 0  
      Boundary: [2x2 double]  
  
x = linspace(srf.Boundary(2,1),srf.Boundary(2,2),1000);  
y = linspace(srf.Boundary(1,1),srf.Boundary(1,2),1000);  
[X,Y] = meshgrid(x,y);  
X1 = X(:)';  
Y1 = Y(:)';  
hts = height(srf,[Y1;X1]);  
hts = reshape(hts,length(x),length(y));  
surf(x,y,hts)  
axis equal  
shading interp  
ylabel('X (m)')  
xlabel('Y (m)')  
zlabel('Height (m)')
```



## References

- [1] Elfouhaily, T., B. Chapron, K. Katsaros, and D. Vandemark. "A Unified Directional Spectrum for Long and Short Wind-Driven Waves." *Journal of Geophysical Research: Oceans* 102, no. C7 (July 15, 1997): 15781-96. <https://doi.org/10.1029/97JC00467>
- [2] Tessendorf, Jerry . "Simulating Ocean Water." *Presented at SigGraph*, 1999 - 2004.

## See Also

radarScenario | seareflectivity | seaSurface | SurfaceManager

**Introduced in R2022a**

# SurfaceManager

Manage surfaces in radar scenario

## Description

The `SurfaceManager` object lists the surfaces in a radar scenario, `radarScenario`. You can enable or disable occlusion by surfaces in the radar scenario. The `occlusion` object function determines if any surfaces occlude the line of sight between two points. Use the `height` object function to query the height of surfaces at any location in the scenario.

## Creation

Use the `landSurface` object function to create `LandSurface` objects. Use `seaSurface` object function to create `SeaSurface` objects. Obtain the `SurfaceManager` object from the `SurfaceManager` property of the `radarScenario` object.

## Properties

### UseOcclusion — Enable line-of-sight occlusion by surfaces

`true` (default) | `false`

Enable line-of-sight occlusion by surfaces, specified as `true` or `false`.

When specified as:

- `true` — The scenario models the occlusion of the line-of-sight by surfaces between points. In this case, the `detect` object function of the `radarScenario` object or the `detect` object function of the `Platform` object accounts for surface occlusion.

---

**Note** If the `IsEarthCentered` property of the `radarScenario` object is specified as `true`, selecting this option also enables horizon occlusion based on the WGS84 Earth model.

---

- `false` — The scenario does not models the occlusion of the line-of-sight due to ground surfaces or the WGS84 Earth model.

### Surfaces — Surfaces in radar scenario

array of `LandSurface` or `SeaSurface` objects

Surfaces in radar scenario, specified as an array of `LandSurface` and `SeaSurface` objects.

You can add land surfaces to a radar scenario using the `landSurface` and sea surfaces using the `seaSurface` object function.

## Object Functions

`height`      Height of point on surface  
`occlusion`    Test for occlusion of point by a surface



## Examples

### Create Land Surface from DTED File

Create a radar scenario and specify its `IsEarthCentered` property as `true` to use DTED file.

```
scene = radarScenario(IsEarthCentered = true);
```

Model the reflectivity as a constant gamma surface.

```
refl = surfaceReflectivityLand(Model = 'ConstantGamma',Gamma = -20);
```

Add a 0.1-by-0.1 degree land surface derived from a DTED file.

```
bdry = [39.5 39.6;-105.51 -105.41];
srf = landSurface(scene,Terrain = 'n39_w106_3arc_v2.dt1', ...
    Boundary = bdry,RadarReflectivity = refl)
```

```
srf =
  LandSurface with properties:

    RadarReflectivity: [1x1 surfaceReflectivityLand]
    ReflectivityMap: 1
    ReferenceHeight: 0
    Boundary: [2x2 double]
    Terrain: 'n39_w106_3arc_v2.dt1'
```

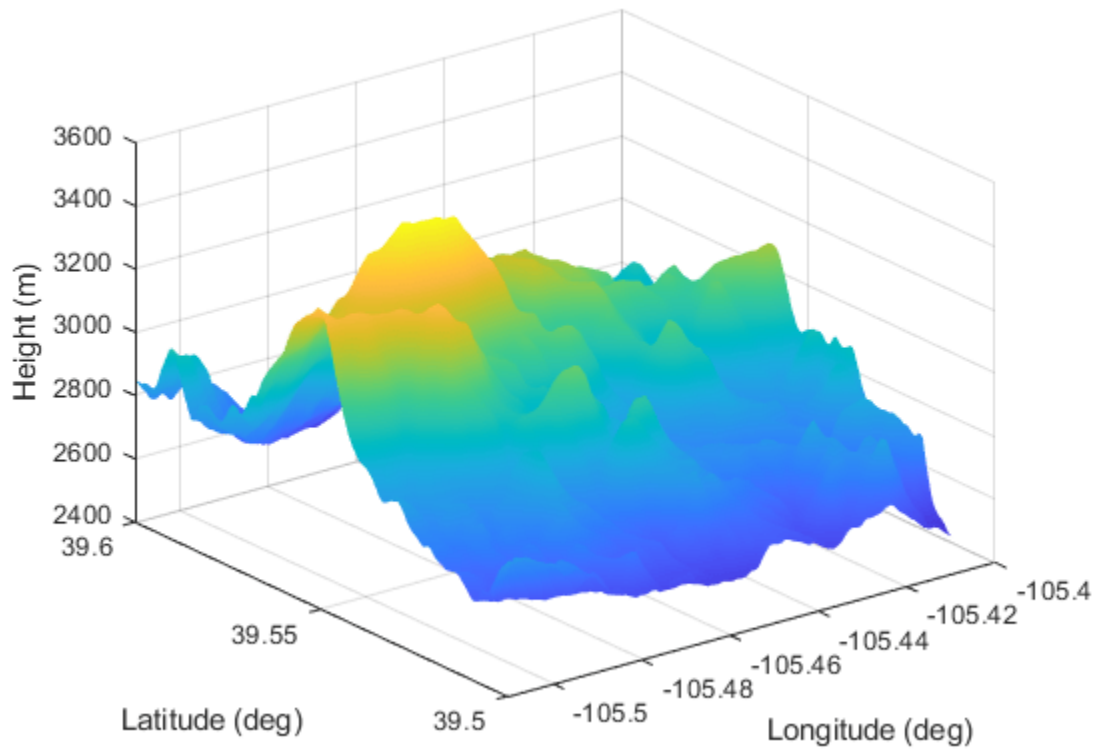
```
mgr = scene.SurfaceManager
```

```
mgr =
  SurfaceManager with properties:

    UseOcclusion: 1
    Surfaces: [1x1 radar.scenario.LandSurface]
```

Plot the surface height.

```
x = linspace(srf.Boundary(2,1),srf.Boundary(2,2),201);
y = linspace(srf.Boundary(1,1),srf.Boundary(1,2),201);
[X,Y] = meshgrid(x,y);
X1 = X(:)';
Y1 = Y(:)';
H = height(srf,[Y1;X1]);
H = reshape(H,length(x),length(y));
surf(x,y,H)
shading interp
ylabel('Latitude (deg)')
xlabel('Longitude (deg)')
zlabel('Height (m)')
```



**See Also**

[landSurface](#) | [LandSurface](#) | [seaSurface](#) | [SeaSurface](#) | [radarScenario](#)

**Introduced in R2022a**

# ClutterGenerator

Clutter generator object

## Description

The `ClutterGenerator` object specifies clutter properties and regions that control and manage radar clutter for sensors in a `radarScenario`. Sensors types include `radarDataGenerator` and `radarTransceiver` objects.

## Creation

Create a clutter generator using the `clutterGenerator` object function. Then use the `getClutterGenerator` object function to obtain information about clutter generators for radar scenarios and radars in the scenarios. Use the `ringClutterRegion` object function to create clutter regions.

## Properties

### **Resolution — Nominal spacing of clutter patches**

40 (default) | positive scalar

Nominal resolution of clutter patches, specified as a positive scalar. The nominal value is the expected surface resolution of the radar system. Units are in meters.

Data Types: `double`

### **RangeLimit — Range limit of clutter generation**

10000 (default) | positive scalar

Range limit of clutter generation, specified as a positive scalar. Clutter generation is limited to this range when the clutter region is unbounded. Units are in meters.

Data Types: `double`

### **UseBeam — Use beam footprint as clutter region**

true (default) | false

Enable the use of the beam footprint as the clutter region, specified as `true` or `false`. When `true`, the mainlobe clutter return is automatically included. Use the `ringClutterRegion` object function of the `ClutterGenerator` to create a custom region.

Data Types: `logical`

### **UseShadowing — Enable use of surface self-occlusion**

true (default) | true

Enable use of surface self-occlusion when generating clutter, specified as `true` or `false`. Surface self-occlusion is referred to as shadowing.

Data Types: `logical`

**Regions — Scenario clutter region**

RingClutterRegion

This property is read-only.

Region of the scenario in which is generated, returned as a RingClutterRegion. There can be multiple clutter regions.

**Radar — Radar**

radarDataGenerator object | radarTransceiver object

This property is read-only.

Radar objects for which clutter is generated, returned as a radarDataGenerator or a radarTransceiver object. There can be multiple radar objects.

**Object Functions**

getClutterGenerator Obtain clutter generator belonging to a radar  
ringClutterRegion Ring clutter region

**Examples****Create Clutter Object with Two Clutter Regions**

Generate clutter from a surface having two clutter regions. Start by creating a radarDataGenerator. Use a radar frequency of 1 GHz, a 100 meter range resolution, a 5 kHz pulse repetition frequency (PRF), and 128 pulses. The beam is symmetric with a 4 degree two-sided beamwidth in azimuth and elevation.

```
fc = 1e9;
rangeRes = 100;
prf = 5e3;
numPulses = 128;
beamwidth = 4;
```

Use the PRF and number of pulses to calculate the nominal Doppler and range-rate resolution. The radar will update once each coherent processing interval (CPI).

```
dopRes = prf/numPulses;
lambda = freq2wavelen(fc);
rangeRateRes = dop2speed(dopRes,lambda)/2;
cpiTime = numPulses/prf;

rdr = radarDataGenerator(1,'No scanning','UpdateRate',1/cpiTime, ...
    'DetectionMode','Monostatic','TargetReportFormat','Detections', ...
    'DetectionCoordinates','Scenario', ...
    'HasINS',true,'HasElevation',true,'HasFalseAlarms',false, ...
    'HasNoise',false,'HasRangeRate',true, ...
    'HasRangeAmbiguities',true,'HasRangeRateAmbiguities',true, ...
    'CenterFrequency',fc,'FieldOfView',beamwidth, ...
    'AzimuthResolution',beamwidth,'ElevationResolution', ...
    beamwidth,'RangeResolution', ...
    rangeRes,'RangeRateResolution',rangeRateRes, ...
```

```
'ReferenceRange',20e3,'ReferenceRCS',0, ...
'DetectionProbability',0.9);
```

Create a scenario using the `radarScenario` object, setting the update rate to zero so that the update interval is derived from sensors in the scene.

```
scenario = radarScenario('UpdateRate',0,'IsEarthCentered',false);
```

Now create the scenario surface. Choose a constant-gamma reflectivity model with a gamma value appropriate for flatland. This gamma value can be found using the `surfacegamma` function. Using this value, create a `surfaceReflectivityLand` object to add to a `LandSurface` using the `RadarReflectivity` property.

```
gammaDB = surfacegamma('Flatland');
refl = surfaceReflectivityLand('Model','ConstantGamma', ...
    'Gamma',gammaDB);
landSurface(scenario,'RadarReflectivity',refl);
```

Add two clutter regions to the scenario. Use the `clutterGenerator` object function to construct a clutter generator and enable clutter generation for the radar. The `Resolution` property defines the nominal spacing of clutter patches. Set this to be 1/5th of the range resolution to get multiple clutter patches per range gate. Set the range limit to 20 km. `UseBeam` indicates if clutter generation should be performed automatically for the mainlobe of the antenna pattern.

```
cluttergen = clutterGenerator(scenario,rdr,'Resolution', ...
    rangeRes/5,'RangeLimit',20e3,'UseBeam',true);
rgn1 = ringClutterRegion(cluttergen,1000,10000,30,45);
rgn2 = ringClutterRegion(cluttergen,1000,10000,30,105);
```

The `getClutterGenerator` function displays the two ring-shaped clutter regions belonging to the scenario.

```
getClutterGenerator(scenario,rdr)
```

```
ans =
    ClutterGenerator with properties:

        Resolution: 20
        RangeLimit: 20000
        UseBeam: 1
        UseShadowing: 1
        Regions: [1x2 radar.scenario.RingClutterRegion]
        Radar: [1x1 radarDataGenerator]
```

## See Also

[landSurface](#) | [seaSurface](#) | [radarDataGenerator](#) | [radarTransceiver](#) | [radarScenario](#) | [clutterGenerator](#) | [getClutterGenerator](#) | [ringClutterRegion](#)

## Topics

“Simulate Radar Detections of Surface Targets in Clutter”  
 “Generate Clutter and Target Returns for MTI Radar”  
 “Simulating Radar Returns from Moving Sea Surfaces”

**Introduced in R2022a**

# clutterGenerator

Add clutter generator for radar

## Syntax

```
genclutter = clutterGenerator(scenario,radar,Name=Value)
```

## Description

`genclutter = clutterGenerator(scenario,radar,Name=Value)` adds a ClutterGenerator object `genclutter` for the radar to the `radarScenario` scenario.

## Examples

### Create Clutter Object with Two Clutter Regions

Generate clutter from a surface having two clutter regions. Start by creating a `radarDataGenerator`. Use a radar frequency of 1 GHz, a 100 meter range resolution, a 5 kHz pulse repetition frequency (PRF), and 128 pulses. The beam is symmetric with a 4 degree two-sided beamwidth in azimuth and elevation.

```
fc = 1e9;
rangeRes = 100;
prf = 5e3;
numPulses = 128;
beamwidth = 4;
```

Use the PRF and number of pulses to calculate the nominal Doppler and range-rate resolution. The radar will update once each coherent processing interval (CPI).

```
dopRes = prf/numPulses;
lambda = freq2wavelen(fc);
rangeRateRes = dop2speed(dopRes,lambda)/2;
cpiTime = numPulses/prf;
```

```
rdr = radarDataGenerator(1,'No scanning','UpdateRate',1/cpiTime, ...
    'DetectionMode','Monostatic','TargetReportFormat','Detections', ...
    'DetectionCoordinates','Scenario', ...
    'HasINS',true,'HasElevation',true,'HasFalseAlarms',false, ...
    'HasNoise',false,'HasRangeRate',true, ...
    'HasRangeAmbiguities',true,'HasRangeRateAmbiguities',true, ...
    'CenterFrequency',fc,'FieldOfView',beamwidth, ...
    'AzimuthResolution',beamwidth,'ElevationResolution', ...
    beamwidth,'RangeResolution', ...
    rangeRes,'RangeRateResolution',rangeRateRes, ...
    'ReferenceRange',20e3,'ReferenceRCS',0, ...
    'DetectionProbability',0.9);
```

Create a scenario using the `radarScenario` object, setting the update rate to zero so that the update interval is derived from sensors in the scene.

```
scenario = radarScenario('UpdateRate',0,'IsEarthCentered',false);
```

Now create the scenario surface. Choose a constant-gamma reflectivity model with a gamma value appropriate for flatland. This gamma value can be found using the `surfacegamma` function. Using this value, create a `surfaceReflectivityLand` object to add to a `LandSurface` using the `RadarReflectivity` property.

```
gammaDB = surfacegamma('Flatland');
refl = surfaceReflectivityLand('Model','ConstantGamma', ...
    'Gamma',gammaDB);
landSurface(scenario,'RadarReflectivity',refl);
```

Add two clutter regions to the scenario. Use the `clutterGenerator` object function to construct a clutter generator and enable clutter generation for the radar. The `Resolution` property defines the nominal spacing of clutter patches. Set this to be 1/5th of the range resolution to get multiple clutter patches per range gate. Set the range limit to 20 km. `UseBeam` indicates if clutter generation should be performed automatically for the mainlobe of the antenna pattern.

```
cluttergen = clutterGenerator(scenario,rdr,'Resolution', ...
    rangeRes/5,'RangeLimit',20e3,'UseBeam',true);
rgn1 = ringClutterRegion(cluttergen,1000,10000,30,45);
rgn2 = ringClutterRegion(cluttergen,1000,10000,30,105);
```

The `getClutterGenerator` function displays the two ring-shaped clutter regions belonging to the scenario.

```
getClutterGenerator(scenario,rdr)
```

```
ans =
  ClutterGenerator with properties:

    Resolution: 20
   RangeLimit: 20000
     UseBeam: 1
  UseShadowing: 1
    Regions: [1x2 radar.scenario.RingClutterRegion]
      Radar: [1x1 radarDataGenerator]
```

## Input Arguments

### **scenario** — Radar scenario

`radarScenario` object

Radar scenario, specified as a `radarScenario` object.

### **radar** — Radar

`radarDataGenerator` object | `radarTransceiver` object

Radar, specified as a `radarDataGenerator` or `radarTransceiver` object.

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.



*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `Resolution = 34`

### **Resolution — Nominal spacing of clutter patches**

40 (default) | positive scalar

Nominal resolution of clutter patches, specified as a positive scalar. The nominal value is the expected resolution ground of the radar system. Units are in meters.

Data Types: `double`

### **RangeLimit — Range limit of clutter generation**

10000 (default) | positive scalar

Range limit of clutter generation, specified as a positive scalar. Clutter generation is limited to this range when the clutter region is unbounded. Units are in meters.

Data Types: `double`

### **UseBeam — Use beam footprint as clutter region**

`true` (default) | `false`

Enable use of the beam footprint as the clutter region, specified as `true` or `false`. When `true`, the mainlobe clutter return is automatically included. Use the `ringClutterRegion` object function of the `ClutterGenerator` to create a custom region.

Data Types: `logical`

### **UseShadowing — Enable use of surface self-occlusion**

`true` (default) | `true`

Enable use of surface self-occlusion when generating clutter, specified as `true` or `false`. Surface self-occlusion is referred to as shadowing.

Data Types: `logical`

## **Output Arguments**

### **genClutter — Clutter generator**

`ClutterGenerator` object

Clutter generator, returned as a `ClutterGenerator` object.

## **See Also**

`landSurface` | `seaSurface` | `radarDataGenerator` | `radarTransceiver` | `radarScenario` | `clutterGenerator` | `getClutterGenerator` | `ringClutterRegion`

## **Topics**

“Simulate Radar Detections of Surface Targets in Clutter”

“Generate Clutter and Target Returns for MTI Radar”

“Simulating Radar Returns from Moving Sea Surfaces”

## **Introduced in R2022a**

## getClutterGenerator

Obtain clutter generator belonging to a radar

### Syntax

```
cluttergen = getClutterGenerator(scenario, radar)
```

### Description

`cluttergen = getClutterGenerator(scenario, radar)` returns a clutter generator object `cluttergen` for the radar belonging to the radar scenario `scenario`.

### Examples

#### Create Clutter Object with Two Clutter Regions

Generate clutter from a surface having two clutter regions. Start by creating a `radarDataGenerator`. Use a radar frequency of 1 GHz, a 100 meter range resolution, a 5 kHz pulse repetition frequency (PRF), and 128 pulses. The beam is symmetric with a 4 degree two-sided beamwidth in azimuth and elevation.

```
fc = 1e9;
rangeRes = 100;
prf = 5e3;
numPulses = 128;
beamwidth = 4;
```

Use the PRF and number of pulses to calculate the nominal Doppler and range-rate resolution. The radar will update once each coherent processing interval (CPI).

```
dopRes = prf/numPulses;
lambda = freq2wavelen(fc);
rangeRateRes = dop2speed(dopRes, lambda)/2;
cpiTime = numPulses/prf;
```

```
rdr = radarDataGenerator(1, 'No scanning', 'UpdateRate', 1/cpiTime, ...
    'DetectionMode', 'Monostatic', 'TargetReportFormat', 'Detections', ...
    'DetectionCoordinates', 'Scenario', ...
    'HasINS', true, 'HasElevation', true, 'HasFalseAlarms', false, ...
    'HasNoise', false, 'HasRangeRate', true, ...
    'HasRangeAmbiguities', true, 'HasRangeRateAmbiguities', true, ...
    'CenterFrequency', fc, 'FieldOfView', beamwidth, ...
    'AzimuthResolution', beamwidth, 'ElevationResolution', ...
    beamwidth, 'RangeResolution', ...
    rangeRes, 'RangeRateResolution', rangeRateRes, ...
    'ReferenceRange', 20e3, 'ReferenceRCS', 0, ...
    'DetectionProbability', 0.9);
```

Create a scenario using the `radarScenario` object, setting the update rate to zero so that the update interval is derived from sensors in the scene.

```
scenario = radarScenario('UpdateRate',0,'IsEarthCentered',false);
```

Now create the scenario surface. Choose a constant-gamma reflectivity model with a gamma value appropriate for flatland. This gamma value can be found using the `surfacegamma` function. Using this value, create a `surfaceReflectivityLand` object to add to a `LandSurface` using the `RadarReflectivity` property.

```
gammaDB = surfacegamma('Flatland');
refl = surfaceReflectivityLand('Model','ConstantGamma', ...
    'Gamma',gammaDB);
landSurface(scenario,'RadarReflectivity',refl);
```

Add two clutter regions to the scenario. Use the `clutterGenerator` object function to construct a clutter generator and enable clutter generation for the radar. The `Resolution` property defines the nominal spacing of clutter patches. Set this to be 1/5th of the range resolution to get multiple clutter patches per range gate. Set the range limit to 20 km. `UseBeam` indicates if clutter generation should be performed automatically for the mainlobe of the antenna pattern.

```
cluttergen = clutterGenerator(scenario,rdr,'Resolution', ...
    rangeRes/5,'RangeLimit',20e3,'UseBeam',true);
rgn1 = ringClutterRegion(cluttergen,1000,10000,30,45);
rgn2 = ringClutterRegion(cluttergen,1000,10000,30,105);
```

The `getClutterGenerator` function displays the two ring-shaped clutter regions belonging to the scenario.

```
getClutterGenerator(scenario,rdr)
```

```
ans =
  ClutterGenerator with properties:

    Resolution: 20
    RangeLimit: 20000
    UseBeam: 1
    UseShadowing: 1
    Regions: [1x2 radar.scenario.RingClutterRegion]
    Radar: [1x1 radarDataGenerator]
```

## Input Arguments

### **scenario** — radar scenario

`radarScenario` object

Radar scenario, specified as a `radarScenario` object.

### **radar** — Radar

`radarDataGenerator` object | `radarTransceiver` object

Radar, specified as a `radarDataGenerator` or `radarTransceiver` object.

## Output Arguments

### **cluttergen** — Clutter generator object

`ClutterGenerator` object

Clutter generator, returned as `ClutterGenerator` object.

**See Also**

`ClutterGenerator` | `clutterGenerator` | `ringClutterRegion`

**Topics**

“Simulate Radar Detections of Surface Targets in Clutter”

“Generate Clutter and Target Returns for MTI Radar”

“Simulating Radar Returns from Moving Sea Surfaces”

**Introduced in R2022a**

# ringClutterRegion

Ring clutter region

## Syntax

```
region = ringClutterRegion(cluttergen,minrad,maxrad,azimuth_span,
azimuth_center)
```

## Description

`region = ringClutterRegion(cluttergen,minrad,maxrad,azimuth_span,azimuth_center)` creates a ring-shaped clutter region in the clutter generator object `cluttergen`. The region ranges from a minimum radius `minrad` to a maximum radius `maxrad` and a span of azimuth angles `azimuth_span` centered on `azimuth_center`. Clutter regions are listed in the Regions of ClutterGenerator object.

## Examples

### Create Clutter Object with Two Clutter Regions

Generate clutter from a surface having two clutter regions. Start by creating a `radarDataGenerator`. Use a radar frequency of 1 GHz, a 100 meter range resolution, a 5 kHz pulse repetition frequency (PRF), and 128 pulses. The beam is symmetric with a 4 degree two-sided beamwidth in azimuth and elevation.

```
fc = 1e9;
rangeRes = 100;
prf = 5e3;
numPulses = 128;
beamwidth = 4;
```

Use the PRF and number of pulses to calculate the nominal Doppler and range-rate resolution. The radar will update once each coherent processing interval (CPI).

```
dopRes = prf/numPulses;
lambda = freq2wavelen(fc);
rangeRateRes = dop2speed(dopRes,lambda)/2;
cpiTime = numPulses/prf;
```

```
rdr = radarDataGenerator(1,'No scanning','UpdateRate',1/cpiTime, ...
'DetectionMode','Monostatic','TargetReportFormat','Detections', ...
'DetectionCoordinates','Scenario', ...
'HasINS',true,'HasElevation',true,'HasFalseAlarms',false, ...
'HasNoise',false,'HasRangeRate',true, ...
'HasRangeAmbiguities',true,'HasRangeRateAmbiguities',true, ...
'CenterFrequency',fc,'FieldOfView',beamwidth, ...
'AzimuthResolution',beamwidth,'ElevationResolution', ...
beamwidth,'RangeResolution', ...
rangeRes,'RangeRateResolution',rangeRateRes, ...
'ReferenceRange',20e3,'ReferenceRCS',0, ...
'DetectionProbability',0.9);
```

Create a scenario using the `radarScenario` object, setting the update rate to zero so that the update interval is derived from sensors in the scene.

```
scenario = radarScenario('UpdateRate',0,'IsEarthCentered',false);
```

Now create the scenario surface. Choose a constant-gamma reflectivity model with a gamma value appropriate for flatland. This gamma value can be found using the `surfacegamma` function. Using this value, create a `surfaceReflectivityLand` object to add to a `LandSurface` using the `RadarReflectivity` property.

```
gammaDB = surfacegamma('Flatland');
refl = surfaceReflectivityLand('Model','ConstantGamma', ...
    'Gamma',gammaDB);
landSurface(scenario,'RadarReflectivity',refl);
```

Add two clutter regions to the scenario. Use the `clutterGenerator` object function to construct a clutter generator and enable clutter generation for the radar. The `Resolution` property defines the nominal spacing of clutter patches. Set this to be 1/5th of the range resolution to get multiple clutter patches per range gate. Set the range limit to 20 km. `UseBeam` indicates if clutter generation should be performed automatically for the mainlobe of the antenna pattern.

```
cluttergen = clutterGenerator(scenario,rdr,'Resolution', ...
    rangeRes/5,'RangeLimit',20e3,'UseBeam',true);
rgn1 = ringClutterRegion(cluttergen,1000,10000,30,45);
rgn2 = ringClutterRegion(cluttergen,1000,10000,30,105);
```

The `getClutterGenerator` function displays the two ring-shaped clutter regions belonging to the scenario.

```
getClutterGenerator(scenario,rdr)
```

```
ans =
    ClutterGenerator with properties:

        Resolution: 20
        RangeLimit: 20000
        UseBeam: 1
        UseShadowing: 1
        Regions: [1x2 radar.scenario.RingClutterRegion]
        Radar: [1x1 radarDataGenerator]
```

## Input Arguments

### **cluttergen** — Clutter generator

clutter generator object

Clutter generator, specified as a `ClutterGenerator` object. You can create a clutter generator using the `clutterGenerator` object function.

### **minrad** — Minimum radius of ring clutter region

scalar

Minimum radius of ring clutter region, specified as a scalar. Range is defined with respect the surface point directly below the radar. Units are in meters.

Data Types: double

### **maxrad — Maximum radius of ring clutter region**

scalar

Maximum radius of ring clutter region, specified as a scalar. Range is defined with respect the surface point directly below the radar. Units are in meters.

Data Types: double

### **azimuth\_span — Azimuth span of ring clutter region**

scalar

Azimuth span of ring clutter region, specified as a scalar. Units are in degrees.

In flat-Earth scenarios, azimuth angles are defined with respect to the global x-y coordinates system, with azimuth angle increasing counter-clockwise from the +x axis. In curved-Earth scenarios, azimuth angle is measured clockwise from North.

Data Types: double

### **azimuth\_center — Azimuth center of ring clutter region**

scalar

Azimuth center of ring clutter region, specified as a scalar. Units are in degrees.

In flat-Earth scenarios, azimuth angles are defined with respect to the global x-y coordinates system, with azimuth angle increasing counter-clockwise from the +x axis. In curved-Earth scenarios, azimuth angle is measured clockwise from North.

Data Types: double

## **Output Arguments**

### **cLuttergen — Ring clutter region**

ringClutterRegion object

Ring clutter region, returned as a RingClutterRegion object.

## **See Also**

ClutterGenerator | clutterGenerator | getClutterGenerator

**Introduced in R2022a**

## twoRayChannel

Two-ray propagation channel

### Description

The `twoRayChannel` models a narrowband two-ray propagation channel. A two-ray propagation channel is the simplest type of multipath channel. You can use a two-ray channel to simulate propagation of signals in a homogeneous, isotropic medium with a single reflecting boundary. This type of medium has two propagation paths: a line-of-sight (direct) propagation path from one point to another and a ray path reflected from the boundary. You can use this System object for short-range radar and mobile communications applications where the signals propagate along straight paths and the earth is assumed to be flat. You can also use this object for sonar and microphone applications. For acoustic applications, you can choose the fields to be non-polarized and adjust the propagation speed to be the speed of sound in air or water. You can use `twoRayChannel` to model propagation from several points simultaneously.

While the System object works for all frequencies, the attenuation models for atmospheric gases and rain are valid for electromagnetic signals in the frequency range 1–1000 GHz only. The attenuation model for fog and clouds is valid for 10–1000 GHz. Outside these frequency ranges, the System object uses the nearest valid value.

The `twoRayChannel` System object applies range-dependent time delays to the signals, and as well as gains or losses, phase shifts, and boundary reflection loss. The System object applies Doppler shift when either the source or destination is moving.

Signals at the channel output can be kept *separate* or be *combined* — controlled by the `CombinedRaysOutput` property. In the *separate* option, both fields arrive at the destination separately and are not combined. For the *combined* option, the two signals at the source propagate separately but are coherently summed at the destination into a single quantity. This option is convenient when the difference between the sensor or array gains in the directions of the two paths is not significant and need not be taken into account.

Unlike the `phased.FreeSpace` System object, the `twoRayChannel` System object does not support two-way propagation.

To compute the propagation delay for specified source and receiver points:

- 1 Define and set up your two-ray channel using the “Construction” on page 4-313 procedure that follows.
- 2 Call the `step` method to compute the propagated signal using the properties of the `twoRayChannel` System object.

The behavior of `step` is specific to each object in the toolbox.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

---



## Construction

`s2Ray = twoRayChannel` creates a two-ray propagation channel System object, `s2Ray`.

`s2Ray = twoRayChannel(Name, Value)` creates a System object, `s2Ray`, with each specified property `Name` set to the specified `Value`. You can specify additional name and value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

## Properties

### PropagationSpeed — Signal propagation speed

`physconst('LightSpeed')` (default) | positive scalar

Signal propagation speed, specified as a positive scalar. Units are in meters per second. The default propagation speed is the value returned by `physconst('LightSpeed')`. See `physconst` for more information.

Example: `3e8`

Data Types: `double`

### OperatingFrequency — Operating frequency

`300e6` (default) | positive scalar

Operating frequency, specified as a positive scalar. Units are in Hz.

Example: `1e9`

Data Types: `double`

### SpecifyAtmosphere — Enable atmospheric attenuation model

`false` (default) | `true`

Option to enable the atmospheric attenuation model, specified as a `false` or `true`. Set this property to `true` to add signal attenuation caused by atmospheric gases, rain, fog, or clouds. Set this property to `false` to ignore atmospheric effects in propagation.

Setting `SpecifyAtmosphere` to `true`, enables the `Temperature`, `DryAirPressure`, `WaterVapourDensity`, `LiquidWaterDensity`, and `RainRate` properties.

Data Types: `logical`

### Temperature — Ambient temperature

`15` (default) | real-valued scalar

Ambient temperature, specified as a real-valued scalar. Units are in degrees Celsius.

Example: `20.0`

### Dependencies

To enable this property, set `SpecifyAtmosphere` to `true`.

Data Types: `double`

### DryAirPressure — Atmospheric dry air pressure

`101.325e3` (default) | positive real-valued scalar

Atmospheric dry air pressure, specified as a positive real-valued scalar. Units are in pascals (Pa). The default value of this property corresponds to one standard atmosphere.

Example: 101.0e3

**Dependencies**

To enable this property, set `SpecifyAtmosphere` to `true`.

Data Types: `double`

**WaterVapourDensity — Atmospheric water vapor density**

7.5 (default) | positive real-valued scalar

Atmospheric water vapor density, specified as a positive real-valued scalar. Units are in  $\text{g/m}^3$ .

Example: 7.4

**Dependencies**

To enable this property, set `SpecifyAtmosphere` to `true`.

Data Types: `double`

**LiquidWaterDensity — Liquid water density**

0.0 (default) | nonnegative real-valued scalar

Liquid water density of fog or clouds, specified as a nonnegative real-valued scalar. Units are in  $\text{g/m}^3$ . Typical values for liquid water density are 0.05 for medium fog and 0.5 for thick fog.

Example: 0.1

**Dependencies**

To enable this property, set `SpecifyAtmosphere` to `true`.

Data Types: `double`

**RainRate — Rainfall rate**

0.0 (default) | nonnegative scalar

Rainfall rate, specified as a nonnegative scalar. Units are in mm/hr.

Example: 10.0

**Dependencies**

To enable this property, set `SpecifyAtmosphere` to `true`.

Data Types: `double`

**SampleRate — Sample rate of signal**

1e6 (default) | positive scalar

Sample rate of signal, specified as a positive scalar. Units are in Hz. The `System` object uses this quantity to calculate the propagation delay in units of samples.

Example: 1e6

Data Types: `double`

**EnablePolarization — Enable polarized fields**

false (default) | true

Option to enable polarized fields, specified as `false` or `true`. Set this property to `true` to enable polarization. Set this property to `false` to ignore polarization.

Data Types: logical

**GroundReflectionCoefficient — Ground reflection coefficient**-1 (default) | complex-valued scalar | complex-valued 1-by-*N* row vector

Ground reflection coefficient for the field at the reflection point, specified as a complex-valued scalar or a complex-valued 1-by-*N* row vector. Each coefficient has an absolute value less than or equal to one. The quantity *N* is the number of two-ray channels. Units are dimensionless. Use this property to model nonpolarized signals. To model polarized signals, use the `GroundRelativePermittivity` property.

Example: -0.5

**Dependencies**

To enable this property, set `EnablePolarization` to `false`.

Data Types: double

Complex Number Support: Yes

**GroundRelativePermittivity — Ground relative permittivity**15 (default) | positive real-valued scalar | real-valued 1-by-*N*row vector of positive values

Relative permittivity of the ground at the reflection point, specified as a positive real-valued scalar or a 1-by-*N* real-valued row vector of positive values. The dimension *N* is the number of two-ray channels. Permittivity units are dimensionless. Relative permittivity is defined as the ratio of actual ground permittivity to the permittivity of free space. This property applies when you set the `EnablePolarization` property to `true`. Use this property to model polarized signals. To model nonpolarized signals, use the `GroundReflectionCoefficient` property.

Example: 5

**Dependencies**

To enable this property, set `EnablePolarization` to `true`.

Data Types: double

**CombinedRaysOutput — Option to combine two rays at output**

true (default) | false

Option to combine the two rays at channel output, specified as `true` or `false`. When this property is `true`, the object coherently adds the line-of-sight propagated signal and the reflected path signal when forming the output signal. Use this mode when you do not need to include the directional gain of an antenna or array in your simulation.

Data Types: logical

**MaximumDistanceSource — Source of maximum one-way propagation distance**

'Auto' (default) | 'Property'

Source of maximum one-way propagation distance, specified as `'Auto'` or `'Property'`. The maximum one-way propagation distance is used to allocate sufficient memory for signal delay

computation. When you set this property to 'Auto', the System object automatically allocates memory. When you set this property to 'Property', you specify the maximum one-way propagation distance using the value of the `MaximumDistance` property.

Data Types: char

#### **MaximumDistance — Maximum one-way propagation distance**

10000 (default) | positive real-valued scalar

Maximum one-way propagation distance, specified as a positive real-valued scalar. Units are in meters. Any signal that propagates more than the maximum one-way distance is ignored. The maximum distance must be greater than or equal to the largest position-to-position distance.

Example: 5000

#### **Dependencies**

To enable this property, set the `MaximumDistanceSource` property to 'Property'.

Data Types: double

#### **MaximumNumInputSamplesSource — Source of maximum number of samples**

'Auto' (default) | 'Property'

The source of the maximum number of samples of the input signal, specified as 'Auto' or 'Property'. When you set this property to 'Auto', the propagation model automatically allocates enough memory to buffer the input signal. When you set this property to 'Property', you specify the maximum number of samples in the input signal using the `MaximumNumInputSamples` property. Any input signal longer than that value is truncated.

To use this object with variable-size signals in a MATLAB Function Block in Simulink, set the `MaximumNumInputSamplesSource` property to 'Property' and set a value for the `MaximumNumInputSamples` property.

Example: 'Property'

#### **Dependencies**

To enable this property, set `MaximumDistanceSource` to 'Property'.

Data Types: char

#### **MaximumNumInputSamples — Maximum number of input signal samples**

100 (default) | positive integer

Maximum number of input signal samples, specified as a positive integer. The input signal is the first argument of the `step` method, after the System object itself. The size of the input signal is the number of rows in the input matrix. Any input signal longer than this number is truncated. To process signals completely, ensure that this property value is greater than any maximum input signal length.

The waveform-generating System objects determine the maximum signal size:

- For any waveform, if the waveform `OutputFormat` property is set to 'Samples', the maximum signal length is the value specified in the `NumSamples` property.
- For pulse waveforms, if the `OutputFormat` is set to 'Pulses', the signal length is the product of the smallest pulse repetition frequency, the number of pulses, and the sample rate.

- For continuous waveforms, if the `OutputFormat` is set to `'Sweeps'`, the signal length is the product of the sweep time, the number of sweeps, and the sample rate.

Example: 2048

### Dependencies

To enable this property, set `MaximumNumInputSamplesSource` to `'Property'`.

Data Types: `double`

### Methods

`reset` Reset states of System object

`step` Propagate signal from point to point using two-ray channel model

#### Common to All System Objects

<code>release</code>	Allow System object property value changes
----------------------	--

### Examples

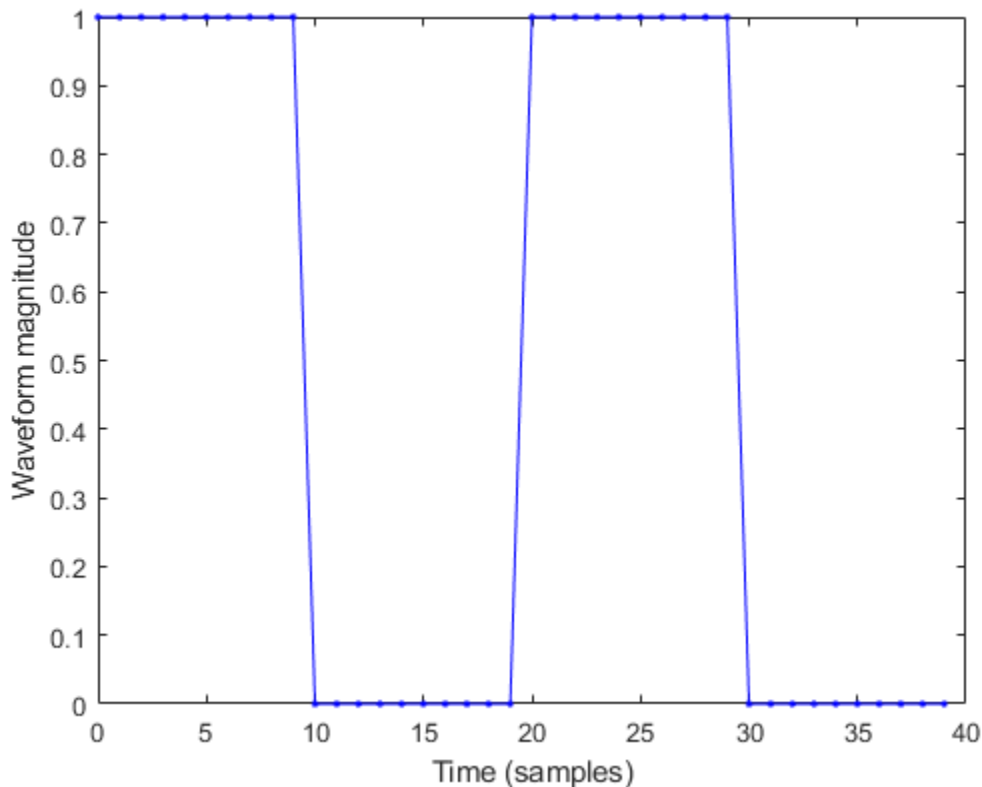
#### Scalar Field Propagating in Two-Ray Channel

This example illustrates the two-ray propagation of a signal, showing how the signals from the line-of-sight and reflected path arrive at the receiver at different times.

#### Create and Plot Propagating Signal

Create a nonpolarized electromagnetic field consisting of two rectangular waveform pulses at a carrier frequency of 100 MHz. Assume the pulse width is 10 ms and the sampling rate is 1 MHz. The bandwidth of the pulse is 0.1 MHz. Assume a 50% duty cycle in so that the pulse width is one-half the pulse repetition interval. Create a two-pulse wave train. Set the `GroundReflectionCoefficient` to 0.9 to model strong ground reflectivity. Propagate the field from a stationary source to a stationary receiver. The vertical separation of the source and receiver is approximately 10 km.

```
c = physconst('LightSpeed');
fs = 1e6;
pw = 10e-6;
pri = 2*pw;
PRF = 1/pri;
fc = 100e6;
lambda = c/fc;
waveform = phased.RectangularWaveform('SampleRate',fs,'PulseWidth',pw,...
    'PRF',PRF,'OutputFormat','Pulses','NumPulses',2);
wav = waveform();
n = size(wav,1);
figure;
plot((0:(n-1)),real(wav),'b.-');
xlabel('Time (samples)')
ylabel('Waveform magnitude')
```



### Specify the Location of Source and Receiver

Place the source and receiver about 1000 meters apart horizontally and approximately 10 km apart vertically.

```
pos1 = [1000;0;10000];
pos2 = [0;100;100];
vel1 = [0;0;0];
vel2 = [0;0;0];
```

Compute the predicted signal delays in units of samples.

```
[rng,ang] = rangeangle(pos2,pos1,'two-ray');
```

### Create a Two-Ray Channel System Object™

Create a two-ray propagation channel System object™ and propagate the signal along both the line-of-sight and reflected ray paths.

```
channel = twoRayChannel('SampleRate',fs,...
    'GroundReflectionCoefficient',.9,'OperatingFrequency',fc,...
    'CombinedRaysOutput',false);
prop_signal = channel([wav,wav],pos1,pos2,vel1,vel2);
```

### Plot the Propagated Signals

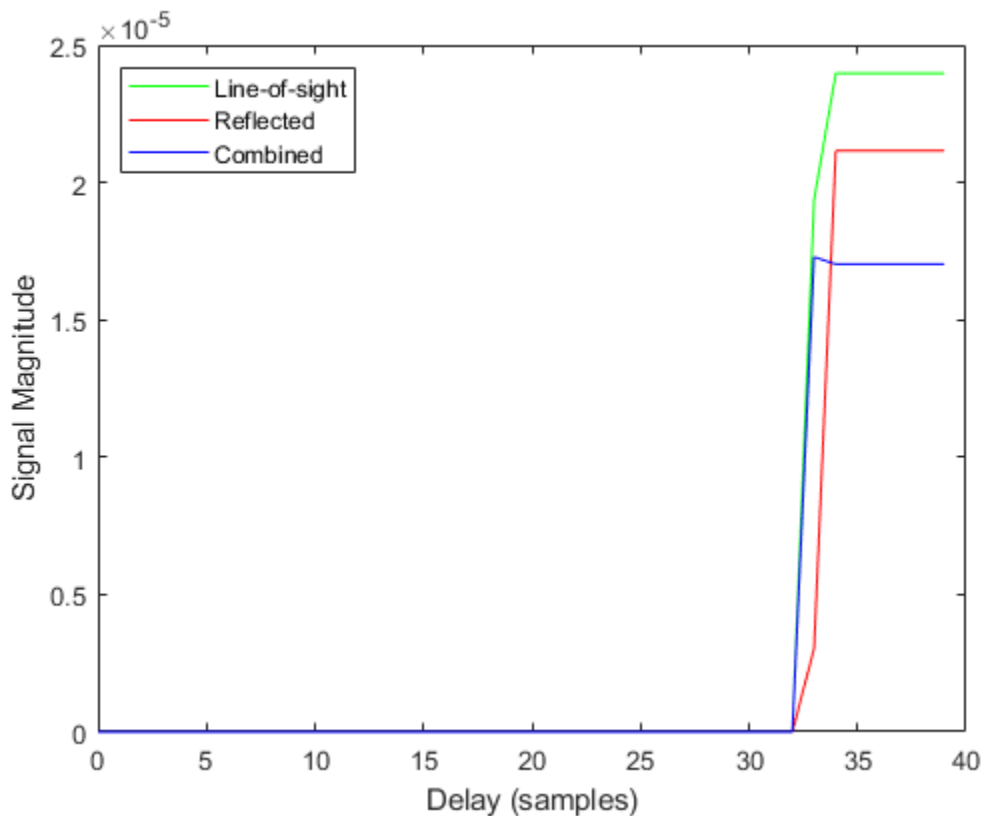
- Plot the signal propagated along the line-of-sight.

- Then, overlay a plot of the signal propagated along the reflected path.
- Finally, overlay a plot of the coherent sum of the two signals.

```

n = size(prop_signal,1);
delay = 0:(n-1);
plot(delay,abs(prop_signal(:,1)),'g')
hold on
plot(delay,abs(prop_signal(:,2)),'r')
plot(delay,abs(prop_signal(:,1) + prop_signal(:,2)),'b')
hold off
legend('Line-of-sight','Reflected','Combined','Location','NorthWest')
xlabel('Delay (samples)')
ylabel('Signal Magnitude')

```



The plot shows that the delay of the reflected path signal agrees with the predicted delay. The magnitude of the coherently combined signal is less than either of the propagated signals indicating that there is some interference between the two signals.

### Polarized Field Propagation in Two-Ray Channel

Create a polarized electromagnetic field consisting of linear FM waveform pulses. Propagate the field from a stationary source with a crossed-dipole antenna element to a stationary receiver approximately 10 km away. The transmitting antenna is 100 meters above the ground. The receiving antenna is 150 m above the ground. The receiving antenna is also a crossed-dipole. Plot the received signal.

### Set Radar Waveform Parameters

Assume the pulse width is  $10\mu\text{s}$  and the sampling rate is 10 MHz. The bandwidth of the pulse is 1 MHz. Assume a 50% duty cycle in which the pulse width is one-half the pulse repetition interval. Create a two-pulse wave train. Assume a carrier frequency of 100 MHz.

```
c = physconst('LightSpeed');
fs = 10e6;
pw = 10e-6;
pri = 2*pw;
PRF = 1/pri;
fc = 100e6;
bw = 1e6;
lambda = c/fc;
```

### Set Up Required System Objects

Use a GroundRelativePermittivity of 10.

```
waveform = phased.LinearFMWaveform('SampleRate',fs,'PulseWidth',pw,...
    'PRF',PRF,'OutputFormat','Pulses','NumPulses',2,'SweepBandwidth',bw,...
    'SweepDirection','Up','Envelope','Rectangular','SweepInterval',...
    'Positive');
antenna = phased.CrossedDipoleAntennaElement(...
    'FrequencyRange',[50,200]*1e6);
radiator = phased.Radiator('Sensor',antenna,'OperatingFrequency',fc,...
    'Polarization','Combined');
channel = twoRayChannel('SampleRate',fs,...
    'OperatingFrequency',fc,'CombinedRaysOutput',false,...
    'EnablePolarization',true,'GroundRelativePermittivity',10);
collector = phased.Collector('Sensor',antenna,'OperatingFrequency',fc,...
    'Polarization','Combined');
```

### Set Up Scene Geometry

Specify transmitter and receiver positions, velocities, and orientations. Place the source and receiver about 1000 m apart horizontally and approximately 50 m apart vertically.

```
posTx = [0;100;100];
posRx = [1000;0;150];
velTx = [0;0;0];
velRx = [0;0;0];
laxRx = rotz(180);
laxTx = rotx(1)*eye(3);
```

### Create and Radiate Signals from Transmitter

Compute the transmission angles for the two rays traveling toward the receiver. These angles are defined with respect to the transmitter local coordinate system. The `phased.Radiator` System object™ uses these angles to apply separate antenna gains to the two signals.

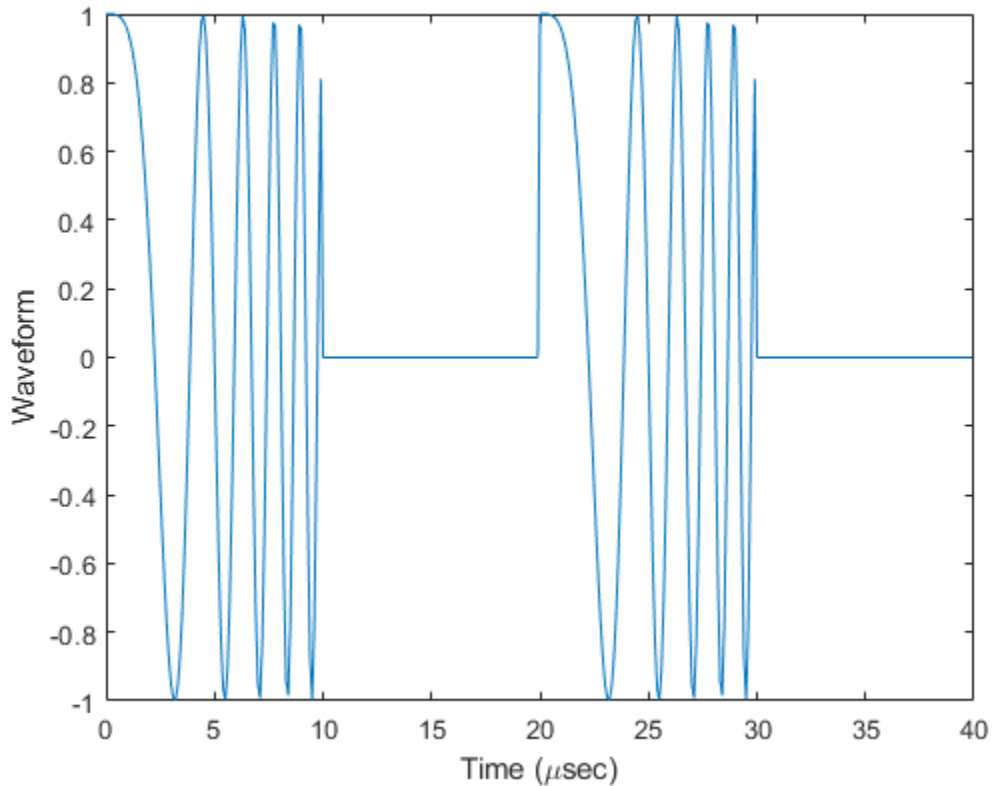
```
[rng,angsTx] = rangeangle(posRx,posTx,laxTx,'two-ray');
wav = waveform();
```

Plot the transmitted Waveform

```
n = size(wav,1);
plot((0:(n-1))/fs*1000000,real(wav))
```



```
xlabel('Time ({\mu}sec)')
ylabel('Waveform')
```



```
sig = radiator(wav,angsTx,laxTx);
```

Propagate signals to receiver via two-ray channel

```
prop_sig = channel(sig,posTx,posRx,velTx,velRx);
```

### Receive Propagated Signal

Compute the reception angles for the two rays arriving at the receiver. These angles are defined with respect to the receiver local coordinate system. The `phased.Collector System` object™ uses these angles to apply separate antenna gains to the two signals.

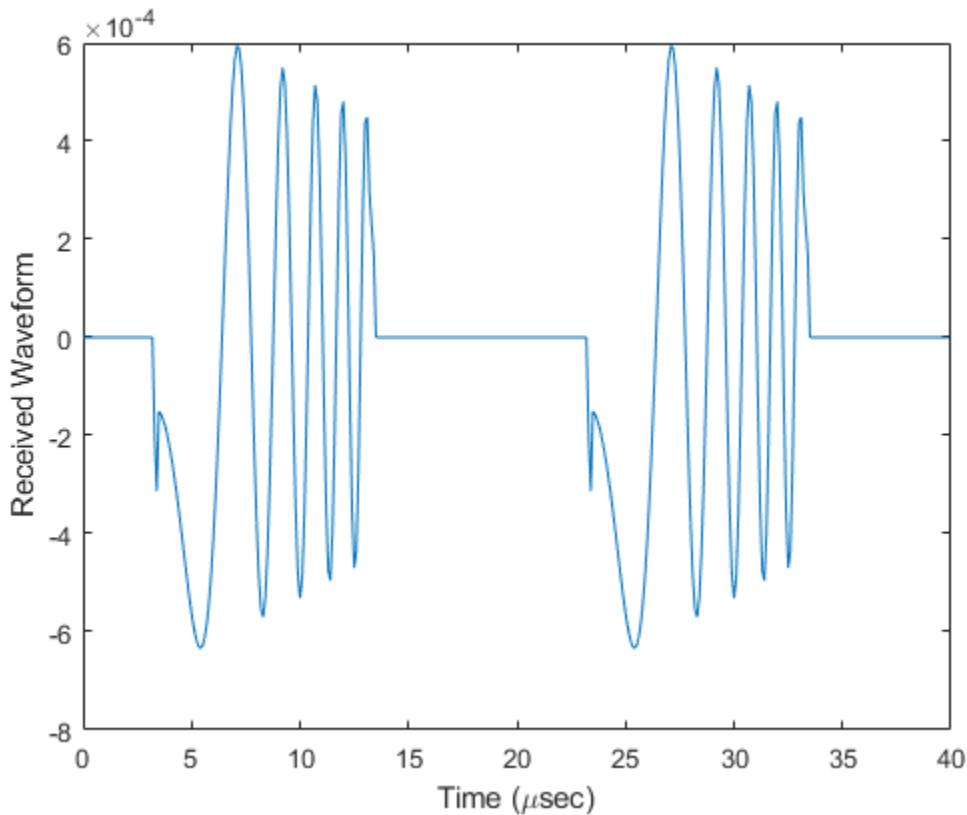
```
[~,angsRx] = rangeangle(posTx,posRx,laxRx,'two-ray');
```

Collect and combine received rays.

```
y = collector(prop_sig,angsRx,laxRx);
```

### Plot received waveform

```
plot((0:(n-1))/fs*1000000,real(y))
xlabel('Time ({\mu}sec)')
ylabel('Received Waveform')
```



## More About

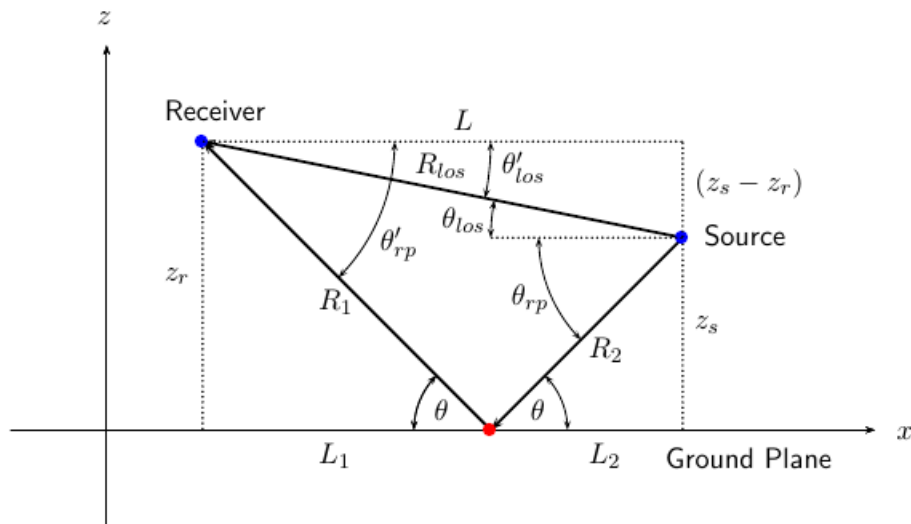
### Two-Ray Propagation Paths

A two-ray propagation channel is the next step up in complexity from a free-space channel and is the simplest case of a multipath propagation environment. The free-space channel models a straight-line *line-of-sight* path from point 1 to point 2. In a two-ray channel, the medium is specified as a homogeneous, isotropic medium with a reflecting planar boundary. The boundary is always set at  $z = 0$ . There are at most two rays propagating from point 1 to point 2. The first ray path propagates along the same line-of-sight path as in the free-space channel. The line-of-sight path is often called the *direct path*. The second ray reflects off the boundary before propagating to point 2. According to the Law of Reflection, the angle of reflection equals the angle of incidence. In short-range simulations such as cellular communications systems and automotive radars, you can assume that the reflecting surface, the ground or ocean surface, is flat.

The `twoRayChannel` and `widebandTwoRayChannel` System objects model propagation time delay, phase shift, Doppler shift, and loss effects for both paths. For the reflected path, loss effects include reflection loss at the boundary.

The figure illustrates two propagation paths. From the source position,  $s_s$ , and the receiver position,  $s_r$ , you can compute the arrival angles of both paths,  $\theta_{los}$  and  $\theta_{rp}$ . The arrival angles are the elevation and azimuth angles of the arriving radiation with respect to a local coordinate system. In this case, the local coordinate system coincides with the global coordinate system. You can also compute the transmitting angles,  $\theta_{los}$  and  $\theta_{rp}$ . In the global coordinates, the angle of reflection at the boundary is the same as the angles  $\theta_{rp}$  and  $\theta'_{rp}$ . The reflection angle is important to know when you use angle-

dependent reflection-loss data. You can determine the reflection angle by using the rangeangle function and setting the reference axes to the global coordinate system. The total path length for the line-of-sight path is shown in the figure by  $R_{los}$  which is equal to the geometric distance between source and receiver. The total path length for the reflected path is  $R_{rp} = R_1 + R_2$ . The quantity  $L$  is the ground range between source and receiver.



You can easily derive exact formulas for path lengths and angles in terms of the ground range and object heights in the global coordinate system.

$$\begin{aligned}\vec{R} &= \vec{x}_s - \vec{x}_r \\ R_{los} &= |\vec{R}| = \sqrt{(z_r - z_s)^2 + L^2} \\ R_1 &= \frac{z_r}{z_r + z_s} \sqrt{(z_r + z_s)^2 + L^2} \\ R_2 &= \frac{z_s}{z_s + z_r} \sqrt{(z_r + z_s)^2 + L^2} \\ R_{rp} &= R_1 + R_2 = \sqrt{(z_r + z_s)^2 + L^2} \\ \tan\theta_{los} &= \frac{(z_s - z_r)}{L} \\ \tan\theta_{rp} &= -\frac{(z_s + z_r)}{L} \\ \theta'_{los} &= -\theta_{los} \\ \theta'_{rp} &= \theta_{rp}\end{aligned}$$

## Two-Ray Attenuation

Attenuation or path loss in the two-ray channel is the product of five components,  $L = L_{\text{tworay}} L_G L_g L_c L_r$ , where

- $L_{\text{tworay}}$  is the two-ray geometric path attenuation
- $L_G$  is the ground reflection attenuation
- $L_g$  is the atmospheric path attenuation
- $L_c$  is the fog and cloud path attenuation
- $L_r$  is the rain path attenuation

Each component is in magnitude units, not in dB.

## Ground Reflection and Propagation Loss

Losses occurs when a signal is reflected from a boundary. You can obtain a simple model of ground reflection loss by representing the electromagnetic field as a scalar field. This approach also works for acoustic and sonar systems. Let  $E$  be a scalar free-space electromagnetic field having amplitude  $E_0$  at a reference distance  $R_0$  from a transmitter (for example, one meter). The propagating free-space field at distance  $R_{\text{los}}$  from the transmitter is

$$E_{\text{los}} = E_0 \left( \frac{R_0}{R_{\text{los}}} \right) e^{i\omega(t - R_{\text{los}}/c)}$$

for the line-of-sight path. You can express the ground-reflected  $E$ -field as

$$E_{\text{rp}} = L_G E_0 \left( \frac{R_0}{R_{\text{rp}}} \right) e^{i\omega(t - R_{\text{rp}}/c)}$$

where  $R_{\text{rp}}$  is the reflected path distance. The quantity  $L_G$  represents the loss due to reflection at the ground plane. To specify  $L_G$ , use the `GroundReflectionCoefficient` property. In general,  $L_G$  depends on the incidence angle of the field. If you have empirical information about the angular dependence of  $L_G$ , you can use `rangeangle` to compute the incidence angle of the reflected path. The total field at the destination is the sum of the line-of-sight and reflected-path fields.

For electromagnetic waves, a more complicated but more realistic model uses a vector representation of the polarized field. You can decompose the incident electric field into two components. One component,  $E_p$ , is parallel to the plane of incidence. The other component,  $E_s$ , is perpendicular to the plane of incidence. The ground reflection coefficients for these components differ and can be written in terms of the ground permittivity and incidence angle.

$$G_p = \frac{Z_1 \cos \theta_1 - Z_2 \cos \theta_2}{Z_1 \cos \theta_1 + Z_2 \cos \theta_2} = \frac{\cos \theta_1 - \frac{Z_2}{Z_1} \cos \theta_2}{\cos \theta_1 + \frac{Z_2}{Z_1} \cos \theta_2}$$

$$G_s = \frac{Z_2 \cos \theta_1 - Z_1 \cos \theta_2}{Z_2 \cos \theta_1 + Z_1 \cos \theta_2} = \frac{\cos \theta_2 - \frac{Z_2}{Z_1} \cos \theta_1}{\cos \theta_2 + \frac{Z_2}{Z_1} \cos \theta_1}$$

$$Z_1 = \sqrt{\frac{\mu_1}{\varepsilon_1}}$$

$$Z_2 = \sqrt{\frac{\mu_2}{\varepsilon_2}}$$

where  $Z$  is the impedance of the medium. Because the magnetic permeability of the ground is almost identical to that of air or free space, the ratio of impedances depends primarily on the ratio of electric permittivities

$$G_p = \frac{\sqrt{\rho} \cos \theta_1 - \cos \theta_2}{\sqrt{\rho} \cos \theta_1 + \cos \theta_2}$$

$$G_s = \frac{\sqrt{\rho} \cos \theta_2 - \cos \theta_1}{\sqrt{\rho} \cos \theta_2 + \cos \theta_1}$$

where the quantity  $\rho = \varepsilon_2/\varepsilon_1$  is the *ground relative permittivity* set by the `GroundRelativePermittivity` property. The angle  $\theta_1$  is the incidence angle and the angle  $\theta_2$  is the refraction angle at the boundary. You can determine  $\theta_2$  using Snell's law of refraction.

After reflection, the full field is reconstructed from the parallel and perpendicular components. The total ground plane attenuation,  $L_G$ , is a combination of  $G_s$  and  $G_p$ .

When the origin and destination are stationary relative to each other, you can write the output  $Y$  of step as  $Y(t) = F(t-\tau)/L$ . The quantity  $\tau$  is the signal delay and  $L$  is the free-space path loss. The delay  $\tau$  is given by  $R/c$ .  $R$  is either the line-of-sight propagation path distance or the reflected path distance, and  $c$  is the propagation speed. The path loss

where  $\lambda$  is the signal wavelength.

### Atmospheric Gas Attenuation Model

This model calculates the attenuation of signals that propagate through atmospheric gases.

Electromagnetic signals attenuate when they propagate through the atmosphere. This effect is due primarily to the absorption resonance lines of oxygen and water vapor, with smaller contributions coming from nitrogen gas. The model also includes a continuous absorption spectrum below 10 GHz. The ITU model *Recommendation ITU-R P676-10: Attenuation by atmospheric gases* is used. The model computes the specific attenuation (attenuation per kilometer) as a function of temperature, pressure, water vapor density, and signal frequency. The atmospheric gas model is valid for frequencies from 1-1000 GHz and applies to polarized and nonpolarized fields.

The formula for specific attenuation at each frequency is

$$\gamma = \gamma_o(f) + \gamma_w(f) = 0.1820fN''(f).$$

The quantity  $N''()$  is the imaginary part of the complex atmospheric refractivity and consists of a spectral line component and a continuous component:

$$N''(f) = \sum_i S_i F_i + N''_D(f)$$

The spectral component consists of a sum of discrete spectrum terms composed of a localized frequency bandwidth function,  $F(f)_i$ , multiplied by a spectral line strength,  $S_i$ . For atmospheric oxygen, each spectral line strength is

$$S_i = a_1 \times 10^{-7} \left( \frac{300}{T} \right)^3 \exp \left[ a_2 \left( 1 - \left( \frac{300}{T} \right) \right) \right] P.$$

For atmospheric water vapor, each spectral line strength is

$$S_i = b_1 \times 10^{-1} \left( \frac{300}{T} \right)^{3.5} \exp \left[ b_2 \left( 1 - \left( \frac{300}{T} \right) \right) \right] W.$$

$P$  is the dry air pressure,  $W$  is the water vapor partial pressure, and  $T$  is the ambient temperature. Pressure units are in hectoPascals (hPa) and temperature is in degrees Kelvin. The water vapor partial pressure,  $W$ , is related to the water vapor density,  $\rho$ , by

$$W = \frac{\rho T}{216.7}.$$

The total atmospheric pressure is  $P + W$ .

For each oxygen line,  $S_i$  depends on two parameters,  $a_1$  and  $a_2$ . Similarly, each water vapor line depends on two parameters,  $b_1$  and  $b_2$ . The ITU documentation cited at the end of this section contains tabulations of these parameters as functions of frequency.

The localized frequency bandwidth functions  $F_i(f)$  are complicated functions of frequency described in the ITU references cited below. The functions depend on empirical model parameters that are also tabulated in the reference.

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the path length,  $R$ . Then, the total attenuation is  $L_g = R(\gamma_o + \gamma_w)$ .

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands, and apply attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

### Fog and Cloud Attenuation Model

This model calculates the attenuation of signals that propagate through fog or clouds.

Fog and cloud attenuation are the same atmospheric phenomenon. The ITU model, *Recommendation ITU-R P.840-6: Attenuation due to clouds and fog* is used. The model computes the specific attenuation (attenuation per kilometer), of a signal as a function of liquid water density, signal frequency, and temperature. The model applies to polarized and nonpolarized fields. The formula for specific attenuation at each frequency is

$$\gamma_c = K_l(f)M,$$

where  $M$  is the liquid water density in  $\text{gm/m}^3$ . The quantity  $K_l(f)$  is the specific attenuation coefficient and depends on frequency. The cloud and fog attenuation model is valid for frequencies 10–1000 GHz. Units for the specific attenuation coefficient are  $(\text{dB/km})/(\text{g/m}^3)$ .

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the path length  $R$ . Total attenuation is  $L_c = R\gamma_c$ .

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands, and apply narrowband attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

### Rainfall Attenuation Model

This model calculates the attenuation of signals that propagate through regions of rainfall. Rain attenuation is a dominant fading mechanism and can vary from location-to-location and from year-to-year.

Electromagnetic signals are attenuated when propagating through a region of rainfall. Rainfall attenuation is computed according to the ITU rainfall model *Recommendation ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. The model computes the specific attenuation (attenuation per kilometer) of a signal as a function of rainfall rate, signal frequency, polarization, and path elevation angle. The specific attenuation,  $\gamma_R$ , is modeled as a power law with respect to rain rate

$$\gamma_R = kR^\alpha,$$

where  $R$  is rain rate. Units are in mm/hr. The parameter  $k$  and exponent  $\alpha$  depend on the frequency, the polarization state, and the elevation angle of the signal path. The specific attenuation model is valid for frequencies from 1–1000 GHz.

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the an effective propagation distance,  $d_{\text{eff}}$ . Then, the total attenuation is  $L = d_{\text{eff}}\gamma_R$ .

The effective distance is the geometric distance,  $d$ , multiplied by a scale factor

$$r = \frac{1}{0.477d^{0.633}R_{0.01}^{0.073\alpha}f^{0.123} - 10.579(1 - \exp(-0.024d))}$$

where  $f$  is the frequency. The article *Recommendation ITU-R P.530-17 (12/2017): Propagation data and prediction methods required for the design of terrestrial line-of-sight systems* presents a complete discussion for computing attenuation.

The rain rate,  $R$ , used in these computations is the long-term statistical rain rate,  $R_{0.01}$ . This is the rain rate that is exceeded 0.01% of the time. The calculation of the statistical rain rate is discussed in *Recommendation ITU-R P.837-7 (06/2017): Characteristics of precipitation for propagation modelling*. This article also explains how to compute the attenuation for other percentages from the 0.01% value.

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands and apply attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

## References

- [1] Saakian, A. *Radio Wave Propagation Fundamentals*. Norwood, MA: Artech House, 2011.
- [2] Balanis, C. *Advanced Engineering Electromagnetics*. New York: Wiley & Sons, 1989.
- [3] Rappaport, T. *Wireless Communications: Principles and Practice, 2nd Ed* New York: Prentice Hall, 2002.
- [4] Radiocommunication Sector of the International Telecommunication Union. *Recommendation ITU-R P.676-10: Attenuation by atmospheric gases*. 2013.
- [5] Radiocommunication Sector of the International Telecommunication Union. *Recommendation ITU-R P.840-6: Attenuation due to clouds and fog*. 2013.
- [6] Radiocommunication Sector of the International Telecommunication Union. *Recommendation ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. 2005.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Functions

`fogpl` | `fspl` | `gaspl` | `rainpl` | `rangeangle`

### Objects

`phased.FreeSpace` | `phased.LOSChannel` | `phased.RadarTarget` |  
`phased.WidebandFreeSpace` | `phased.WidebandLOSChannel` | `widebandTwoRayChannel`

**Introduced in R2021a**



# reset

**System object:** twoRayChannel

Reset states of System object

## Syntax

```
reset(s2Ray)
```

## Description

`reset(s2Ray)` resets the internal state of the `twoRayChannel` object, `S`. This method resets the random number generator state if `SeedSource` is a property of this System object and has the value 'Property'.

## Input Arguments

**s2Ray — Two-ray channel**

System object

Two-ray channel, specified as a System object.

Example: `twoRayChannel`

## Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**Introduced in R2021a**

## step

**System object:** twoRayChannel

Propagate signal from point to point using two-ray channel model

### Syntax

```
prop_sig = step(channel,sig,origin_pos,dest_pos,origin_vel,dest_vel)
```

### Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

`prop_sig = step(channel,sig,origin_pos,dest_pos,origin_vel,dest_vel)` returns the resulting signal, `prop_sig`, when a narrowband signal, `sig`, propagates through a two-ray channel from the `origin_pos` position to the `dest_pos` position. Either the `origin_pos` or `dest_pos` arguments can have multiple points but you cannot specify both as having multiple points. The velocity of the signal origin is specified in `origin_vel` and the velocity of the signal destination is specified in `dest_vel`. The dimensions of `origin_vel` and `dest_vel` must agree with the dimensions of `origin_pos` and `dest_pos`, respectively.

Electromagnetic fields propagated through a two-ray channel can be polarized or nonpolarized. For nonpolarized fields, such as an acoustic field, the propagating signal field, `sig`, is a vector or matrix. When the fields are polarized, `sig` is an array of structures. Every structure element represents an electric field vector in Cartesian form.

In the two-ray environment, there are two signal paths connecting every signal origin and destination pair. For  $N$  signal origins (or  $N$  signal destinations), there are  $2N$  number of paths. The signals for each origin-destination pair do not have to be related. The signals along the two paths for any single source-destination pair can also differ due to phase or amplitude differences.

You can keep the two signals at the destination *separate* or *combined* — controlled by the `CombinedRaysOutput` property. *Combined* means that the signals at the source propagate separately along the two paths but are coherently summed at the destination into a single quantity. To use the *separate* option, set `CombinedRaysOutput` to `false`. To use the *combined* option, set `CombinedRaysOutput` to `true`. This option is convenient when the difference between the sensor or array gains in the directions of the two paths is not significant and need not be taken into account.

---

**Note** The object performs an initialization the first time the object is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

## Input Arguments

### channel — Two-ray channel

System object

Two-ray channel, specified as a System object.

Example: `twoRayChannel`

### sig — Narrowband signal

$M$ -by- $N$  complex-valued matrix |  $M$ -by- $2N$  complex-valued matrix | 1-by- $N$  struct array containing complex-valued fields | 1-by- $2N$  struct array containing complex-valued fields

- Narrowband nonpolarized scalar signal, specified as an
  - $M$ -by- $N$  complex-valued matrix. Each column contains a common signal propagated along both the line-of-sight path and the reflected path. You can use this form when both path signals are the same.
  - $M$ -by- $2N$  complex-valued matrix. Each adjacent pair of columns represents a different channel. Within each pair, the first column represents the signal propagated along the line-of-sight path and the second column represents the signal propagated along the reflected path.
- Narrowband polarized signal, specified as a
  - 1-by- $N$  struct array containing complex-valued fields. Each struct contains a common polarized signal propagated along both the line-of-sight path and the reflected path. Each structure element contains an  $M$ -by-1 column vector of electromagnetic field components (`sig.X`, `sig.Y`, `sig.Z`). You can use this form when both path signals are the same.
  - 1-by- $2N$  struct array containing complex-valued fields. Each adjacent pair of array columns represents a different channel. Within each pair, the first column represents the signal along the line-of-sight path and the second column represents the signal along the reflected path. Each structure element contains an  $M$ -by-1 column vector of electromagnetic field components (`sig.X`, `sig.Y`, `sig.Z`).

For nonpolarized fields, the quantity  $M$  is the number of samples of the signal and  $N$  is the number of two-ray channels. Each channel corresponds to a source-destination pair.

The size of the first dimension of the input matrix can vary to simulate a changing signal length. A size change can occur, for example, in the case of a pulse waveform with variable pulse repetition frequency.

For polarized fields, the struct element contains three  $M$ -by-1 complex-valued column vectors, `sig.X`, `sig.Y`, and `sig.Z`. These vectors represent the  $x$ ,  $y$ , and  $z$  Cartesian components of the polarized signal.

The size of the first dimension of the matrix fields within the struct can vary to simulate a changing signal length such as a pulse waveform with variable pulse repetition frequency.

Example: `[1,1;j,1;0.5,0]`

Data Types: `double`

Complex Number Support: Yes

### origin\_pos — Origin of the signal or signals

3-by-1 real-valued column vector | 3-by- $N$  real-valued matrix

Origin of the signal or signals, specified as a 3-by-1 real-valued column vector or 3-by- $N$  real-valued matrix. The quantity  $N$  is the number of two-ray channels. If `origin_pos` is a column vector, it takes the form  $[x;y;z]$ . If `origin_pos` is a matrix, each column specifies a different signal origin and has the form  $[x;y;z]$ . Position units are meters.

`origin_pos` and `dest_pos` cannot both be specified as matrices — at least one must be a 3-by-1 column vector.

Example: `[1000;100;500]`

Data Types: `double`

### **dest\_pos — Destination position of the signal or signals**

3-by-1 real-valued column vector | 3-by- $N$  real-valued matrix

Destination position of the signal or signals, specified as a 3-by-1 real-valued column vector or 3-by- $N$  real-valued matrix. The quantity  $N$  is the number of two-ray channels propagating from or to  $N$  signal origins. If `dest_pos` is a 3-by-1 column vector, it takes the form  $[x;y;z]$ . If `dest_pos` is a matrix, each column specifies a different signal destination and takes the form  $[x;y;z]$ . Position units are in meters.

You cannot specify `origin_pos` and `dest_pos` as matrices. At least one must be a 3-by-1 column vector.

Example: `[0;0;0]`

Data Types: `double`

### **origin\_vel — Velocity of signal origin**

3-by-1 real-valued column vector | 3-by- $N$  real-valued matrix

Velocity of signal origin, specified as a 3-by-1 real-valued column vector or 3-by- $N$  real-valued matrix. The dimensions of `origin_vel` must match the dimensions of `origin_pos`. If `origin_vel` is a column vector, it takes the form  $[V_x;V_y;V_z]$ . If `origin_vel` is a 3-by- $N$  matrix, each column specifies a different origin velocity and has the form  $[V_x;V_y;V_z]$ . Velocity units are in meters per second.

Example: `[10;0;5]`

Data Types: `double`

### **dest\_vel — Velocity of signal destinations**

3-by-1 real-valued column vector | 3-by- $N$  real-valued matrix

Velocity of signal destinations, specified as a 3-by-1 real-valued column vector or 3-by- $N$  real-valued matrix. The dimensions of `dest_vel` must match the dimensions of `dest_pos`. If `dest_vel` is a column vector, it takes the form  $[V_x;V_y;V_z]$ . If `dest_vel` is a 3-by- $N$  matrix, each column specifies a different destination velocity and has the form  $[V_x;V_y;V_z]$ . Velocity units are in meters per second.

Example: `[0;0;0]`

Data Types: `double`

## **Output Arguments**

### **prop\_sig — Propagated signal**

$M$ -by- $N$  complex-valued matrix |  $M$ -by- $2N$  complex-valued matrix | 1-by- $N$  struct array containing complex-valued fields | 1-by- $2N$  struct array containing complex-valued fields

- Narrowband nonpolarized scalar signal, returned as an:
  - $M$ -by- $N$  complex-valued matrix. To return this format, set the `CombinedRaysOutput` property to `true`. Each matrix column contains the coherently combined signals from the line-of-sight path and the reflected path.
  - $M$ -by- $2N$  complex-valued matrix. To return this format set the `CombinedRaysOutput` property to `false`. Alternate columns of the matrix contain the signals from the line-of-sight path and the reflected path.
- Narrowband polarized scalar signal, returned as:
  - 1-by- $N$  struct array containing complex-valued fields. To return this format, set the `CombinedRaysOutput` property to `true`. Each column of the array contains the coherently combined signals from the line-of-sight path and the reflected path. Each structure element contains the electromagnetic field vector (`prop_sig.X`,`prop_sig.Y`,`prop_sig.Z`).
  - 1-by- $2N$  struct array containing complex-valued fields. To return this format, set the `CombinedRaysOutput` property to `false`. Alternate columns contains the signals from the line-of-sight path and the reflected path. Each structure element contains the electromagnetic field vector (`prop_sig.X`,`prop_sig.Y`,`prop_sig.Z`).

The output `prop_sig` contains signal samples arriving at the signal destination within the current input time frame. Whenever it takes longer than the current time frame for the signal to propagate from the origin to the destination, the output may not contain all contributions from the input of the current time frame. The remaining output will appear in the next call to `step`.

## Examples

### Compare Two-Ray with Free Space Propagation

Propagate a signal in a two-ray channel environment from a radar at (0,0,10) meters to a target at (300,200,30) meters. Assume that the radar and target are stationary and that the transmitting antenna has a cosine pattern. Compare the combined signals from the two paths with the single signal resulting from free space propagation. Set the `CombinedRaysOutput` to `true` to produce a combined propagated signal.

#### Create a Rectangular Waveform

Set the sample rate to 2 MHz.

```
fs = 2e6;
waveform = phased.RectangularWaveform('SampleRate',fs);
wavfrm = waveform();
```

#### Create the Transmitting Antenna and Radiator

Set up a `phased.Radiator System` object™ to transmit from a cosine antenna

```
antenna = phased.CosineAntennaElement;
radiator = phased.Radiator('Sensor',antenna);
```

#### Specify Transmitter and Target Coordinates

```
posTx = [0;0;10];
posTgt = [300;200;30];
```

```
velTx = [0;0;0];  
velTgt = [0;0;0];
```

### Free Space Propagation

Compute the transmitting direction toward the target for the free-space model. Then, radiate the signal.

```
[~,angFS] = rangeangle(posTgt,posTx);  
wavTx = radiator(wavfrm,angFS);
```

Propagate the signal to the target.

```
fchannel = phased.FreeSpace('SampleRate',waveform.SampleRate);  
yfs = fchannel(wavTx,posTx,posTgt,velTx,velTgt);  
release(radiator);
```

### Two-Ray Propagation

Compute the two transmit angles toward the target for line-of-sight (LOS) path and reflected paths. Compute the transmitting directions toward the target for the two rays. Then, radiate the signals.

```
[~,angTwoRay] = rangeangle(posTgt,posTx,'two-ray');  
wavTwoRay = radiator(wavfrm,angTwoRay);
```

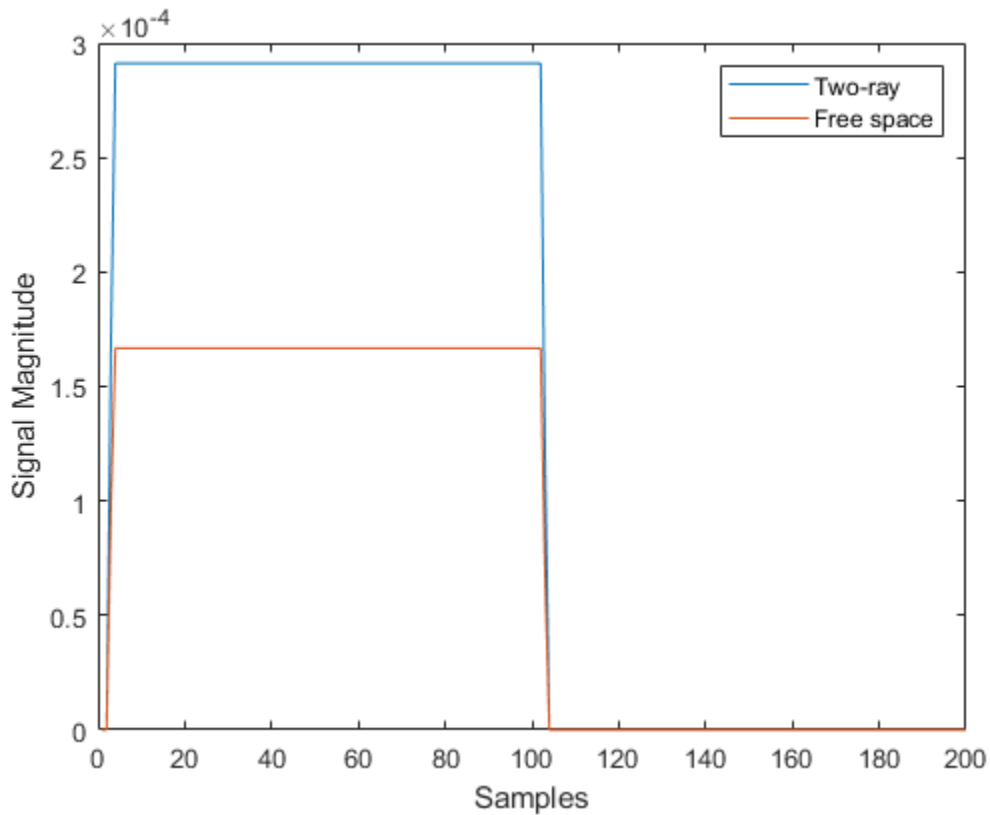
Propagate the signals to the target.

```
channel = twoRayChannel('SampleRate',waveform.SampleRate,...  
    'CombinedRaysOutput',true);  
y2ray = channel(wavTwoRay,posTx,posTgt,velTx,velTgt);
```

### Plot the Propagated Signals

Plot the combined signal against the free-space signal

```
plot(abs([y2ray yfs]))  
legend('Two-ray','Free space')  
xlabel('Samples')  
ylabel('Signal Magnitude')
```



### Polarized Field Propagation in Two-Ray Channel

Create a polarized electromagnetic field consisting of linear FM waveform pulses. Propagate the field from a stationary source with a crossed-dipole antenna element to a stationary receiver approximately 10 km away. The transmitting antenna is 100 meters above the ground. The receiving antenna is 150 m above the ground. The receiving antenna is also a crossed-dipole. Plot the received signal.

#### Set Radar Waveform Parameters

Assume the pulse width is  $10\mu\text{s}$  and the sampling rate is 10 MHz. The bandwidth of the pulse is 1 MHz. Assume a 50% duty cycle in which the pulse width is one-half the pulse repetition interval. Create a two-pulse wave train. Assume a carrier frequency of 100 MHz.

```
c = physconst('LightSpeed');
fs = 10e6;
pw = 10e-6;
pri = 2*pw;
PRF = 1/pri;
fc = 100e6;
bw = 1e6;
lambda = c/fc;
```

#### Set Up Required System Objects

Use a GroundRelativePermittivity of 10.

```

waveform = phased.LinearFMWaveform('SampleRate',fs,'PulseWidth',pw,...
    'PRF',PRF,'OutputFormat','Pulses','NumPulses',2,'SweepBandwidth',bw,...
    'SweepDirection','Up','Envelope','Rectangular','SweepInterval',...
    'Positive');
antenna = phased.CrossedDipoleAntennaElement(...
    'FrequencyRange',[50,200]*1e6);
radiator = phased.Radiator('Sensor',antenna,'OperatingFrequency',fc,...
    'Polarization','Combined');
channel = twoRayChannel('SampleRate',fs,...
    'OperatingFrequency',fc,'CombinedRaysOutput',false,...
    'EnablePolarization',true,'GroundRelativePermittivity',10);
collector = phased.Collector('Sensor',antenna,'OperatingFrequency',fc,...
    'Polarization','Combined');

```

### Set Up Scene Geometry

Specify transmitter and receiver positions, velocities, and orientations. Place the source and receiver about 1000 m apart horizontally and approximately 50 m apart vertically.

```

posTx = [0;100;100];
posRx = [1000;0;150];
velTx = [0;0;0];
velRx = [0;0;0];
laxRx = rotz(180);
laxTx = rotx(1)*eye(3);

```

### Create and Radiate Signals from Transmitter

Compute the transmission angles for the two rays traveling toward the receiver. These angles are defined with respect to the transmitter local coordinate system. The `phased.Radiator System` object™ uses these angles to apply separate antenna gains to the two signals.

```

[rng,angsTx] = rangeangle(posRx,posTx,laxTx,'two-ray');
wav = waveform();

```

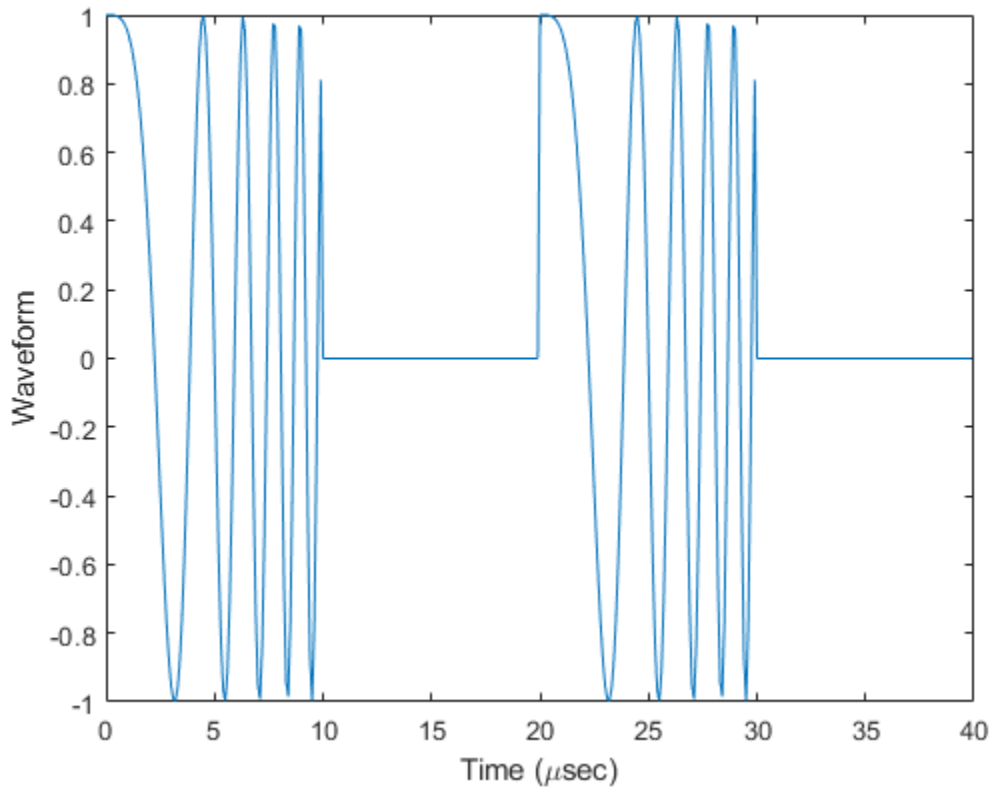
Plot the transmitted Waveform

```

n = size(wav,1);
plot((0:(n-1))/fs*1000000,real(wav))
xlabel('Time ({\mu}sec)')
ylabel('Waveform')

```





```
sig = radiator(wav,angsTx,laxTx);
```

Propagate signals to receiver via two-ray channel

```
prop_sig = channel(sig,posTx,posRx,velTx,velRx);
```

### Receive Propagated Signal

Compute the reception angles for the two rays arriving at the receiver. These angles are defined with respect to the receiver local coordinate system. The `phased.Collector System` object™ uses these angles to apply separate antenna gains to the two signals.

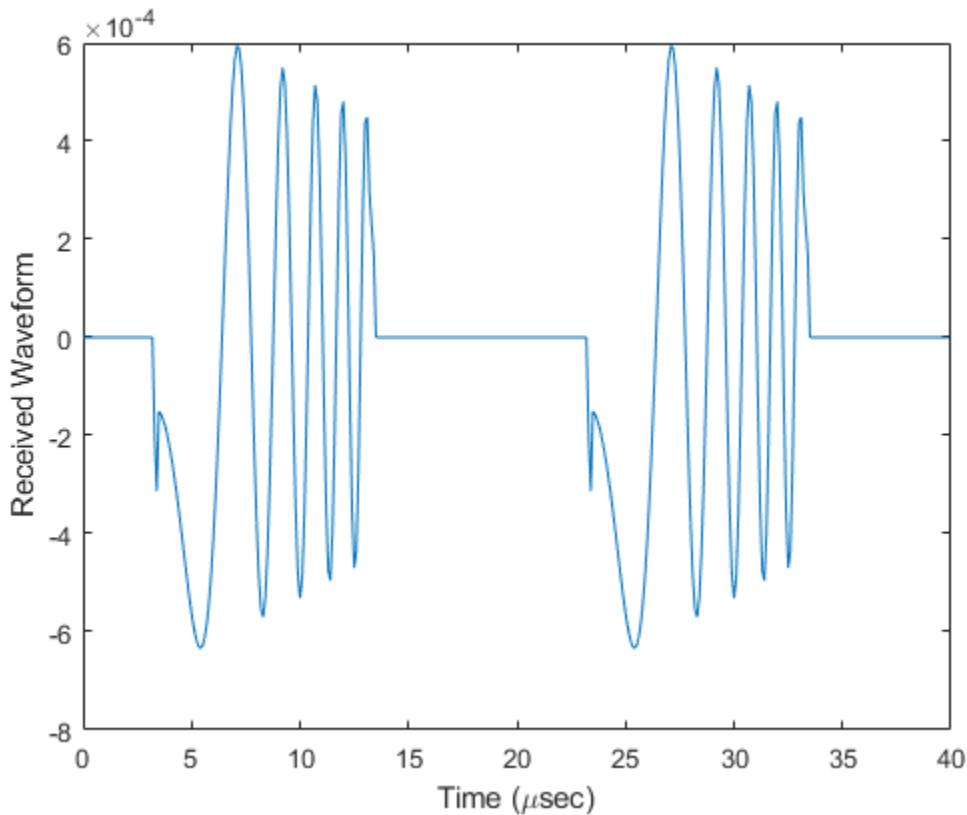
```
[~,angsRx] = rangeangle(posTx,posRx,laxRx,'two-ray');
```

Collect and combine received rays.

```
y = collector(prop_sig,angsRx,laxRx);
```

### Plot received waveform

```
plot((0:(n-1))/fs*1000000,real(y))
xlabel('Time (μsec)')
ylabel('Received Waveform')
```



### Two-Ray Propagation of LFM Waveform

Propagate a linear FM signal in a two-ray channel. The signal propagates from a transmitter located at (1000, 10, 10) meters in the global coordinate system to a receiver at (10000, 200, 30) meters. Assume that the transmitter and the receiver are stationary and that they both have cosine antenna patterns. Plot the received signal.

Set up the radar scenario. First, create the required System objects.

```

waveform = phased.LinearFMWaveform('SampleRate',1000000,...
    'OutputFormat','Pulses','NumPulses',2);
fs = waveform.SampleRate;
antenna = phased.CosineAntennaElement;
radiator = phased.Radiator('Sensor',antenna);
collector = phased.Collector('Sensor',antenna);
channel = twoRayChannel('SampleRate',fs,...
    'CombinedRaysOutput',false,'GroundReflectionCoefficient',0.95);

```

Set up the scene geometry. Specify transmitter and receiver positions and velocities. The transmitter and receiver are stationary.

```

posTx = [1000;10;10];
posRx = [10000;200;30];
velTx = [0;0;0];
velRx = [0;0;0];

```

Specify the transmitting and receiving radar antenna orientations with respect to the global coordinates. The transmitting antenna points along the +x direction and the receiving antenna points near but not directly in the -x direction.

```
laxTx = eye(3);
laxRx = rotx(5)*rotz(170);
```

Compute the transmission angles which are the angles that the two rays traveling toward the receiver leave the transmitter. The `phased.Radiator System` object™ uses these angles to apply separate antenna gains to the two signals. Because the antenna gains depend on path direction, you must transmit and receive the two rays separately.

```
[~,angTx] = rangeangle(posRx,posTx,laxTx,'two-ray');
```

Create and radiate signals from transmitter along the transmission directions.

```
wavfrm = waveform();
wavtrans = radiator(wavfrm,angTx);
```

Propagate signals to receiver via two-ray channel.

```
wavrcv = channel(wavtrans,posTx,posRx,velTx,velRx);
```

Collect signals at the receiver. Compute the angle at which the two rays traveling from the transmitter arrive at the receiver. The `phased.Collector System` object™ uses these angles to apply separate antenna gains to the two signals.

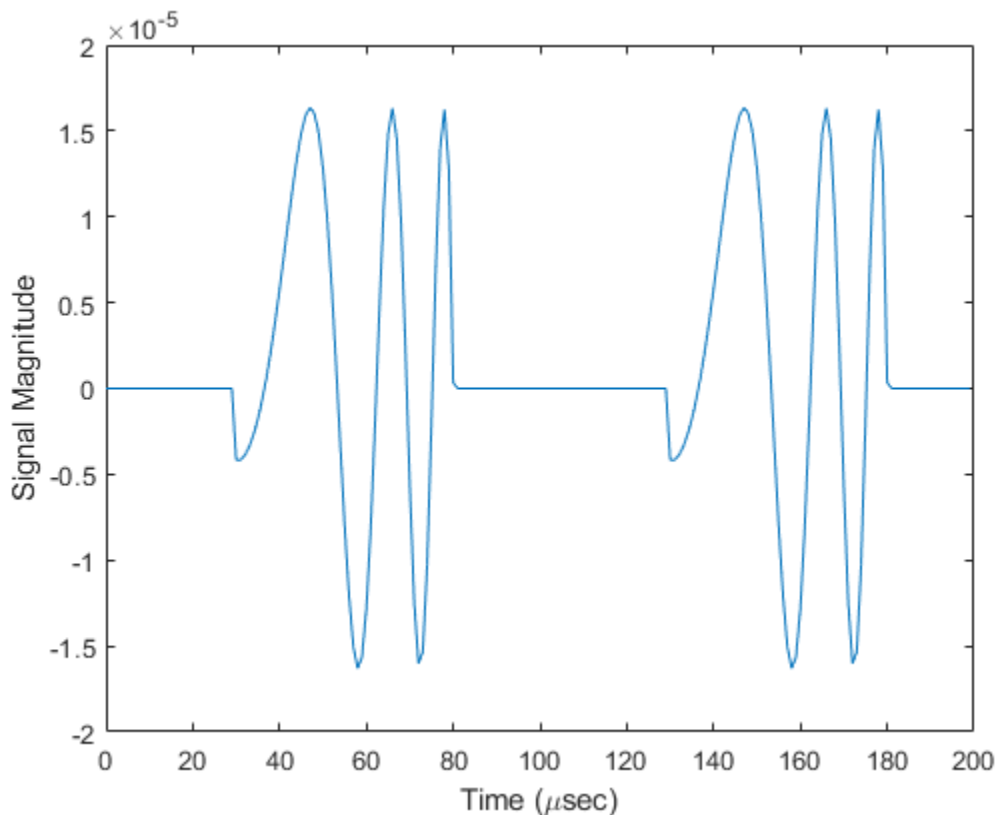
```
[~,angRcv] = rangeangle(posTx,posRx,laxRx,'two-ray');
```

Collect and combine the two received rays.

```
yR = collector(wavrcv,angRcv);
```

Plot the received signals.

```
dt = 1/fs;
n = size(yR,1);
plot((0:(n-1))*dt*1000000,real(yR))
xlabel('Time ({\mu}sec)')
ylabel('Signal Magnitude')
```



### Two-Ray Propagation of LFM Waveform with Atmospheric Losses

Propagate a 100 MHz linear FM signal into a two-ray channel. Assume there is signal loss caused by atmospheric gases and rain. The signal propagates from a transmitter located at  $(0, 0, 0)$  meters in the global coordinate system to a receiver at  $(10000, 200, 30)$  meters. Assume that the transmitter and the receiver are stationary and that they both have cosine antenna patterns. Plot the received signal. Set the dry air pressure to 102.5 Pa and the rain rate to 5 mm/hr.

#### Set Up Radar Scenario

```

waveform = phased.LinearFMWaveform('SampleRate',1e6,...
    'OutputFormat','Pulses','NumPulses',2);
antenna = phased.CosineAntennaElement;
radiator = phased.Radiator('Sensor',antenna);
collector = phased.Collector('Sensor',antenna);
channel = twoRayChannel('SampleRate',waveform.SampleRate,...
    'CombinedRaysOutput',false,'GroundReflectionCoefficient',0.95,...
    'SpecifyAtmosphere',true,'Temperature',20,...
    'DryAirPressure',102.5,'RainRate',5.0);

```

Set up the scene geometry giving the transmitter and receiver positions and velocities. The transmitter and receiver are stationary.

```

posTx = [0;0;0];
posRx = [10000;200;30];

```

```
velTx = [0;0;0];
velRx = [0;0;0];
```

Specify the transmitting and receiving radar antenna orientations with respect to the global coordinates. The transmitting antenna points along the +x-direction and the receiving antenna points close to the -x-direction.

```
laxTx = eye(3);
laxRx = rotx(5)*rotz(170);
```

Compute the transmission angles which are the angles that the two rays traveling toward the receiver leave the transmitter. The `phased.Radiator System` object™ uses these angles to apply separate antenna gains to the two signals. Because the antenna gains depend on path direction, you must transmit and receive the two rays separately.

```
[~,angTx] = rangeangle(posRx,posTx,laxTx,'two-ray');
```

### Create and Radiate Signals from Transmitter

Radiate the signals along the transmission directions.

```
wavfrm = waveform();
wavtrans = radiator(wavfrm,angTx);
```

Propagate signals to receiver via two-ray channel.

```
wavrcv = channel(wavtrans,posTx,posRx,velTx,velRx);
```

### Collect Signal at Receiver

Compute the angle at which the two rays traveling from the transmitter arrive at the receiver. The `phased.Collector System` object™ uses these angles to apply separate antenna gains to the two signals.

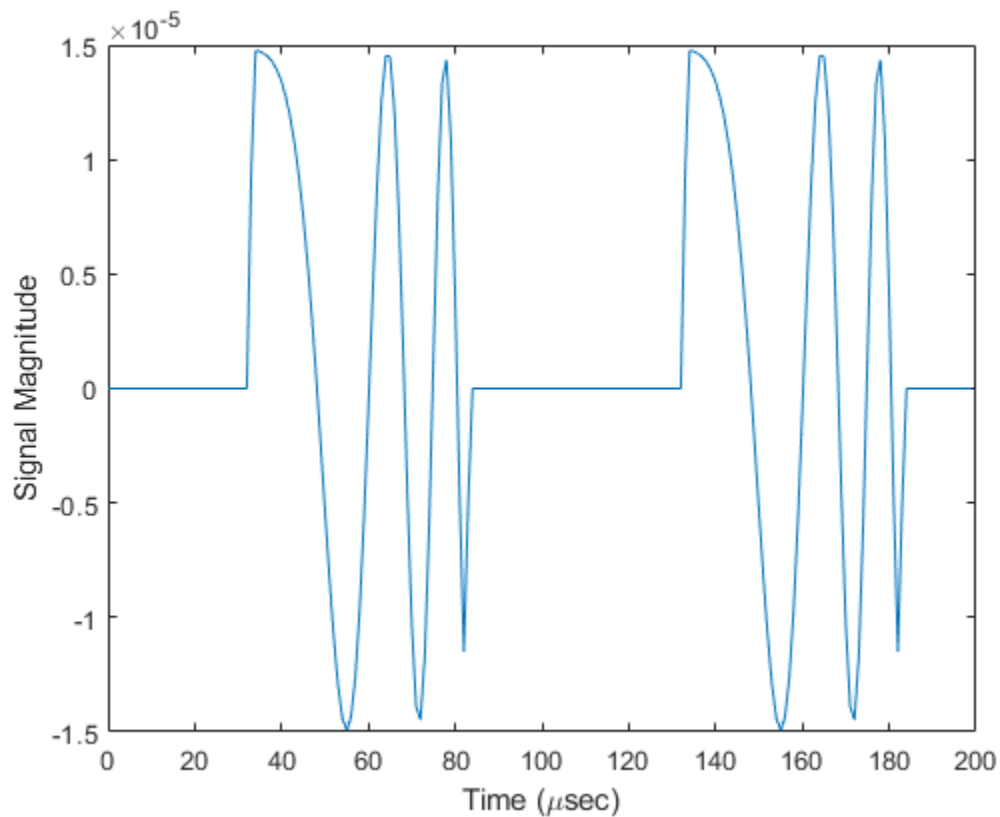
```
[~,angRcv] = rangeangle(posTx,posRx,laxRx,'two-ray');
```

Collect and combine the two received rays.

```
yR = collector(wavrcv,angRcv);
```

### Plot Received Signal

```
dt = 1/waveform.SampleRate;
n = size(yR,1);
plot((0:(n-1))*dt*1000000,real(yR))
xlabel('Time ({\mu}sec)')
ylabel('Signal Magnitude')
```



## References

- [1] Proakis, J. *Digital Communications*. New York: McGraw-Hill, 2001.
- [2] Skolnik, M. *Introduction to Radar Systems*, 3rd Ed. New York: McGraw-Hill
- [3] Saakian, A. *Radio Wave Propagation Fundamentals*. Norwood, MA: Artech House, 2011.
- [4] Balanis, C. *Advanced Engineering Electromagnetics*. New York: Wiley & Sons, 1989.
- [5] Rappaport, T. *Wireless Communications: Principles and Practice, 2nd Ed* New York: Prentice Hall, 2002.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### Introduced in R2021a

# widebandTwoRayChannel

Wideband two-ray propagation channel

## Description

The `widebandTwoRayChannel` models a wideband two-ray propagation channel. A two-ray propagation channel is the simplest type of multipath channel. You can use a two-ray channel to simulate propagation of signals in a homogeneous, isotropic medium with a single reflecting boundary. This type of medium has two propagation paths: a line-of-sight (direct) propagation path from one point to another and a ray path reflected from the boundary.

You can use this System object for short-range radar and mobile communications applications where the signals propagate along straight paths and the earth is assumed to be flat. You can also use this object for sonar and microphone applications. For acoustic applications, you can choose nonpolarized fields and adjust the propagation speed to be the speed of sound in air or water. You can use `widebandTwoRayChannel` to model propagation from several points simultaneously.

Although the System object works for all frequencies, the attenuation models for atmospheric gases and rain are valid for electromagnetic signals in the frequency range 1-1000 GHz only. The attenuation model for fog and clouds is valid for 10-1000 GHz. Outside these frequency ranges, the System object uses the nearest valid value.

The `widebandTwoRayChannel` System object applies range-dependent time delays to the signals, as well as gains or losses, phase shifts, and boundary reflection loss. When either the source or destination is moving, the System object applies Doppler shifts to the signals.

Signals at the channel output can be kept *separate* or be *combined*. If you keep the signals separate, both signals arrive at the destination separately and are not combined. If you choose to combine the signals, the two signals from the source propagate separately but are coherently summed at the destination into a single quantity. Choose this option when the difference between the sensor or array gains in the directions of the two paths is insignificant.

In contrast to the `phased.WidebandFreeSpace` and `phased.WidebandLOSChannel` System objects, this System object does not support two-way propagation.

To compute the propagation delay for specified source and receiver points:

- 1 Define and set up your two-ray channel. See “Construction” on page 4-313.
- 2 Call the `step` method to compute the propagated signal using the properties of the `widebandTwoRayChannel` System object.

---

**Note** Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

---

## Construction

`channel = widebandTwoRayChannel` creates a two-ray propagation channel System object, `channel`.

`channel = widebandTwoRayChannel(Name, Value)` creates a System object, `channel`, with each specified property `Name` set to the specified `Value`. You can specify additional name and value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

## Properties

### PropagationSpeed — Signal propagation speed

`physconst('LightSpeed')` (default) | positive scalar

Signal propagation speed, specified as a positive scalar. Units are in meters per second. The default propagation speed is the value returned by `physconst('LightSpeed')`. See `physconst` for more information.

Example: `3e8`

Data Types: `double`

### OperatingFrequency — Operating frequency

`300e6` (default) | positive scalar

Operating frequency, specified as a positive scalar. Units are in Hz.

Example: `1e9`

Data Types: `double`

### SpecifyAtmosphere — Enable atmospheric attenuation model

`false` (default) | `true`

Option to enable the atmospheric attenuation model, specified as a `false` or `true`. Set this property to `true` to add signal attenuation caused by atmospheric gases, rain, fog, or clouds. Set this property to `false` to ignore atmospheric effects in propagation.

Setting `SpecifyAtmosphere` to `true`, enables the `Temperature`, `DryAirPressure`, `WaterVapourDensity`, `LiquidWaterDensity`, and `RainRate` properties.

Data Types: `logical`

### Temperature — Ambient temperature

`15` (default) | real-valued scalar

Ambient temperature, specified as a real-valued scalar. Units are in degrees Celsius.

Example: `20.0`

### Dependencies

To enable this property, set `SpecifyAtmosphere` to `true`.

Data Types: `double`

### DryAirPressure — Atmospheric dry air pressure

`101.325e3` (default) | positive real-valued scalar



Atmospheric dry air pressure, specified as a positive real-valued scalar. Units are in pascals (Pa). The default value of this property corresponds to one standard atmosphere.

Example: 101.0e3

#### **Dependencies**

To enable this property, set `SpecifyAtmosphere` to `true`.

Data Types: `double`

#### **WaterVapourDensity — Atmospheric water vapor density**

7.5 (default) | positive real-valued scalar

Atmospheric water vapor density, specified as a positive real-valued scalar. Units are in  $\text{g/m}^3$ .

Example: 7.4

#### **Dependencies**

To enable this property, set `SpecifyAtmosphere` to `true`.

Data Types: `double`

#### **LiquidWaterDensity — Liquid water density**

0.0 (default) | nonnegative real-valued scalar

Liquid water density of fog or clouds, specified as a nonnegative real-valued scalar. Units are in  $\text{g/m}^3$ . Typical values for liquid water density are 0.05 for medium fog and 0.5 for thick fog.

Example: 0.1

#### **Dependencies**

To enable this property, set `SpecifyAtmosphere` to `true`.

Data Types: `double`

#### **RainRate — Rainfall rate**

0.0 (default) | nonnegative scalar

Rainfall rate, specified as a nonnegative scalar. Units are in mm/hr.

Example: 10.0

#### **Dependencies**

To enable this property, set `SpecifyAtmosphere` to `true`.

Data Types: `double`

#### **SampleRate — Sample rate of signal**

1e6 (default) | positive scalar

Sample rate of signal, specified as a positive scalar. Units are in Hz. The System object uses this quantity to calculate the propagation delay in units of samples.

Example: 1e6

Data Types: `double`

**NumSubbands — Number of processing subbands**

64 (default) | positive integer

Number of processing subbands, specified as a positive integer.

Example: 128

Data Types: double

**EnablePolarization — Enable polarized fields**

false (default) | true

Option to enable polarized fields, specified as `false` or `true`. Set this property to `true` to enable polarization. Set this property to `false` to ignore polarization.

Data Types: logical

**GroundReflectionCoefficient — Ground reflection coefficient**-1 (default) | complex-valued scalar | complex-valued 1-by- $N$  row vector

Ground reflection coefficient for the field at the reflection point, specified as a complex-valued scalar or a complex-valued 1-by- $N$  row vector. Each coefficient has an absolute value less than or equal to one. The quantity  $N$  is the number of two-ray channels. Units are dimensionless. Use this property to model nonpolarized signals. To model polarized signals, use the `GroundRelativePermittivity` property.

Example: -0.5

**Dependencies**

To enable this property, set `EnablePolarization` to `false`.

Data Types: double

Complex Number Support: Yes

**GroundRelativePermittivity — Ground relative permittivity**15 (default) | positive real-valued scalar | real-valued 1-by- $N$ row vector of positive values

Relative permittivity of the ground at the reflection point, specified as a positive real-valued scalar or a 1-by- $N$  real-valued row vector of positive values. The dimension  $N$  is the number of two-ray channels. Permittivity units are dimensionless. Relative permittivity is defined as the ratio of actual ground permittivity to the permittivity of free space. This property applies when you set the `EnablePolarization` property to `true`. Use this property to model polarized signals. To model nonpolarized signals, use the `GroundReflectionCoefficient` property.

Example: 5

**Dependencies**

To enable this property, set `EnablePolarization` to `true`.

Data Types: double

**CombinedRaysOutput — Option to combine two rays at output**

true (default) | false

Option to combine the two rays at channel output, specified as `true` or `false`. When this property is `true`, the object coherently adds the line-of-sight propagated signal and the reflected path signal

when forming the output signal. Use this mode when you do not need to include the directional gain of an antenna or array in your simulation.

Data Types: logical

### **MaximumDistanceSource — Source of maximum one-way propagation distance**

'Auto' (default) | 'Property'

Source of maximum one-way propagation distance, specified as 'Auto' or 'Property'. The maximum one-way propagation distance is used to allocate sufficient memory for signal delay computation. When you set this property to 'Auto', the System object automatically allocates memory. When you set this property to 'Property', you specify the maximum one-way propagation distance using the value of the MaximumDistance property.

Data Types: char

### **MaximumDistance — Maximum one-way propagation distance**

10000 (default) | positive real-valued scalar

Maximum one-way propagation distance, specified as a positive real-valued scalar. Units are in meters. Any signal that propagates more than the maximum one-way distance is ignored. The maximum distance must be greater than or equal to the largest position-to-position distance.

Example: 5000

#### **Dependencies**

To enable this property, set the MaximumDistanceSource property to 'Property'.

Data Types: double

### **MaximumNumInputSamplesSource — Source of maximum number of samples**

'Auto' (default) | 'Property'

The source of the maximum number of samples of the input signal, specified as 'Auto' or 'Property'. When you set this property to 'Auto', the propagation model automatically allocates enough memory to buffer the input signal. When you set this property to 'Property', you specify the maximum number of samples in the input signal using the MaximumNumInputSamples property. Any input signal longer than that value is truncated.

To use this object with variable-size signals in a MATLAB Function Block in Simulink, set the MaximumNumInputSamplesSource property to 'Property' and set a value for the MaximumNumInputSamples property.

Example: 'Property'

#### **Dependencies**

To enable this property, set MaximumDistanceSource to 'Property'.

Data Types: char

### **MaximumNumInputSamples — Maximum number of input signal samples**

100 (default) | positive integer

Maximum number of input signal samples, specified as a positive integer. The input signal is the first argument of the step method, after the System object itself. The size of the input signal is the number of rows in the input matrix. Any input signal longer than this number is truncated. To process signals completely, ensure that this property value is greater than any maximum input signal length.

The waveform-generating System objects determine the maximum signal size:

- For any waveform, if the waveform `OutputFormat` property is set to 'Samples', the maximum signal length is the value specified in the `NumSamples` property.
- For pulse waveforms, if the `OutputFormat` is set to 'Pulses', the signal length is the product of the smallest pulse repetition frequency, the number of pulses, and the sample rate.
- For continuous waveforms, if the `OutputFormat` is set to 'Sweeps', the signal length is the product of the sweep time, the number of sweeps, and the sample rate.

Example: 2048

### Dependencies

To enable this property, set `MaximumNumInputSamplesSource` to 'Property'.

Data Types: `double`

### Methods

`reset` Reset states of System object

`step` Propagate wideband signal from point to point using two-ray channel model

Common to All System Objects	
<code>release</code>	Allow System object property value changes

### Examples

#### Scalar Wideband Signal Propagating in Two-Ray Channel

This example illustrates the two-ray propagation of a wideband signal, showing how the signals from the line-of-sight path and reflected path arrive at the receiver at different times.

**Note:** You can replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

#### Create and Plot Transmitted Waveform

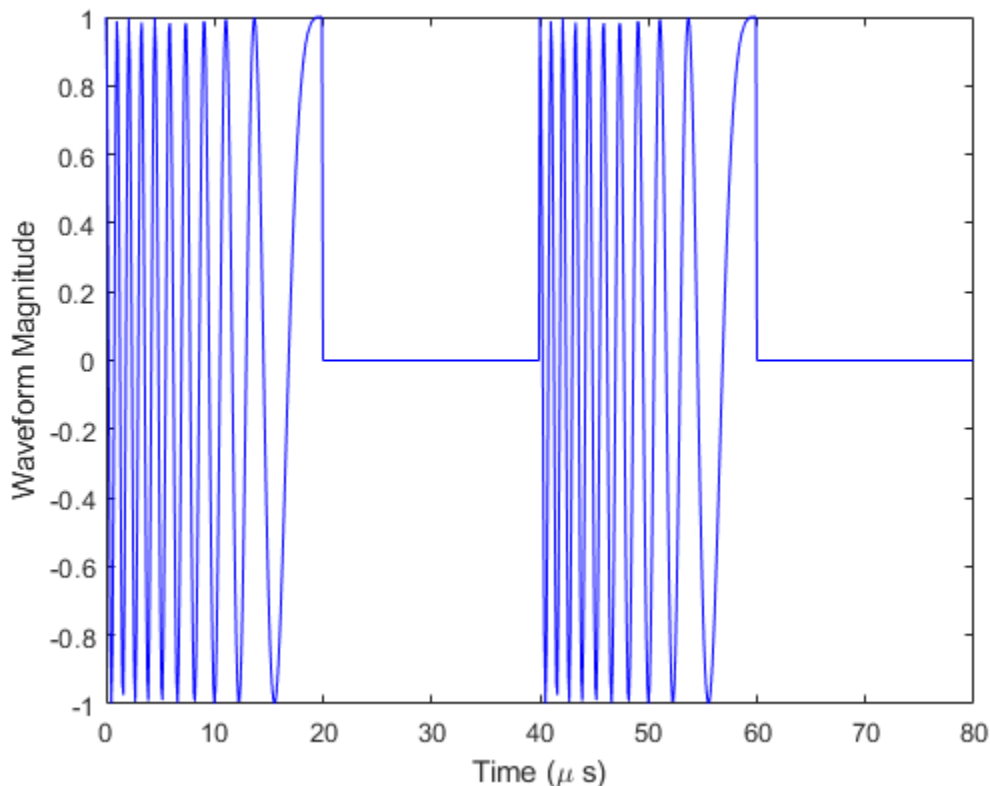
Create a nonpolarized electromagnetic field consisting of two linear FM waveform pulses at a carrier frequency of 100 MHz. Assume the pulse width is 20  $\mu$ s and the sampling rate is 10 MHz. The bandwidth of the pulse is 1 MHz. Assume a 50% duty cycle so that the pulse width is one-half the pulse repetition interval. Create a two-pulse wave train. Set the `GroundReflectionCoefficient` to -0.9 to model strong ground reflectivity. Propagate the field from a stationary source to a stationary receiver. The vertical separation of the source and receiver is approximately 10 km.

```
c = physconst('LightSpeed');
fs = 10e6;
pw = 20e-6;
pri = 2*pw;
PRF = 1/pri;
fc = 100e6;
lambda = c/fc;
bw = 1e6;
```

```

waveform = phased.LinearFMWaveform('SampleRate',fs,'PulseWidth',pw,...
    'PRF',PRF,'OutputFormat','Pulses','NumPulses',2,'SweepBandwidth',bw,...
    'SweepDirection','Down','Envelope','Rectangular','SweepInterval',...
    'Positive');
wav = waveform();
n = size(wav,1);
plot([0:(n-1)]/fs*1e6,real(wav),'b')
xlabel('Time (\mu s)')
ylabel('Waveform Magnitude')

```



### Specify the Location of Source and Receiver

Place the source and receiver about 1 km apart horizontally and approximately 5 km apart vertically.

```

pos1 = [0;0;100];
pos2 = [1e3;0;5.0e3];
vel1 = [0;0;0];
vel2 = [0;0;0];

```

### Create a Wideband Two-Ray Channel System Object

Create a two-ray propagation channel System object™ and propagate the signal along both the line-of-sight and reflected ray paths. The same signal is propagated along both paths.

```

channel = widebandTwoRayChannel('SampleRate',fs,...
    'GroundReflectionCoefficient',-0.9,'OperatingFrequency',fc,...
    'CombinedRaysOutput',false);
prop_signal = channel([wav,wav],pos1,pos2,vel1,vel2);

```

```
[rng2,angs] = rangeangle(pos2,pos1, 'two-ray');
```

Calculate time delays in  $\mu\text{s}$ .

```
tm = rng2/c*1e6;  
disp(tm)
```

```
16.6815 17.3357
```

Display the calculated propagation paths azimuth and elevation angles in degrees.

```
disp(angs)
```

```
0 0  
78.4654 -78.9063
```

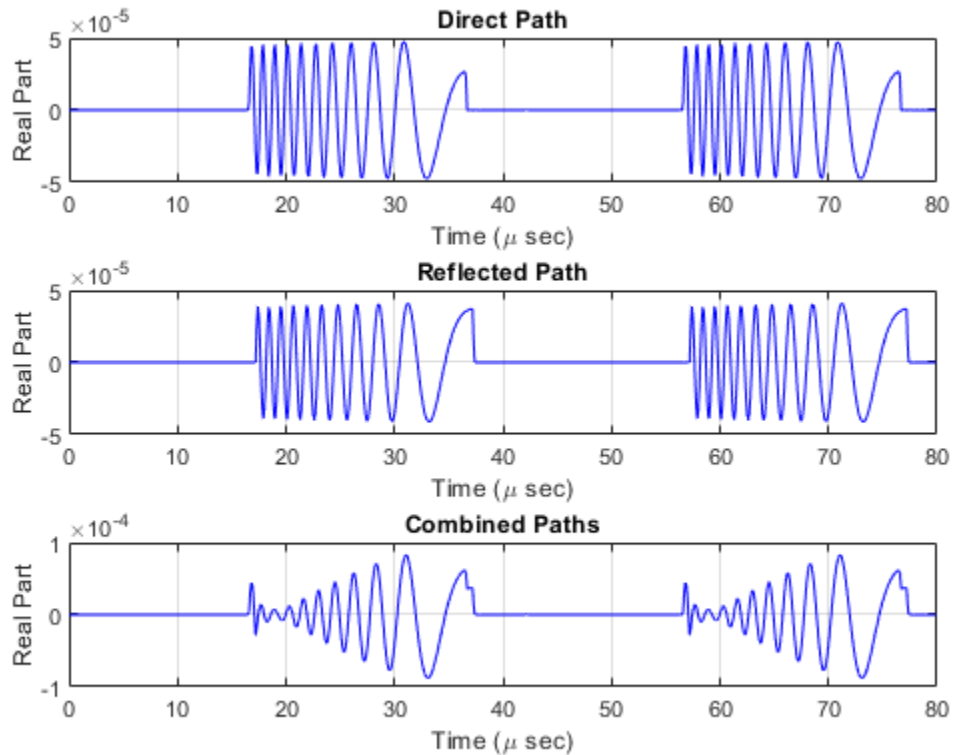
### Plot the Propagated Signals

- 1 Plot the real part of the signal propagated along the line-of-sight path.
- 2 Plot the real part of the signal propagated along the reflected path.
- 3 Plot the real part of the coherent sum of the two signals.

```
n = size(prop_signal,1);  
delay = [0:(n-1)]/fs*1e6;  
subplot(3,1,1)  
plot(delay,real([prop_signal(:,1)]),'b')  
grid  
xlabel('Time (\mu sec)')  
ylabel('Real Part')  
title('Direct Path')
```

```
subplot(3,1,2)  
plot(delay,real([prop_signal(:,2)]),'b')  
grid  
xlabel('Time (\mu sec)')  
ylabel('Real Part')  
title('Reflected Path')
```

```
subplot(3,1,3)  
plot(delay,real([prop_signal(:,1) + prop_signal(:,2)]),'b')  
grid  
xlabel('Time (\mu sec)')  
ylabel('Real Part')  
title('Combined Paths')
```



The delay of the reflected path signal agrees with the predicted delay. The magnitude of the coherently combined signal is less than either of the propagated signals. This result indicates that the two signals contain some interference.

### Compare Wideband Two-Ray Channel Propagation to Free Space

Compute the result of propagating a wideband LFM signal in a two-ray environment from a radar 10 meters above the origin  $(0,0,10)$  to a target at  $(3000,2000,2000)$  meters. Assume that the radar and target are stationary and that the transmitting antenna is isotropic. Combine the signal from the two paths and compare the signal to a signal propagating in free space. The system operates at 300 MHz. Set the `CombinedRaysOutput` property to `true` to combine the direct path and reflected path signals when forming the output signal.

**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)`.

Create a linear FM waveform.

```
fop = 300.0e6;
fs = 1.0e6;
waveform = phased.LinearFMWaveform();
x = waveform();
```

Specify the target position and velocity.

```

posTx = [0; 0; 10];
posTgt = [3000; 2000; 2000];
velTx = [0;0;0];
velTgt = [0;0;0];

```

Model the free space propagation.

```

fschannel = phased.WidebandFreeSpace('SampleRate', waveform.SampleRate);
y_fs = fschannel(x, posTx, posTgt, velTx, velTgt);

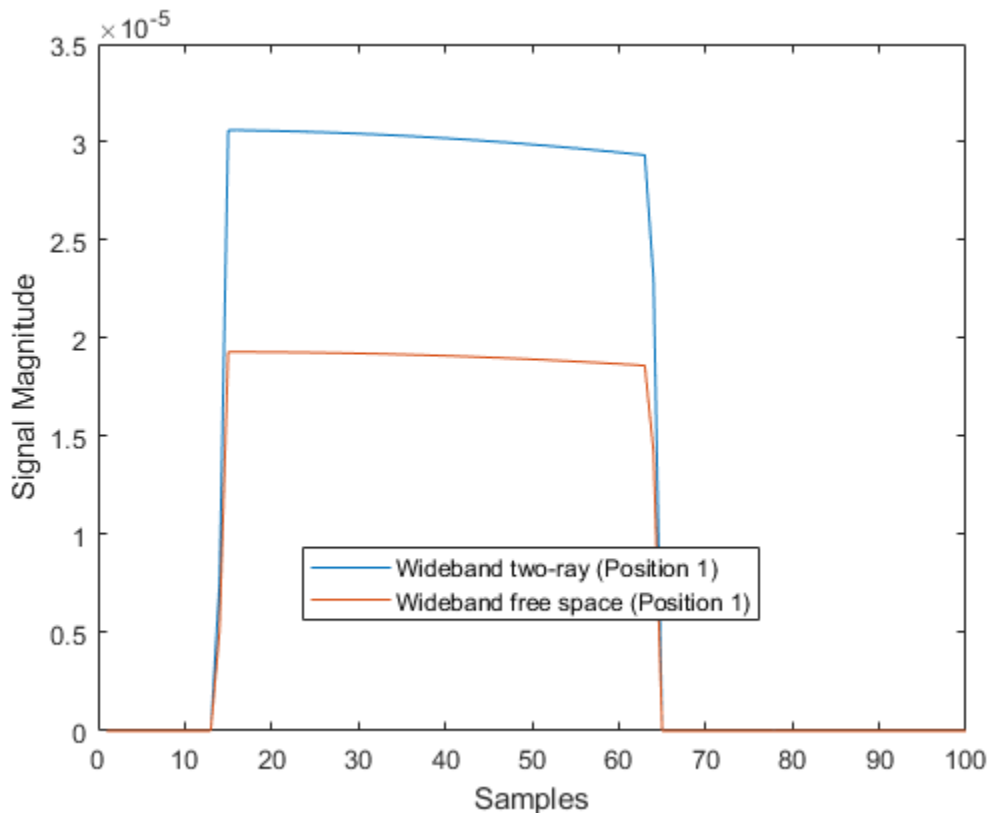
```

Model two-ray propagation from the position of the radar to the target.

```

tworaychannel = widebandTwoRayChannel('SampleRate', waveform.SampleRate, ...
    'CombinedRaysOutput', true);
y_tworay = tworaychannel(x, posTx, posTgt, velTx, velTgt);
plot(abs([y_tworay y_fs]))
legend('Wideband two-ray (Position 1)', 'Wideband free space (Position 1)', ...
    'Location', 'best')
xlabel('Samples')
ylabel('Signal Magnitude')
hold on

```



Move the radar by 10 meters horizontally to a second position.

```

posTx = posTx + [10;0;0];
y_fs = fschannel(x, posTx, posTgt, velTx, velTgt);
y_tworay = tworaychannel(x, posTx, posTgt, velTx, velTgt);
plot(abs([y_tworay y_fs]))

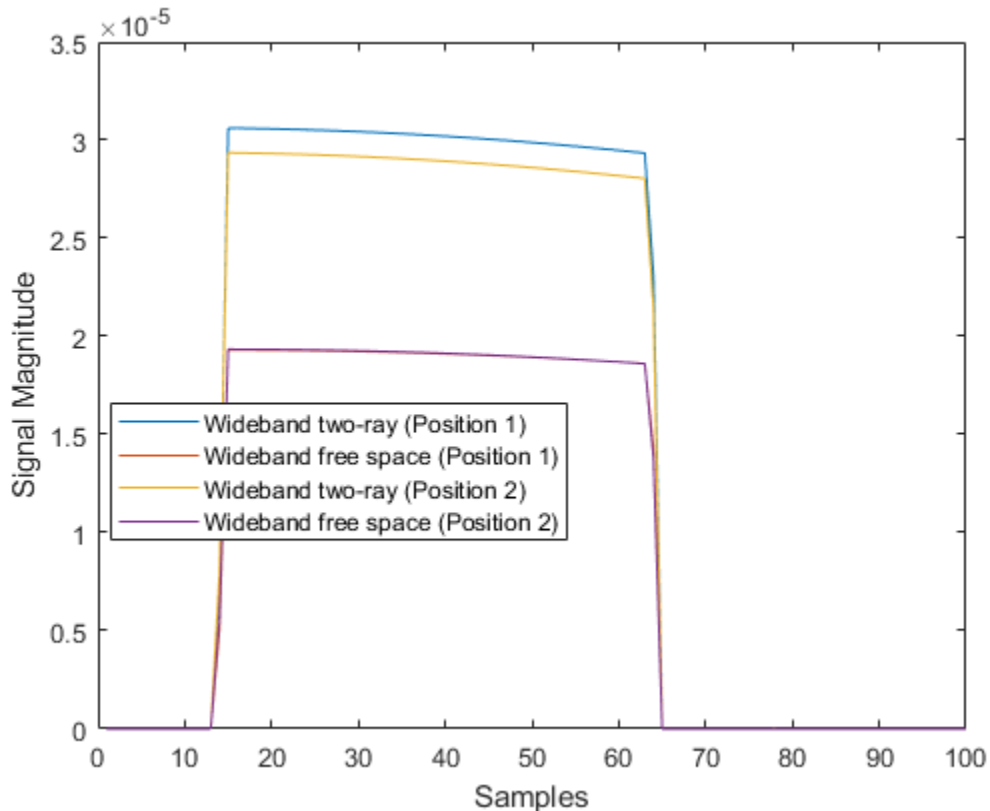
```



```

legend('Wideband two-ray (Position 1)', 'Wideband free space (Position 1)', ...
       'Wideband two-ray (Position 2)', 'Wideband free space (Position 2)', ...
       'Location', 'best')
hold off

```



The free-space propagation losses are the same for both the first and second positions of the radar. The two-ray losses are different due to the interference effect of the two-ray paths.

### Wideband Polarized Field Propagation in Two-Ray Channel

Create a polarized electromagnetic field consisting of linear FM waveform pulses. Propagate the field from a stationary source with a crossed-dipole antenna element to a stationary receiver approximately 10 km away. The transmitting antenna is 100 m above the ground. The receiving antenna is 150 m above the ground. The receiving antenna is also a crossed-dipole. Plot the received signal.

**Note:** You can replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject, x)`.

### Set Radar Waveform Parameters

Assume the pulse width is  $10\mu\text{s}$  and the sampling rate is 10 MHz. The bandwidth of the pulse is 1 MHz. Assume a 50% duty cycle in which the pulse width is one-half the pulse repetition interval. Create a two-pulse wave train. Assume a carrier frequency of 100 MHz.

```

c = physconst('LightSpeed');
fs = 20e6;
pw = 10e-6;
pri = 2*pw;
PRF = 1/pri;
fc = 100e6;
bw = 1e6;
lambda = c/fc;

```

### Set Up Required System Objects

Use a GroundRelativePermittivity of 10.

```

waveform = phased.LinearFMWaveform('SampleRate',fs,'PulseWidth',pw,...
    'PRF',PRF,'OutputFormat','Pulses','NumPulses',2,'SweepBandwidth',bw,...
    'SweepDirection','Down','Envelope','Rectangular','SweepInterval',...
    'Positive');
antenna = phased.CrossedDipoleAntennaElement(...
    'FrequencyRange',[50,200]*1e6);
radiator = phased.Radiator('Sensor',antenna,'OperatingFrequency',fc,...
    'Polarization','Combined');
channel = phased.WidebandTwoRayChannel('SampleRate',fs,...
    'OperatingFrequency',fc,'CombinedRaysOutput',false,...
    'EnablePolarization',true,'GroundRelativePermittivity',10);
collector = phased.Collector('Sensor',antenna,'OperatingFrequency',fc,...
    'Polarization','Combined');

```

### Set Up Scene Geometry

Specify transmitter and receiver positions, velocities, and orientations. Place the source and receiver approximately 1000 m apart horizontally and approximately 50 m apart vertically.

```

posTx = [0;100;100];
posRx = [1000;0;150];
velTx = [0;0;0];
velRx = [0;0;0];
laxRx = rotz(180);
laxTx = rotx(1)*eye(3);

```

### Create and Radiate Signals from Transmitter

Compute the transmission angles for the two rays traveling toward the receiver. These angles are defined with respect to the transmitter local coordinate system. The `phased.Radiator System` object(TM) uses these angles to apply separate antenna gains to the two signals.

```

[rng,angsTx] = rangeangle(posRx,posTx,laxTx,'two-ray');
wav = waveform();

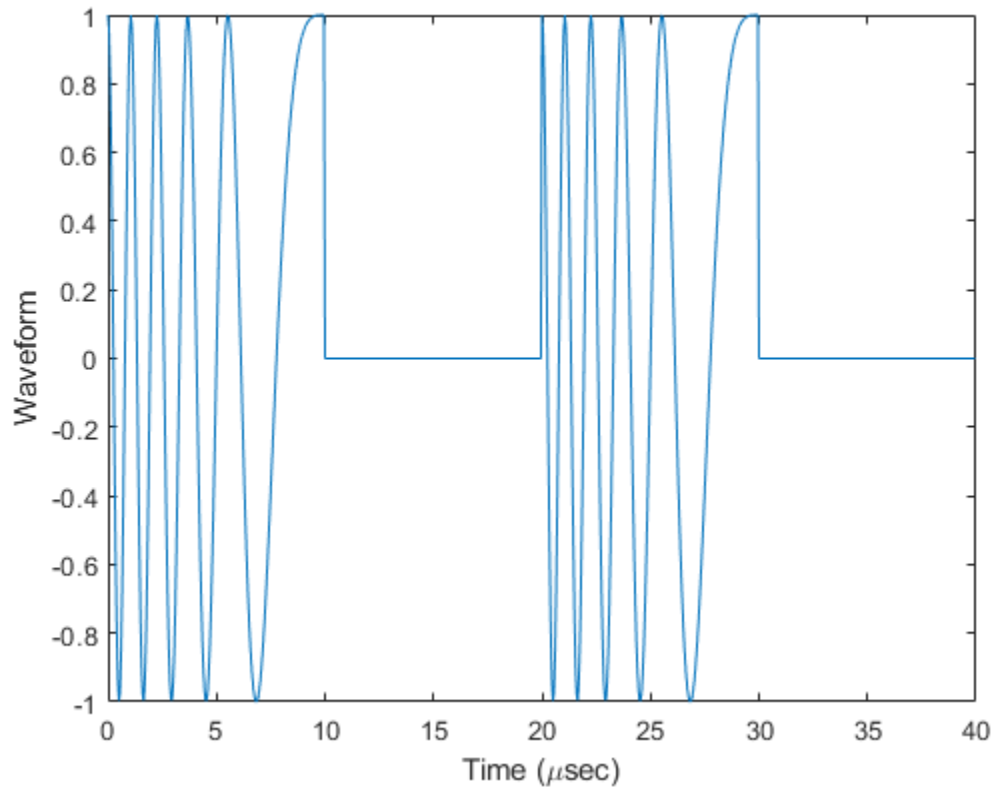
```

Plot the transmitted waveform.

```

n = size(wav,1);
plot([0:(n-1)]/fs*1000000,real(wav))
xlabel('Time ({\mu}sec)')
ylabel('Waveform')

```



```
sig = radiator(wav,angsTx,laxTx);
```

Propagate the signals to the receiver via a two-ray channel.

```
prop_sig = channel(sig,posTx,posRx,velTx,velRx);
```

### Receive Propagated Signal

Compute the reception angles for the two rays arriving at the receiver. These angles are defined with respect to the receiver local coordinate system. The phased.Collector System object(TM) uses these angles to apply separate antenna gains to the two signals.

```
[rng1,angsRx] = rangeangle(posTx,posRx,laxRx,'two-ray');
delays = rng1/c*1e6
```

```
delays = 1x2
```

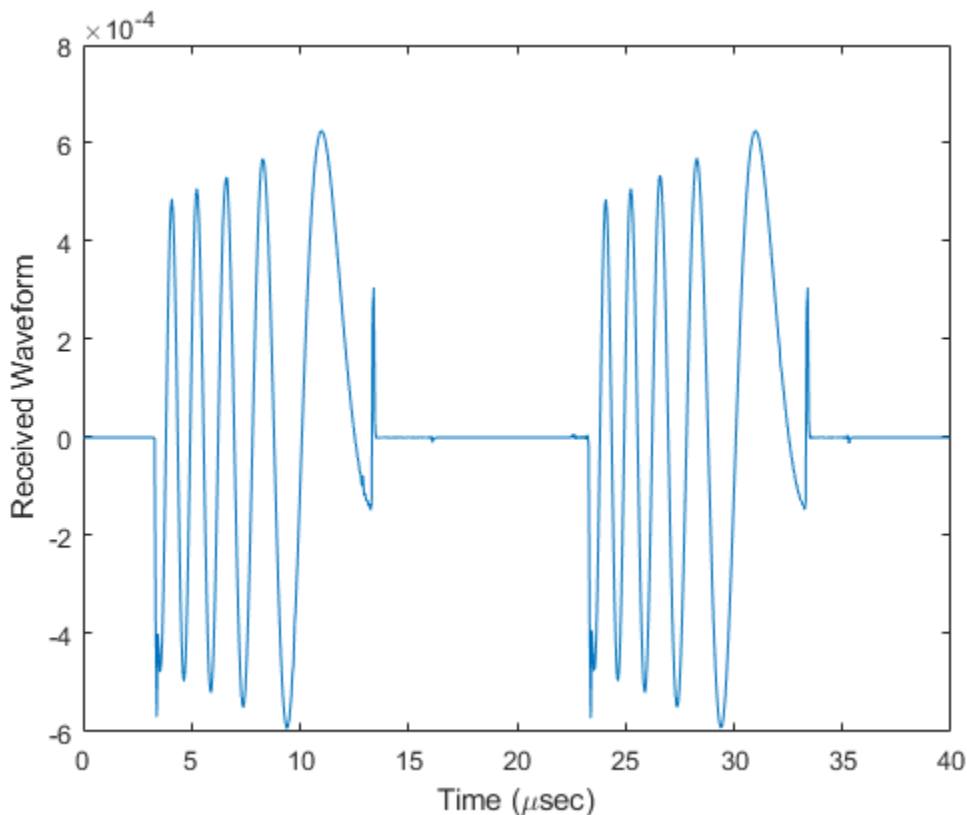
```
3.3564    3.4544
```

Collect and combine the received rays.

```
y = collector(prop_sig,angsRx,laxRx);
```

Plot the received waveform.

```
plot([0:(n-1)]/fs*1000000,real(y))
xlabel('Time (μsec)')
ylabel('Received Waveform')
```



### Two-Ray Propagation of Wideband LFM Waveform with Atmospheric Losses

Propagate a wideband linear FM signal in a two-ray channel. The signal bandwidth is 15% of the carrier frequency. Assume there is signal loss caused by atmospheric gases and rain. The signal propagates from a transmitter located at  $(0, 0, 0)$  meters in the global coordinate system to a receiver at  $(10000, 200, 30)$  meters. Assume that the transmitter and the receiver are stationary and that they both have cosine antenna patterns. Plot the received signal. Set the dry air pressure to 102.0 Pa and the rain rate to 5 mm/hr.

#### Set Radar Waveform Parameters

```
c = physconst('LightSpeed');
fs = 40e6;
pw = 10e-6;
pri = 2.5*pw;
PRF = 1/pri;
fc = 100e6;
bw = 15e6;
lambda = c/fc;
```

#### Set Up Radar Scenario

Create the required System objects.

```
waveform = phased.LinearFMWaveform('SampleRate',fs,'PulseWidth',pw,...
    'PRF',PRF,'OutputFormat','Pulses','NumPulses',2,'SweepBandwidth',bw,...
```

```

        'SweepDirection', 'Down', 'Envelope', 'Rectangular', 'SweepInterval', ...
        'Positive');
antenna = phased.CosineAntennaElement;
radiator = phased.Radiator('Sensor', antenna);
collector = phased.Collector('Sensor', antenna);
channel = widebandTwoRayChannel('SampleRate', waveform.SampleRate, ...
    'CombinedRaysOutput', false, 'GroundReflectionCoefficient', 0.95, ...
    'SpecifyAtmosphere', true, 'Temperature', 20, ...
    'DryAirPressure', 102.5, 'RainRate', 5.0);

```

Set up the scene geometry. Specify transmitter and receiver positions and velocities. The transmitter and receiver are stationary.

```

posTx = [0;0;0];
posRx = [10000;200;30];
velTx = [0;0;0];
velRx = [0;0;0];

```

Specify the transmitting and receiving radar antenna orientations with respect to the global coordinates. The transmitting antenna points along the positive x-direction and the receiving antenna points close to the negative x-direction.

```

laxTx = eye(3);
laxRx = rotx(5)*rotz(170);

```

Compute the transmission angles which are the angles at which the two rays traveling toward the receiver leave the transmitter. The `phased.Radiator System` object™ uses these angles to apply separate antenna gains to the two signals. Because the antenna gains depend on path direction, you must transmit and receive the two rays separately.

```

[~,angTx] = rangeangle(posRx,posTx,laxTx,'two-ray');

```

### Create and Radiate Signals from Transmitter

Radiate the signals along the transmission directions.

```

wavfrm = waveform();
wavtrans = radiator(wavfrm,angTx);

```

Propagate the signals to the receiver via a two-ray channel.

```

wavrcv = channel(wavtrans,posTx,posRx,velTx,velRx);

```

### Collect Signal at Receiver

Compute the angle at which the two rays traveling from the transmitter arrive at the receiver. The `phased.Collector System` object™ uses these angles to apply separate antenna gains to the two signals.

```

[~,angRcv] = rangeangle(posTx,posRx,laxRx,'two-ray');

```

Collect and combine the two received rays.

```

yR = collector(wavrcv,angRcv);

```

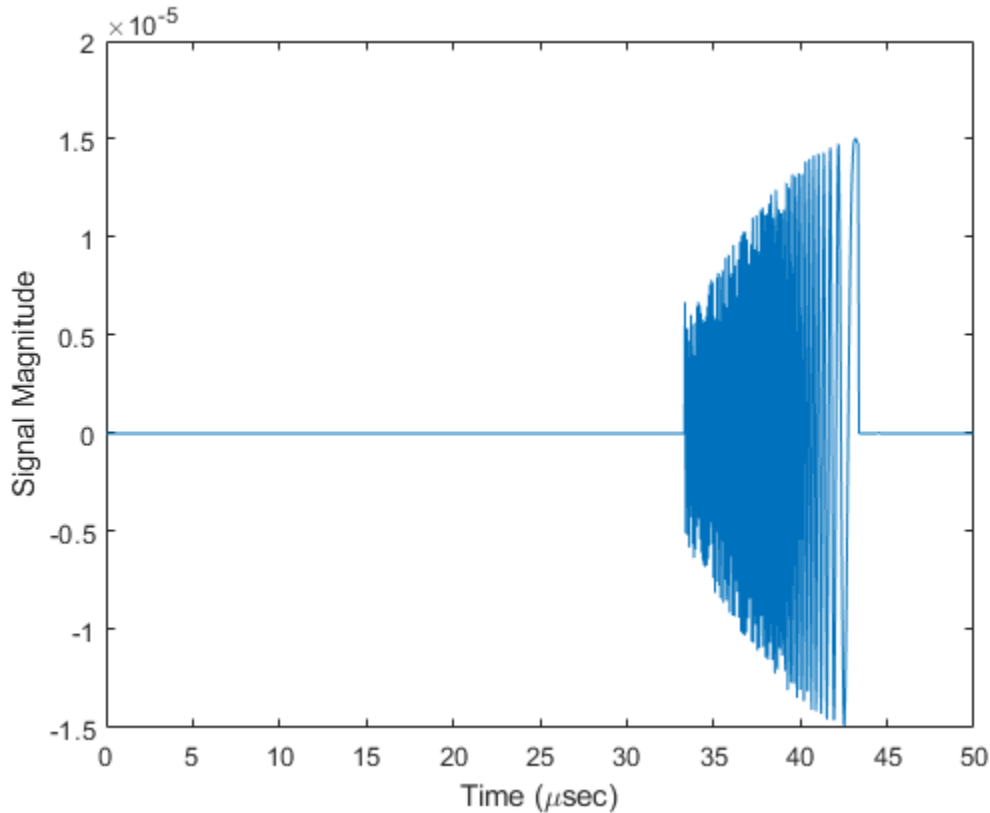
### Plot Received Signal

```

dt = 1/waveform.SampleRate;
n = size(yR,1);

```

```
plot([0:(n-1)]*dt*1e6, real(yR))
xlabel('Time ( $\mu$ sec)')
ylabel('Signal Magnitude')
```



## More About

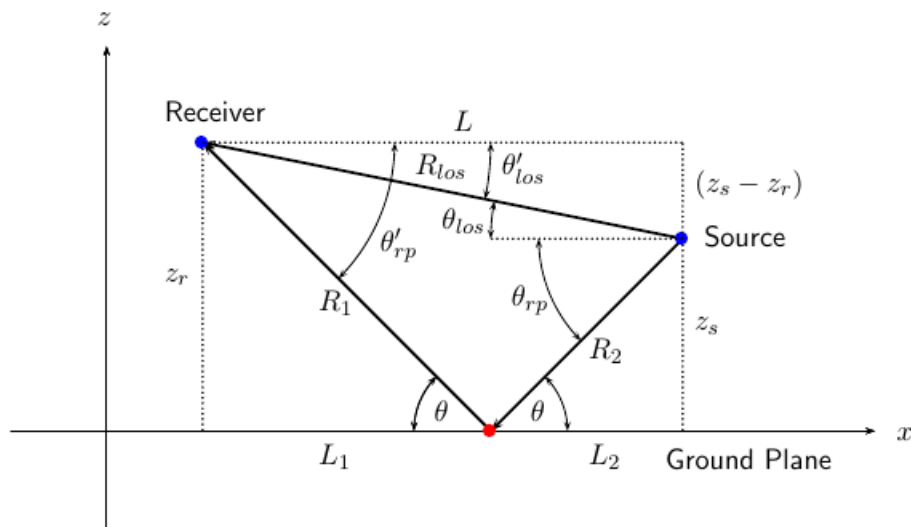
### Two-Ray Propagation Paths

A two-ray propagation channel is the next step up in complexity from a free-space channel and is the simplest case of a multipath propagation environment. The free-space channel models a straight-line *line-of-sight* path from point 1 to point 2. In a two-ray channel, the medium is specified as a homogeneous, isotropic medium with a reflecting planar boundary. The boundary is always set at  $z = 0$ . There are at most two rays propagating from point 1 to point 2. The first ray path propagates along the same line-of-sight path as in the free-space channel. The line-of-sight path is often called the *direct path*. The second ray reflects off the boundary before propagating to point 2. According to the Law of Reflection, the angle of reflection equals the angle of incidence. In short-range simulations such as cellular communications systems and automotive radars, you can assume that the reflecting surface, the ground or ocean surface, is flat.

The `twoRayChannel` and `widebandTwoRayChannel` System objects model propagation time delay, phase shift, Doppler shift, and loss effects for both paths. For the reflected path, loss effects include reflection loss at the boundary.

The figure illustrates two propagation paths. From the source position,  $s_s$ , and the receiver position,  $s_r$ , you can compute the arrival angles of both paths,  $\theta_{los}$  and  $\theta_{rp}$ . The arrival angles are the elevation

and azimuth angles of the arriving radiation with respect to a local coordinate system. In this case, the local coordinate system coincides with the global coordinate system. You can also compute the transmitting angles,  $\theta_{los}$  and  $\theta_{rp}$ . In the global coordinates, the angle of reflection at the boundary is the same as the angles  $\theta_{rp}$  and  $\theta'_{rp}$ . The reflection angle is important to know when you use angle-dependent reflection-loss data. You can determine the reflection angle by using the `rangeangle` function and setting the reference axes to the global coordinate system. The total path length for the line-of-sight path is shown in the figure by  $R_{los}$  which is equal to the geometric distance between source and receiver. The total path length for the reflected path is  $R_{rp} = R_1 + R_2$ . The quantity  $L$  is the ground range between source and receiver.



You can easily derive exact formulas for path lengths and angles in terms of the ground range and object heights in the global coordinate system.

$$\vec{R} = \vec{x}_s - \vec{x}_r$$

$$R_{los} = |\vec{R}| = \sqrt{(z_r - z_s)^2 + L^2}$$

$$R_1 = \frac{z_r}{z_r + z_s} \sqrt{(z_r + z_s)^2 + L^2}$$

$$R_2 = \frac{z_s}{z_s + z_r} \sqrt{(z_r + z_s)^2 + L^2}$$

$$R_{rp} = R_1 + R_2 = \sqrt{(z_r + z_s)^2 + L^2}$$

$$\tan\theta_{los} = \frac{(z_s - z_r)}{L}$$

$$\tan\theta_{rp} = -\frac{(z_s + z_r)}{L}$$

$$\theta'_{los} = -\theta_{los}$$

$$\theta'_{rp} = \theta_{rp}$$

### Two-Ray Attenuation

Attenuation or path loss in the two-ray channel is the product of five components,  $L = L_{tworay} L_G L_g L_c L_r$ , where

- $L_{tworay}$  is the two-ray geometric path attenuation
- $L_G$  is the ground reflection attenuation
- $L_g$  is the atmospheric path attenuation
- $L_c$  is the fog and cloud path attenuation
- $L_r$  is the rain path attenuation

Each component is in magnitude units, not in dB.

### Ground Reflection and Propagation Loss

Losses occurs when a signal is reflected from a boundary. You can obtain a simple model of ground reflection loss by representing the electromagnetic field as a scalar field. This approach also works for acoustic and sonar systems. Let  $E$  be a scalar free-space electromagnetic field having amplitude  $E_0$  at a reference distance  $R_0$  from a transmitter (for example, one meter). The propagating free-space field at distance  $R_{los}$  from the transmitter is

$$E_{los} = E_0 \left( \frac{R_0}{R_{los}} \right) e^{i\omega(t - R_{los}/c)}$$

for the line-of-sight path. You can express the ground-reflected  $E$ -field as

$$E_{rp} = L_G E_0 \left( \frac{R_0}{R_{rp}} \right) e^{i\omega(t - R_{rp}/c)}$$

where  $R_{rp}$  is the reflected path distance. The quantity  $L_G$  represents the loss due to reflection at the ground plane. To specify  $L_G$ , use the `GroundReflectionCoefficient` property. In general,  $L_G$  depends on the incidence angle of the field. If you have empirical information about the angular dependence of  $L_G$ , you can use `rangeangle` to compute the incidence angle of the reflected path. The total field at the destination is the sum of the line-of-sight and reflected-path fields.



For electromagnetic waves, a more complicated but more realistic model uses a vector representation of the polarized field. You can decompose the incident electric field into two components. One component,  $E_p$ , is parallel to the plane of incidence. The other component,  $E_s$ , is perpendicular to the plane of incidence. The ground reflection coefficients for these components differ and can be written in terms of the ground permittivity and incidence angle.

$$G_p = \frac{Z_1 \cos \theta_1 - Z_2 \cos \theta_2}{Z_1 \cos \theta_1 + Z_2 \cos \theta_2} = \frac{\cos \theta_1 - \frac{Z_2}{Z_1} \cos \theta_2}{\cos \theta_1 + \frac{Z_2}{Z_1} \cos \theta_2}$$

$$G_s = \frac{Z_2 \cos \theta_1 - Z_1 \cos \theta_2}{Z_2 \cos \theta_1 + Z_1 \cos \theta_2} = \frac{\cos \theta_2 - \frac{Z_2}{Z_1} \cos \theta_1}{\cos \theta_2 + \frac{Z_2}{Z_1} \cos \theta_1}$$

$$Z_1 = \sqrt{\frac{\mu_1}{\epsilon_1}}$$

$$Z_2 = \sqrt{\frac{\mu_2}{\epsilon_2}}$$

where  $Z$  is the impedance of the medium. Because the magnetic permeability of the ground is almost identical to that of air or free space, the ratio of impedances depends primarily on the ratio of electric permittivities

$$G_p = \frac{\sqrt{\rho} \cos \theta_1 - \cos \theta_2}{\sqrt{\rho} \cos \theta_1 + \cos \theta_2}$$

$$G_s = \frac{\sqrt{\rho} \cos \theta_2 - \cos \theta_1}{\sqrt{\rho} \cos \theta_2 + \cos \theta_1}$$

where the quantity  $\rho = \epsilon_2/\epsilon_1$  is the *ground relative permittivity* set by the `GroundRelativePermittivity` property. The angle  $\theta_1$  is the incidence angle and the angle  $\theta_2$  is the refraction angle at the boundary. You can determine  $\theta_2$  using Snell's law of refraction.

After reflection, the full field is reconstructed from the parallel and perpendicular components. The total ground plane attenuation,  $L_G$ , is a combination of  $G_s$  and  $G_p$ .

When the origin and destination are stationary relative to each other, you can write the output `Y` of `step` as  $Y(t) = F(t-\tau)/L$ . The quantity  $\tau$  is the signal delay and  $L$  is the free-space path loss. The delay  $\tau$  is given by  $R/c$ .  $R$  is either the line-of-sight propagation path distance or the reflected path distance, and  $c$  is the propagation speed. The path loss

where  $\lambda$  is the signal wavelength.

### Atmospheric Gas Attenuation Model

This model calculates the attenuation of signals that propagate through atmospheric gases.

Electromagnetic signals attenuate when they propagate through the atmosphere. This effect is due primarily to the absorption resonance lines of oxygen and water vapor, with smaller contributions coming from nitrogen gas. The model also includes a continuous absorption spectrum below 10 GHz. The ITU model *Recommendation ITU-R P.676-10: Attenuation by atmospheric gases* is used. The model computes the specific attenuation (attenuation per kilometer) as a function of temperature,

pressure, water vapor density, and signal frequency. The atmospheric gas model is valid for frequencies from 1-1000 GHz and applies to polarized and nonpolarized fields.

The formula for specific attenuation at each frequency is

$$\gamma = \gamma_o(f) + \gamma_w(f) = 0.1820fN''(f).$$

The quantity  $N''()$  is the imaginary part of the complex atmospheric refractivity and consists of a spectral line component and a continuous component:

$$N''(f) = \sum_i S_i F_i + N''_D(f)$$

The spectral component consists of a sum of discrete spectrum terms composed of a localized frequency bandwidth function,  $F(f)_i$ , multiplied by a spectral line strength,  $S_i$ . For atmospheric oxygen, each spectral line strength is

$$S_i = a_1 \times 10^{-7} \left( \frac{300}{T} \right)^3 \exp \left[ a_2 \left( 1 - \left( \frac{300}{T} \right) \right) \right] P.$$

For atmospheric water vapor, each spectral line strength is

$$S_i = b_1 \times 10^{-1} \left( \frac{300}{T} \right)^{3.5} \exp \left[ b_2 \left( 1 - \left( \frac{300}{T} \right) \right) \right] W.$$

$P$  is the dry air pressure,  $W$  is the water vapor partial pressure, and  $T$  is the ambient temperature. Pressure units are in hectoPascals (hPa) and temperature is in degrees Kelvin. The water vapor partial pressure,  $W$ , is related to the water vapor density,  $\rho$ , by

$$W = \frac{\rho T}{216.7}.$$

The total atmospheric pressure is  $P + W$ .

For each oxygen line,  $S_i$  depends on two parameters,  $a_1$  and  $a_2$ . Similarly, each water vapor line depends on two parameters,  $b_1$  and  $b_2$ . The ITU documentation cited at the end of this section contains tabulations of these parameters as functions of frequency.

The localized frequency bandwidth functions  $F_i(f)$  are complicated functions of frequency described in the ITU references cited below. The functions depend on empirical model parameters that are also tabulated in the reference.

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the path length,  $R$ . Then, the total attenuation is  $L_g = R(\gamma_o + \gamma_w)$ .

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands, and apply attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

### Fog and Cloud Attenuation Model

This model calculates the attenuation of signals that propagate through fog or clouds.

Fog and cloud attenuation are the same atmospheric phenomenon. The ITU model, *Recommendation ITU-R P.840-6: Attenuation due to clouds and fog* is used. The model computes the specific attenuation (attenuation per kilometer), of a signal as a function of liquid water density, signal

frequency, and temperature. The model applies to polarized and nonpolarized fields. The formula for specific attenuation at each frequency is

$$\gamma_c = K_l(f)M,$$

where  $M$  is the liquid water density in  $\text{gm/m}^3$ . The quantity  $K_l(f)$  is the specific attenuation coefficient and depends on frequency. The cloud and fog attenuation model is valid for frequencies 10–1000 GHz. Units for the specific attenuation coefficient are  $(\text{dB/km})/(\text{g/m}^3)$ .

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the path length  $R$ . Total attenuation is  $L_c = R\gamma_c$ .

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands, and apply narrowband attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

### Rainfall Attenuation Model

This model calculates the attenuation of signals that propagate through regions of rainfall. Rain attenuation is a dominant fading mechanism and can vary from location-to-location and from year-to-year.

Electromagnetic signals are attenuated when propagating through a region of rainfall. Rainfall attenuation is computed according to the ITU rainfall model *Recommendation ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. The model computes the specific attenuation (attenuation per kilometer) of a signal as a function of rainfall rate, signal frequency, polarization, and path elevation angle. The specific attenuation,  $\gamma_R$ , is modeled as a power law with respect to rain rate

$$\gamma_R = kR^\alpha,$$

where  $R$  is rain rate. Units are in mm/hr. The parameter  $k$  and exponent  $\alpha$  depend on the frequency, the polarization state, and the elevation angle of the signal path. The specific attenuation model is valid for frequencies from 1–1000 GHz.

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the an effective propagation distance,  $d_{\text{eff}}$ . Then, the total attenuation is  $L = d_{\text{eff}}\gamma_R$ .

The effective distance is the geometric distance,  $d$ , multiplied by a scale factor

$$r = \frac{1}{0.477d^{0.633}R_{0.01}^{0.073\alpha}f^{0.123} - 10.579(1 - \exp(-0.024d))}$$

where  $f$  is the frequency. The article *Recommendation ITU-R P.530-17 (12/2017): Propagation data and prediction methods required for the design of terrestrial line-of-sight systems* presents a complete discussion for computing attenuation.

The rain rate,  $R$ , used in these computations is the long-term statistical rain rate,  $R_{0.01}$ . This is the rain rate that is exceeded 0.01% of the time. The calculation of the statistical rain rate is discussed in *Recommendation ITU-R P.837-7 (06/2017): Characteristics of precipitation for propagation modelling*. This article also explains how to compute the attenuation for other percentages from the 0.01% value.

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands and apply attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

### **Subband Frequency Processing**

Subband processing decomposes a wideband signal into multiple subbands and applies narrowband processing to the signal in each subband. The signals for all subbands are summed to form the output signal.

When using wideband frequency System objects or blocks, you specify the number of subbands,  $N_B$ , in which to decompose the wideband signal. Subband center frequencies and widths are automatically computed from the total bandwidth and number of subbands. The total frequency band is centered on the carrier or operating frequency,  $f_c$ . The overall bandwidth is given by the sample rate,  $f_s$ . Frequency subband widths are  $\Delta f = f_s/N_B$ . The center frequencies of the subbands are

Some System objects let you obtain the subband center frequencies as output when you run the object. The returned subband frequencies are ordered consistently with the ordering of the discrete Fourier transform. Frequencies above the carrier appear first, followed by frequencies below the carrier.

### **References**

- [1] Proakis, J. *Digital Communications*. New York: McGraw-Hill, 2001.
- [2] Skolnik, M. *Introduction to Radar Systems*, 3rd Ed. New York: McGraw-Hill.
- [3] Saakian, A. *Radio Wave Propagation Fundamentals*. Norwood, MA: Artech House, 2011.
- [4] Balanis, C. *Advanced Engineering Electromagnetics*. New York: Wiley & Sons, 1989.
- [5] Rappaport, T. *Wireless Communications: Principles and Practice, 2nd Ed* New York: Prentice Hall, 2002.
- [6] Radiocommunication Sector of the International Telecommunication Union. *Recommendation ITU-R P.676-10: Attenuation by atmospheric gases*. 2013.
- [7] Radiocommunication Sector of the International Telecommunication Union. *Recommendation ITU-R P.840-6: Attenuation due to clouds and fog*. 2013.
- [8] Radiocommunication Sector of the International Telecommunication Union. *Recommendation ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. 2005.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Functions

fogpl | fspl | gaspl | rangeangle | rainpl

### Objects

phased.FreeSpace | phased.LOSChannel | twoRayChannel | phased.WidebandLOSChannel |  
phased.WidebandFreeSpace | phased.WidebandBackscatterRadarTarget

### Introduced in R2021a

## reset

**System object:** widebandTwoRayChannel

Reset states of System object

### Syntax

```
reset(channel)
```

### Description

`reset(channel)` resets the internal state of the `widebandTwoRayChannel` System object, `channel`.

### Input Arguments

**channel** — Wideband two-ray channel

`widebandTwoRayChannel` System object

Wideband two-ray channel, specified as a System object.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

**Introduced in R2021a**

## step

**System object:** widebandTwoRayChannel

Propagate wideband signal from point to point using two-ray channel model

### Syntax

```
prop_sig = step(channel,sig,origin_pos,dest_pos,origin_vel,dest_vel)
```

### Description

---

**Note** Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

`prop_sig = step(channel,sig,origin_pos,dest_pos,origin_vel,dest_vel)` returns the resulting signal, `prop_sig`, when a wideband signal, `sig`, propagates through a two-ray channel from the `origin_pos` position to the `dest_pos` position. Either the `origin_pos` or `dest_pos` arguments can have multiple points but you cannot specify both as having multiple points. Specify the velocity of the signal origin in `origin_vel` and the velocity of the signal destination in `dest_vel`. The dimensions of `origin_vel` and `dest_vel` must agree with the dimensions of `origin_pos` and `dest_pos`, respectively.

In the two-ray environment, two signal paths connect every signal origin and destination pair. For  $N$  signal origins (or  $N$  signal destinations), there are  $2N$  paths. The signals for each origin-destination pair do not have to be identical. The signals along the two paths for any source-destination pair can have different amplitudes or phases.

The `CombinedRaysOutput` property controls whether the two signals at the destination are kept *separate* or *combined*. *Combined* means that the signals at the source propagate separately along the two paths but are coherently summed at the destination into a single quantity. *Separate* means that the two signals are not summed at the destination. To use the *combined* option, set `CombinedRaysOutput` to `true`. To use the *separate* option, set `CombinedRaysOutput` to `false`. The *combined* option is convenient when the difference between the sensor or array gains in the directions of the two paths is not significant.

---

**Note** The object performs an initialization the first time the object is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

### Input Arguments

**channel** — Wideband two-ray channel

System object

Wideband two-ray channel, specified as a System object.

Example: `widebandTwoRayChannel`

### **sig** — Wideband signal

$M$ -by- $N$  complex-valued matrix |  $M$ -by- $2N$  complex-valued matrix | 1-by- $N$  struct array containing complex-valued fields | 1-by- $2N$  struct array containing complex-valued fields

Electromagnetic fields propagated through a two-ray channel can be polarized or nonpolarized. For nonpolarized fields, such as an acoustic field, the propagating signal field, `sig`, is a vector or matrix. When the fields are polarized, `sig` is an array of structures. Every structure element contains an array of electric field vectors in Cartesian form.

- Specify wideband nonpolarized scalar signals as a
  - $M$ -by- $N$  complex-valued matrix. The same signal is propagated along both the line-of-sight path and the reflected path.
  - $M$ -by- $2N$  complex-valued matrix. Each adjacent pair of columns represents a different channel. Within each pair, the first column represents the signal propagated along the line-of-sight path and the second column represents the signal propagated along the reflected path.
- Specify wideband polarized signals as a
  - 1-by- $N$  struct array containing complex-valued fields. Each struct element contains an  $M$ -by-1 column vector of electromagnetic field components (`sig.X`, `sig.Y`, `sig.Z`). The same signal is propagated along both the line-of-sight path and the reflected path.
  - 1-by- $2N$  struct array containing complex-valued fields. Each pair of array columns represents a different source-receiver channel. The first column of the pair represents the signal along the line-of-sight path and the second column represents the signal along the reflected path. Each structure element contains an  $M$ -by-1 column vector of electromagnetic field components (`sig.X`, `sig.Y`, `sig.Z`).

For nonpolarized fields, the quantity  $M$  is the number of samples of the signal and  $N$  is the number of two-ray channels. Each channel corresponds to a source-destination pair.

The size of the first dimension of the input matrix can vary to simulate a changing signal length. A size change can occur, for example, in the case of a pulse waveform with variable pulse repetition frequency.

For polarized fields, the struct element contains three  $M$ -by-1 complex-valued column vectors, `sig.X`, `sig.Y`, and `sig.Z`. These vectors represent the  $x$ ,  $y$ , and  $z$  Cartesian components of the polarized signal.

The size of the first dimension of the matrix fields within the struct can vary to simulate a changing signal length such as a pulse waveform with variable pulse repetition frequency.

Example: `[1,1;j,1;0.5,0]`

Data Types: `double`

Complex Number Support: Yes

### **origin\_pos** — Signal origins

3-by-1 real-valued column vector | 3-by- $N$  real-valued matrix

Origin of the signal or signals, specified as a 3-by-1 real-valued column vector or 3-by- $N$  real-valued matrix. The quantity  $N$  is the number of two-ray channels. If `origin_pos` is a column vector, it takes



the form  $[x;y;z]$ . If `origin_pos` is a matrix, each column specifies a different signal origin and has the form  $[x;y;z]$ . Position units are in meters.

You cannot specify both `origin_pos` and `dest_pos` as matrices. At least one must be a 3-by-1 column vector.

Example: `[1000;100;500]`

Data Types: `double`

### **dest\_pos — Signal destinations**

3-by-1 real-valued column vector | 3-by- $N$  real-valued matrix

Destination position of the signal or signals, specified as a 3-by-1 real-valued column vector or 3-by- $N$  real-valued matrix. The quantity  $N$  is the number of two-ray channels propagating from or to  $N$  signal origins. If `dest_pos` is a 3-by-1 column vector, it takes the form  $[x;y;z]$ . If `dest_pos` is a matrix, each column specifies a different signal destination and takes the form  $[x;y;z]$ . Position units are in meters.

You cannot specify both `origin_pos` and `dest_pos` as matrices. At least one must be a 3-by-1 column vector.

Example: `[0;0;0]`

Data Types: `double`

### **origin\_vel — Velocity of signal origin**

3-by-1 real-valued column vector | 3-by- $N$  real-valued matrix

Velocity of signal origin, specified as a 3-by-1 real-valued column vector or 3-by- $N$  real-valued matrix. The dimensions of `origin_vel` must match the dimensions of `origin_pos`. If `origin_vel` is a column vector, it takes the form  $[V_x;V_y;V_z]$ . If `origin_vel` is a 3-by- $N$  matrix, each column specifies a different origin velocity and has the form  $[V_x;V_y;V_z]$ . Velocity units are in meters per second.

Example: `[10;0;5]`

Data Types: `double`

### **dest\_vel — Velocity of signal destinations**

3-by-1 real-valued column vector | 3-by- $N$  real-valued matrix

Velocity of signal destinations, specified as a 3-by-1 real-valued column vector or 3-by- $N$  real-valued matrix. The dimensions of `dest_vel` must match the dimensions of `dest_pos`. If `dest_vel` is a column vector, it takes the form  $[V_x;V_y;V_z]$ . If `dest_vel` is a 3-by- $N$  matrix, each column specifies a different destination velocity and has the form  $[V_x;V_y;V_z]$ . Velocity units are in meters per second.

Example: `[0;0;0]`

Data Types: `double`

## **Output Arguments**

### **prop\_sig — Propagated signal**

$M$ -by- $N$  complex-valued matrix |  $M$ -by- $2N$  complex-valued matrix | 1-by- $N$  struct array containing complex-valued fields | 1-by- $2N$  struct array containing complex-valued fields

- Wideband nonpolarized scalar signal, returned as an:
  - $M$ -by- $N$  complex-valued matrix. To return this format, set the `CombinedRaysOutput` property to `true`. Each matrix column contains the coherently combined signals from the line-of-sight path and the reflected path.
  - $M$ -by- $2N$  complex-valued matrix. To return this format set the `CombinedRaysOutput` property to `false`. Alternate columns of the matrix contain the signals from the line-of-sight path and the reflected path.
- Wideband polarized scalar signal, returned as:
  - 1-by- $N$  struct array containing complex-valued fields. To return this format, set the `CombinedRaysOutput` property to `true`. Each column of the array contains the coherently combined signals from the line-of-sight path and the reflected path. Each structure element contains the electromagnetic field vector (`prop_sig.X,prop_sig.Y,prop_sig.Z`).
  - 1-by- $2N$  struct array containing complex-valued fields. To return this format, set the `CombinedRaysOutput` property to `false`. Alternate columns contains the signals from the line-of-sight path and the reflected path. Each structure element contains the electromagnetic field vector (`prop_sig.X,prop_sig.Y,prop_sig.Z`).

The output `prop_sig` contains signal samples arriving at the signal destination within the current input time frame. Sometimes it can take longer than the current time frame for the signal to propagate from the origin to the destination, the output may not contain all contributions from the input of the current time frame. In this case, the output does not need to contain all contributions from the input of the current time frame. The remaining output appears in the next call to `step`.

## Examples

### Scalar Wideband Signal Propagating in Two-Ray Channel

This example illustrates the two-ray propagation of a wideband signal, showing how the signals from the line-of-sight path and reflected path arrive at the receiver at different times.

**Note:** You can replace each call to the function with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

#### Create and Plot Transmitted Waveform

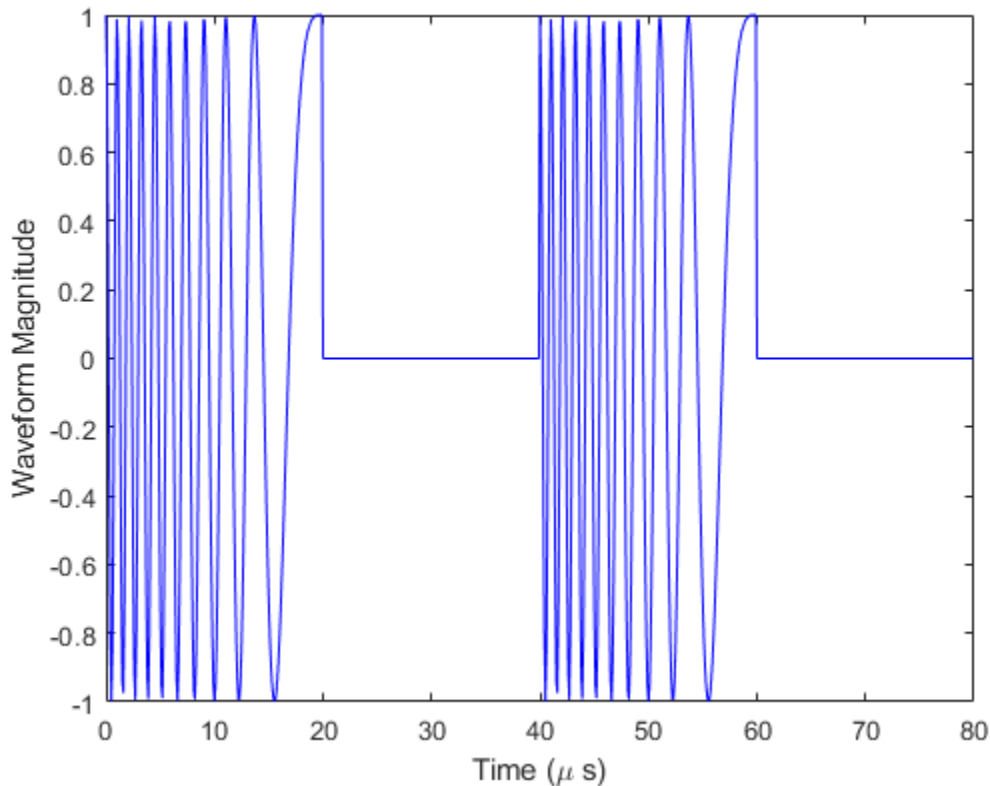
Create a nonpolarized electromagnetic field consisting of two linear FM waveform pulses at a carrier frequency of 100 MHz. Assume the pulse width is 20  $\mu$ s and the sampling rate is 10 MHz. The bandwidth of the pulse is 1 MHz. Assume a 50% duty cycle so that the pulse width is one-half the pulse repetition interval. Create a two-pulse wave train. Set the `GroundReflectionCoefficient` to `-0.9` to model strong ground reflectivity. Propagate the field from a stationary source to a stationary receiver. The vertical separation of the source and receiver is approximately 10 km.

```
c = physconst('LightSpeed');
fs = 10e6;
pw = 20e-6;
pri = 2*pw;
PRF = 1/pri;
fc = 100e6;
lambda = c/fc;
bw = 1e6;
```

```

waveform = phased.LinearFMWaveform('SampleRate',fs,'PulseWidth',pw,...
    'PRF',PRF,'OutputFormat','Pulses','NumPulses',2,'SweepBandwidth',bw,...
    'SweepDirection','Down','Envelope','Rectangular','SweepInterval',...
    'Positive');
wav = waveform();
n = size(wav,1);
plot([0:(n-1)]/fs*1e6,real(wav),'b')
xlabel('Time (\mu s)')
ylabel('Waveform Magnitude')

```



### Specify the Location of Source and Receiver

Place the source and receiver about 1 km apart horizontally and approximately 5 km apart vertically.

```

pos1 = [0;0;100];
pos2 = [1e3;0;5.0e3];
vel1 = [0;0;0];
vel2 = [0;0;0];

```

### Create a Wideband Two-Ray Channel System Object

Create a two-ray propagation channel System object™ and propagate the signal along both the line-of-sight and reflected ray paths. The same signal is propagated along both paths.

```

channel = widebandTwoRayChannel('SampleRate',fs,...
    'GroundReflectionCoefficient',-0.9,'OperatingFrequency',fc,...
    'CombinedRaysOutput',false);
prop_signal = channel([wav,wav],pos1,pos2,vel1,vel2);

```

```
[rng2,angs] = rangeangle(pos2,pos1, 'two-ray');
```

Calculate time delays in  $\mu\text{s}$ .

```
tm = rng2/c*1e6;  
disp(tm)
```

```
16.6815 17.3357
```

Display the calculated propagation paths azimuth and elevation angles in degrees.

```
disp(angs)
```

```
0 0  
78.4654 -78.9063
```

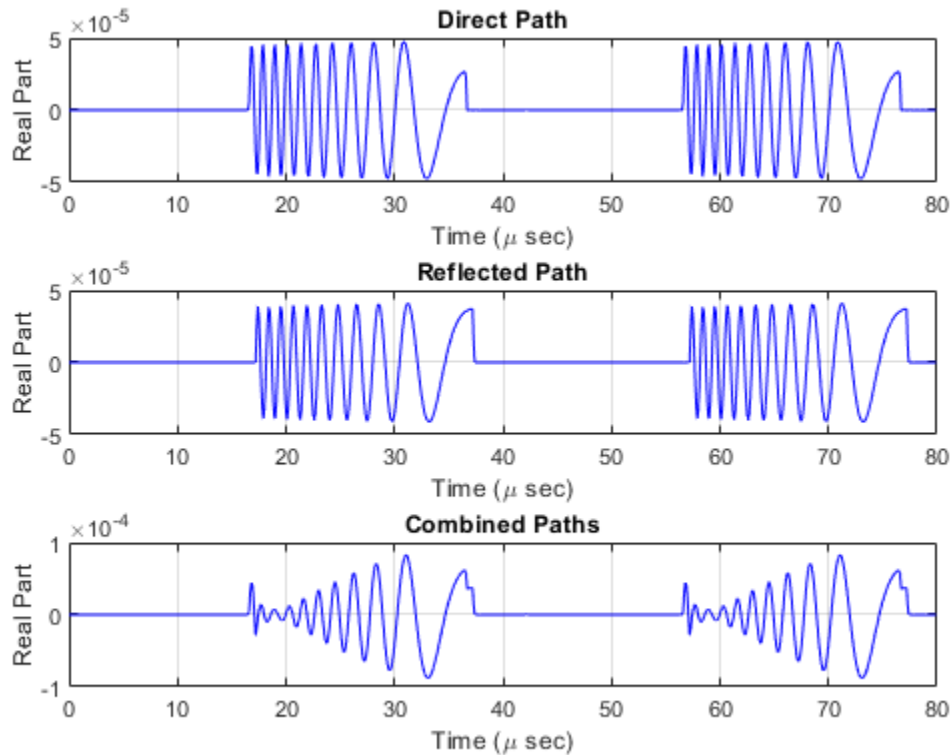
### Plot the Propagated Signals

- 1 Plot the real part of the signal propagated along the line-of-sight path.
- 2 Plot the real part of the signal propagated along the reflected path.
- 3 Plot the real part of the coherent sum of the two signals.

```
n = size(prop_signal,1);  
delay = [0:(n-1)]/fs*1e6;  
subplot(3,1,1)  
plot(delay,real([prop_signal(:,1)]),'b')  
grid  
xlabel('Time (\mu sec)')  
ylabel('Real Part')  
title('Direct Path')
```

```
subplot(3,1,2)  
plot(delay,real([prop_signal(:,2)]),'b')  
grid  
xlabel('Time (\mu sec)')  
ylabel('Real Part')  
title('Reflected Path')
```

```
subplot(3,1,3)  
plot(delay,real([prop_signal(:,1) + prop_signal(:,2)]),'b')  
grid  
xlabel('Time (\mu sec)')  
ylabel('Real Part')  
title('Combined Paths')
```



The delay of the reflected path signal agrees with the predicted delay. The magnitude of the coherently combined signal is less than either of the propagated signals. This result indicates that the two signals contain some interference.

### Compare Wideband Two-Ray Channel Propagation to Free Space

Compute the result of propagating a wideband LFM signal in a two-ray environment from a radar 10 meters above the origin  $(0,0,10)$  to a target at  $(3000,2000,2000)$  meters. Assume that the radar and target are stationary and that the transmitting antenna is isotropic. Combine the signal from the two paths and compare the signal to a signal propagating in free space. The system operates at 300 MHz. Set the `CombinedRaysOutput` property to `true` to combine the direct path and reflected path signals when forming the output signal.

**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)`.

Create a linear FM waveform.

```
fop = 300.0e6;
fs = 1.0e6;
waveform = phased.LinearFMWaveform();
x = waveform();
```

Specify the target position and velocity.

```

posTx = [0; 0; 10];
posTgt = [3000; 2000; 2000];
velTx = [0;0;0];
velTgt = [0;0;0];

```

Model the free space propagation.

```

fschannel = phased.WidebandFreeSpace('SampleRate', waveform.SampleRate);
y_fs = fschannel(x, posTx, posTgt, velTx, velTgt);

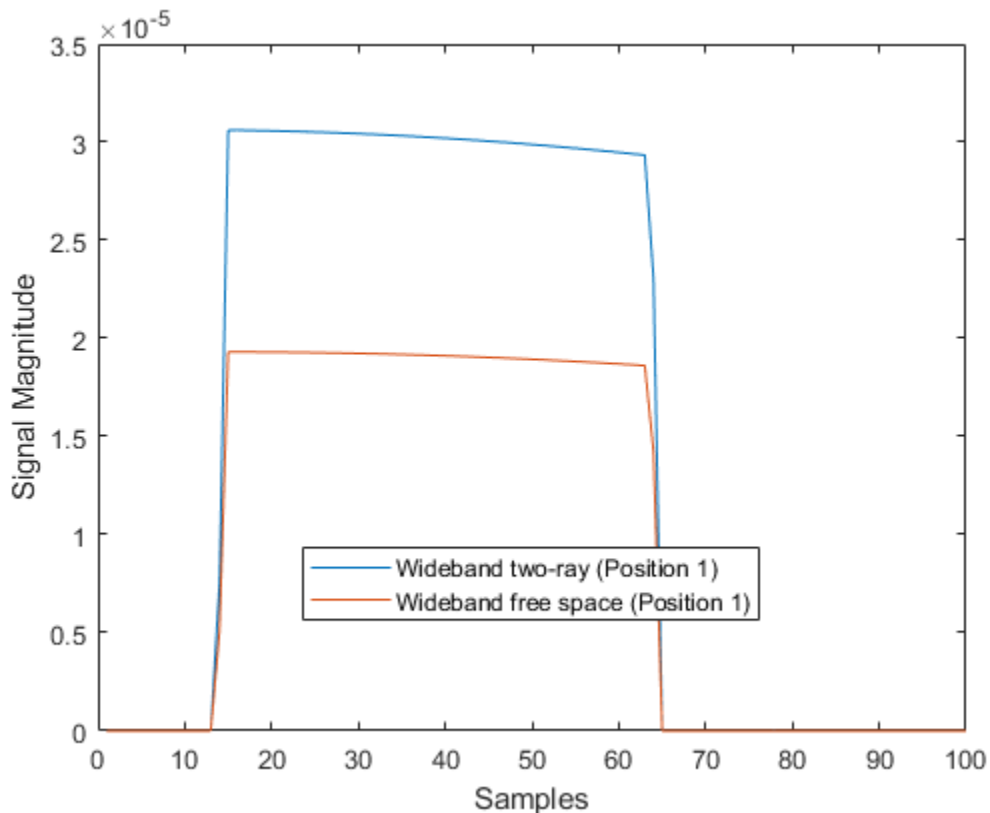
```

Model two-ray propagation from the position of the radar to the target.

```

tworaychannel = widebandTwoRayChannel('SampleRate', waveform.SampleRate, ...
    'CombinedRaysOutput', true);
y_tworay = tworaychannel(x, posTx, posTgt, velTx, velTgt);
plot(abs([y_tworay y_fs]))
legend('Wideband two-ray (Position 1)', 'Wideband free space (Position 1)', ...
    'Location', 'best')
xlabel('Samples')
ylabel('Signal Magnitude')
hold on

```



Move the radar by 10 meters horizontally to a second position.

```

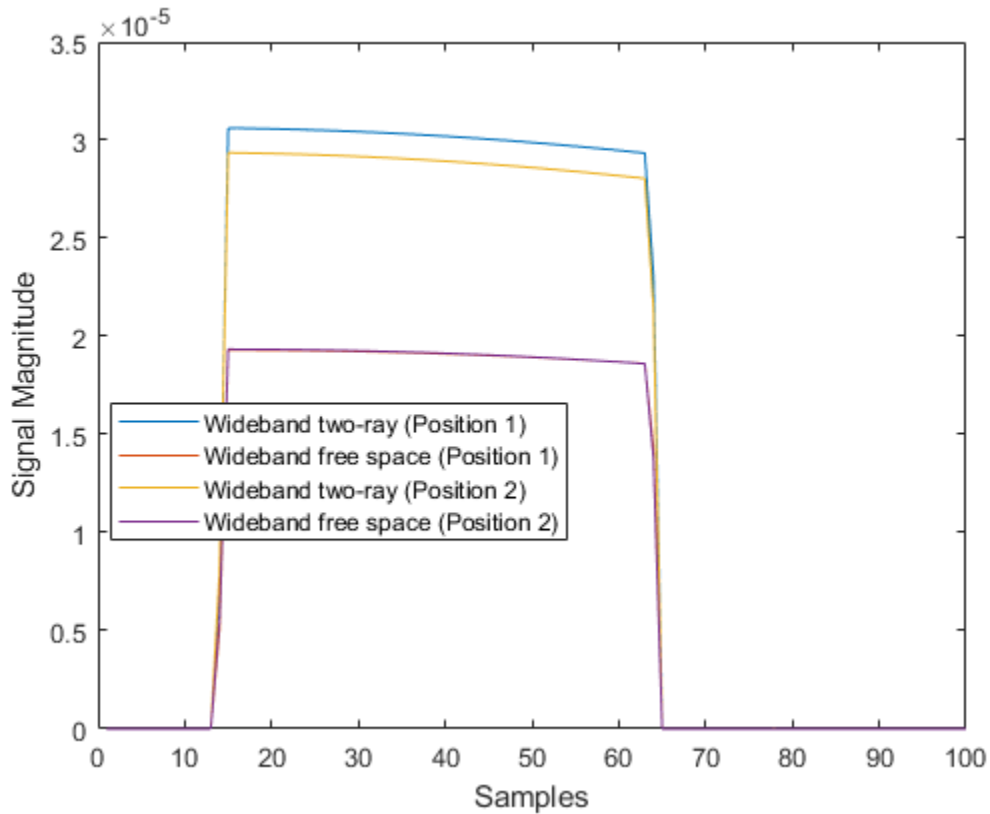
posTx = posTx + [10;0;0];
y_fs = fschannel(x, posTx, posTgt, velTx, velTgt);
y_tworay = tworaychannel(x, posTx, posTgt, velTx, velTgt);
plot(abs([y_tworay y_fs]))

```

```

legend('Wideband two-ray (Position 1)', 'Wideband free space (Position 1)', ...
       'Wideband two-ray (Position 2)', 'Wideband free space (Position 2)', ...
       'Location', 'best')
hold off

```



The free-space propagation losses are the same for both the first and second positions of the radar. The two-ray losses are different due to the interference effect of the two-ray paths.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### Introduced in R2021a

# pulseCompressionLibrary

Create a library of pulse compression specifications

## Description

The `pulseCompressionLibrary` System object creates a pulse compression library. The library contains sets of parameters that describe pulse compression operations performed on received signals to generate their range response. You can use this library to perform matched filtering or stretch processing. This object can process waveforms created by the `pulseWaveformLibrary` object.

To make a pulse compression library

- 1 Create the `pulseCompressionLibrary` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
complib = pulseCompressionLibrary()
complib = pulseCompressionLibrary(Name,Value)
```

### Description

`complib = pulseCompressionLibrary()` System object creates a pulse compression library, `complib`, with default property values.

`complib = pulseCompressionLibrary(Name,Value)` creates a pulse compression library with each property `Name` set to a specified `Value`. You can specify additional name-value pair arguments in any order as `(Name1,Value1,...,NameN,ValueN)`. Enclose each property name in single quotes.

Example: `complib = pulseCompressionLibrary('SampleRate',1e9,'WaveformSpecification',{{'Rectangular','PRF',1e4,'PulseWidth',100e-6},{'SteppedFM','PRF',1e4}},'ProcessingSpecification',{'MatchedFilter','SpectrumWindow','Hann'},{'MatchedFilter','SpectrumWindow','Taylor'})` creates a library with two matched filters. One is matched to a rectangular waveform and the other to a stepped FM waveform. The matched filters use a Hann window and a Taylor window, respectively.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.



If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### **SampleRate — Waveform sample rate**

1e6 (default) | positive scalar

Waveform sample rate, specified as a positive scalar. All waveforms have the same sample rate. Units are in hertz.

Example: 100e3

Data Types: double

### **PropagationSpeed — Signal propagation speed**

physconst('LightSpeed') (default) | positive scalar

Signal propagation speed, specified as a positive scalar. Units are in meters per second. The default propagation speed is the value returned by `physconst('LightSpeed')`. See `physconst` for more information.

Example: 3e8

Data Types: double

### **WaveformSpecification — Pulse waveforms**

```
{{'Rectangular','PRF',10e3,'PulseWidth',100e-6},
{'LinearFM','PRF',1e4,'PulseWidth',50e-6,'SweepBandwidth',1e5,'SweepDirection',
'Up','SweepInterval','Positive'}} (default) | cell array
```

Pulse waveforms, specified as a cell array. Each cell of the array contains the specification of one waveform.

```
{{Waveform 1 Specification},{Waveform 2 Specification},{Waveform 3 Specification}, ...}
```

Each waveform specification is also a cell array containing the parameters of the waveform. The entries in a specification cell are the pulse identifier and a set of name-value pairs specific to that waveform.

```
{PulseIdentifier,Name1,Value1,Name2,Value2, ...}
```

This System object supports four built-in waveforms and also lets you specify custom waveforms. For the built-in waveforms, the waveform specifier consists of a waveform identifier followed by several name-value pairs setting the properties of the waveform. For the custom waveforms, the waveform specifier consists of a handle to a user-defined waveform function and the functions input arguments.

## Waveform Types

Pulse Type	Pulse Identifier	Waveform Arguments
Linear FM	'LinearFM'	"Linear FM Waveform Arguments" on page 4-379
Phase coded	'PhaseCoded'	"Phase-Coded Waveform Arguments" on page 4-381
Rectangular	'Rectangular'	"Rectangular Waveform Arguments" on page 4-382
Stepped FM	'SteppedFM'	"Stepped FM Waveform Arguments" on page 4-383
Custom	<i>Function handle</i>	"Custom Waveform Arguments" on page 4-399

Example: `{{'Rectangular','PRF',10e3,'PulseWidth',100e-6},  
'Rectangular','PRF',100e3,'PulseWidth',20e-6}}`

Data Types: cell

### ProcessingSpecification — Pulse compression descriptions

`{{'MatchedFilter','SpectrumWindow','None'},  
'StretchProcessor','RangeSpan',200,'ReferenceRange',5e3,'RangeWindow','None'  
}}` (default) | cell array

Pulse compression descriptions, specified as a cell array of processing specifications. Each cell defines a different processing specification. Each processing specification is itself a cell array containing the processing type and processing arguments.

`{{Processing 1 Specification},{Processing 2 Specification},{Processing 3 Specification}, ...}`

Each processing specification indicates which type of processing to apply to a waveform and the arguments needed for processing.

`{ProcessType,Name,Value,...}`

The value of `ProcessType` is either `'MatchedFilter'` or `'StretchProcessor'`.

- `'MatchedFilter'` - The name-value pair arguments are
  - `'Coefficients',coeff` - specifies the matched filter coefficients, `coeff`, as a column vector. When not specified, the coefficients are calculated from the `WaveformSpecification` property. For the Stepped FM waveform containing multiple pulses, `coeff` corresponds to each pulse until the pulse index, `idx` changes.
  - `'SpectrumWindow',sw` - specifies the spectrum weighting window, `sw`, applied to the waveform. Window values are one of `'None'`, `'Hamming'`, `'Chebyshev'`, `'Hann'`, `'Kaiser'`, and `'Taylor'`. The default value is `'None'`.
  - `'SidelobeAttenuation',slb` - specifies the sidelobe attenuation window, `slb`, of the Chebyshev or Taylor window as a positive scalar. The default value is 30. This parameter applies when you set `'SpectrumWindow'` to `'Chebyshev'` or `'Taylor'`.
  - `'Beta',beta` - specifies the parameter, `beta`, that determines the Kaiser window sidelobe attenuation as a nonnegative scalar. The default value is 0.5. This parameter applies when you set `'SpectrumWindow'` to `'Kaiser'`.

- 'Nbar', nbar – specifies the number of nearly constant level sidelobes, nbar, next to the main lobe in a Taylor window as a positive integer. The default value is 4. This parameter applies when you set 'SpectrumWindow' to 'Taylor'.
- 'SpectrumRange', sr – specifies the spectrum region, sr, on which the spectrum window is applied as a 1-by-2 vector having the form [StartFrequency EndFrequency]. The default value is [0 1.0e5]. This parameter applies when you set the 'SpectrumWindow' to any value other than 'None'. Units are in Hz.

Both StartFrequency and EndFrequency are measured in the baseband region  $[-Fs/2 Fs/2]$ .  $F_s$  is the sample rate specified by the SampleRate property. StartFrequency cannot be larger than EndFrequency.

- 'StretchProcessor' – The name-value pair arguments are
  - 'ReferenceRange', refrng – specifies the center of the ranges of interest, refrng, as a positive scalar. The refrng must be within the unambiguous range of one pulse. The default value is 5000. Units are in meters.
  - 'RangeSpan', rngspan – specifies the span of the ranges of interest. rngspan, as a positive scalar. The range span is centered at the range value specified in the 'ReferenceRange' parameter. The default value is 500. Units are in meters.
  - 'RangeFFTLength', len – specifies the FFT length in the range domain, len, as a positive integer. If not specified, the default value is same as the input data length.
  - 'RangeWindow', rw specifies the window used for range processing, rw, as one of 'None', 'Hamming', 'Chebyshev', 'Hann', 'Kaiser', and 'Taylor'. The default value is 'None'.

Example: 'StretchProcessor'

Data Types: string | struct

### Linear FM Waveform Arguments

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: {'LinearFM', 'PRF', 1e4, 'PulseWidth', 50e-6, 'SweepBandwidth', 1e5, . . . 'SweepDirection', 'Up', 'SweepInterval', 'Positive'}

### PRF — Pulse repetition frequency

1e4 (default) | positive scalar

Pulse repetition frequency (PRF), specified as a positive scalar. Units are in hertz. See “Pulse Repetition Frequency Restrictions” on page 4-406 for restrictions on the PRF.

Example: 20e3

Data Types: double

### PulseWidth — Pulse duration

5e-5 (default) | positive scalar

Pulse duration, specified as a positive scalar. Units are in seconds. You cannot specify both PulseWidth and DutyCycle.

Example: 100e-6

Data Types: double

**DutyCycle — Pulse duty cycle**

0.5 | positive scalar

Pulse duty cycle, specified as a positive scalar greater than zero and less than or equal to one. You cannot specify both `PulseWidth` and `DutyCycle`.

Example: 0.7

Data Types: double

**SweepBandwidth — Bandwidth of the FM sweep**

1e5 (default) | positive scalar

Bandwidth of the FM sweep, specified as a positive scalar. Units are in hertz.

Example: 100e3

Data Types: double

**SweepDirection — Bandwidth of the FM sweep**

'Up' (default) | 'Down'

Direction of the FM sweep, specified as 'Up' or 'Down'. 'Up' corresponds to increasing frequency. 'Down' corresponds to decreasing frequency.

Data Types: char

**SweepInterval — FM sweep interval**

'Positive' (default) | 'Symmetric'

FM sweep interval, specified as 'Positive' or 'Symmetric'. If you set this property value to 'Positive', the waveform sweeps the interval between 0 and  $B$ , where  $B$  is the `SweepBandwidth` argument value. If you set this property value to 'Symmetric', the waveform sweeps the interval between  $-B/2$  and  $B/2$ .

Example: 'Symmetric'

Data Types: char

**Envelope — Envelope function**

'Rectangular' (default) | 'Gaussian'

Envelope function, specified as 'Rectangular' or 'Gaussian'.

Example: 'Gaussian'

Data Types: char

**FrequencyOffset — Frequency offset of pulse**

0 (default) | scalar

Frequency offset of pulse, specified as a scalar. The frequency offset shifts the frequency of the generated pulse waveform. Units are in hertz.

Example: 100e3

Data Types: double

### Phase-Coded Waveform Arguments

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: {'PhaseCoded', 'PRF', 1e4, 'Code', 'Zadoff-Chu', 'SequenceIndex', 3, 'ChipWidth', 5e-6, 'NumChips', 8}

### PRF — Pulse repetition frequency

1e4 (default) | positive scalar

Pulse repetition frequency (PRF), specified as a positive scalar. Units are in hertz. See “Pulse Repetition Frequency Restrictions” on page 4-406 for restrictions on the PRF.

Example: 20e3

Data Types: double

### Code — Type of phase modulation code

'Frank' (default) | 'P1' | 'P2' | 'Px' | 'Zadoff-Chu' | 'P3' | 'P4' | 'Barker'

Type of phase modulation code, specified as 'Frank', 'P1', 'P2', 'Px', 'Zadoff-Chu', 'P3', 'P4', or 'Barker'.

Example: 'P1'

Data Types: char

### SequenceIndex — Zadoff-Chu sequence index

1 (default) | positive integer

Sequence index used for the Zadoff-Chu code, specified as a positive integer. The value of SequenceIndex must be relatively prime to the value of NumChips.

Example: 3

### Dependencies

To enable this name-value pair, set the Code property to 'Zadoff-Chu'.

Data Types: double

### ChipWidth — Chip duration

1e-5 (default) | positive scalar

Chip duration, specified as a positive scalar. Units are in seconds. See “Chip Restrictions” on page 4-407 for restrictions on chip sizes.

Example: 30e-3

Data Types: double

### NumChips — Number of chips in waveform

4 (default) | positive integer

Number of chips in waveform, specified as a positive integer. See “Chip Restrictions” on page 4-407 for restrictions on chip sizes.

Example: 3

Data Types: double

**FrequencyOffset — Frequency offset of pulse**

0 (default) | scalar

Frequency offset of pulse, specified as a scalar. The frequency offset shifts the frequency of the generated pulse waveform. Units are in hertz.

Example: 100e3

Data Types: double

**Rectangular Waveform Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: {'Rectangular', 'PRF', 10e3, 'PulseWidth', 100e-6}

**PRF — Pulse repetition frequency**

1e4 (default) | positive scalar

Pulse repetition frequency (PRF), specified as a positive scalar. Units are in hertz. See “Pulse Repetition Frequency Restrictions” on page 4-406 for restrictions on the PRF.

Example: 20e3

Data Types: double

**PulseWidth — Pulse duration**

5e-5 (default) | positive scalar

Pulse duration, specified as a positive scalar. Units are in seconds. You cannot specify both `PulseWidth` and `DutyCycle`.

Example: 100e-6

Data Types: double

**DutyCycle — Pulse duty cycle**

0.5 | positive scalar

Pulse duty cycle, specified as a positive scalar greater than zero and less than or equal to one. You cannot specify both `PulseWidth` and `DutyCycle`.

Example: 0.7

Data Types: double

**FrequencyOffset — Frequency offset of pulse**

0 (default) | scalar

Frequency offset of pulse, specified as a scalar. The frequency offset shifts the frequency of the generated pulse waveform. Units are in hertz.

Example: 100e3

Data Types: double

### Stepped FM Waveform Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: {'SteppedFM', 'PRF', 10e-4}

### PRF — Pulse repetition frequency

1e4 (default) | positive scalar

Pulse repetition frequency (PRF), specified as a positive scalar. Units are in hertz. See “Pulse Repetition Frequency Restrictions” on page 4-406 for restrictions on the PRF.

Example: 20e3

Data Types: double

### PulseWidth — Pulse duration

5e-5 (default) | positive scalar

Pulse duration, specified as a positive scalar. Units are in seconds. You cannot specify both `PulseWidth` and `DutyCycle`.

Example: 100e-6

Data Types: double

### DutyCycle — Pulse duty cycle

0.5 | positive scalar

Pulse duty cycle, specified as a positive scalar greater than zero and less than or equal to one. You cannot specify both `PulseWidth` and `DutyCycle`.

Example: 0.7

Data Types: double

### NumSteps — Number of frequency steps in waveform

5 (default) | positive integer

Number of frequency steps in waveform, specified as a positive integer.

Example: 3

Data Types: double

### FrequencyStep — Linear frequency step size

20e3 (default) | positive scalar

Linear frequency step size, specified as a positive scalar.

Example: 100.0

Data Types: double

### **FrequencyOffset — Frequency offset of pulse**

0 (default) | scalar

Frequency offset of pulse, specified as a scalar. The frequency offset shifts the frequency of the generated pulse waveform. Units are in hertz.

Example: 100e3

Data Types: double

### **Custom Waveform Arguments**

You can create a custom waveform from a user-defined function. The first input argument of the function must be the sample rate. For example, specify a hyperbolic waveform function,

```
function wav = HyperbolicFM(fs,prf,pw,freq,bw,fcent),
```

where *fs* is the sample rate and *prf*, *pw*, *freq*, *bw*, and *fcent* are other waveform arguments. The function must have at least one output argument, *wav*, to return the samples of each pulse. This output must be a column vector. There can be other outputs returned following the waveform samples.

Then, create a waveform specification using a function handle instead of the waveform identifier. The first cell in the waveform specification must be a function handle. The remaining cells contain all function input arguments except the sample rate. Specify all input arguments in the order they are passed into the function.

```
waveformspec = {@HyperbolicFM,prf,pw,freq,bw,fcent}
```

See “Add Custom Waveform to Pulse Waveform Library” on page 4-405 for an example that uses a custom waveform.

## **Usage**

### **Syntax**

```
[Y,rng] = pulselib(X,idx)
```

### **Description**

`[Y,rng] = pulselib(X,idx)` returns samples of a compressed pulse waveform, *Y*, specified by its index, *idx*, in the library. *RNG* denotes the ranges corresponding to *Y*.

### **Input Arguments**

#### **X — Input signal**

complex-valued *K*-by-*L* matrix | complex-valued *K*-by-*N* matrix | complex-valued *K*-by-*N*-by-*L* array

Input signal, specified as a complex-valued *K*-by-*L* matrix, complex-valued *K*-by-*N* matrix, or a complex-valued *K*-by-*N*-by-*L* array. *K* denotes the number of fast time samples, *L* the number of pulses, and *N* is the number of channels. Channels can be array elements or beams.



Data Types: double

### **idx — Index of processing specification in pulse compression library**

positive integer

Index of the processing specification in the pulse compression library, specified as a positive integer.

Data Types: double

### **Output Arguments**

#### **Y — Output signal**

complex-valued  $K$ -by- $L$  matrix | complex-valued  $K$ -by- $N$  matrix | complex-valued  $K$ -by- $N$ -by- $L$  array

Output signal, returned as a complex-valued  $M$ -by- $L$  matrix, complex-valued  $M$ -by- $N$  matrix, or a complex-valued  $M$ -by- $N$ -by- $L$  array.  $M$  denotes the number of fast time samples,  $L$  the number of pulses, and  $N$  is the number of channels. Channels can be array elements or beams. The number of dimensions of  $Y$  matches the number of dimensions of  $X$ .

When matched filtering is performed,  $M$  is equal to the number of rows in  $X$ . When stretch processing is performed and you specify a value for the RangeFFTLenght name-value pair,  $M$  is set to the value of RangeFFTLenght. When you do not specify RangeFFTLenght,  $M$  is equal to the number of rows in  $X$ .

Data Types: double

#### **rng — Sample range**

real-valued length- $M$  vector

Sample ranges, returned as a real-valued length- $M$  vector where  $M$  is the number of rows of  $Y$ . Elements of this vector denote the ranges corresponding to the rows of  $Y$ .

Data Types: double

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## **Specific to pulseCompressionLibrary**

plotResponse Plot range response from pulse compression library

## **Common to All System Objects**

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

## **Examples**

### Range Processing of Two Waveforms

Create a rectangular waveform and a linear FM waveform. Use the processing methods in the pulse compression library to range-process the waveforms. Use matched filtering for the rectangular waveform and stretch processing for the linear FM waveform.

Create two waveforms using the `pulseWaveformLibrary` System object™. The sampling frequency is 1 MHz and the pulse repetition frequency for both waveforms is 1 kHz. The pulse width is also the same at 50 microsec.

```
fs = 1.0e6;
prf = 1e3;
pw = 50e-6;
waveform1 = {'Rectangular', 'PRF', prf, 'PulseWidth', pw};
waveform2 = {'LinearFM', 'PRF', prf, 'PulseWidth', pw, ...
    'SweepBandwidth', 1e5, 'SweepDirection', 'Up', ...
    'SweepInterval', 'Positive'};
pulselib = pulseWaveformLibrary('WaveformSpecification', ...
    {waveform1, waveform2}, 'SampleRate', fs);
```

Retrieve the waveforms for processing by the pulse compression library.

```
rectwav = pulselib(1);
lfmwav = pulselib(2);
```

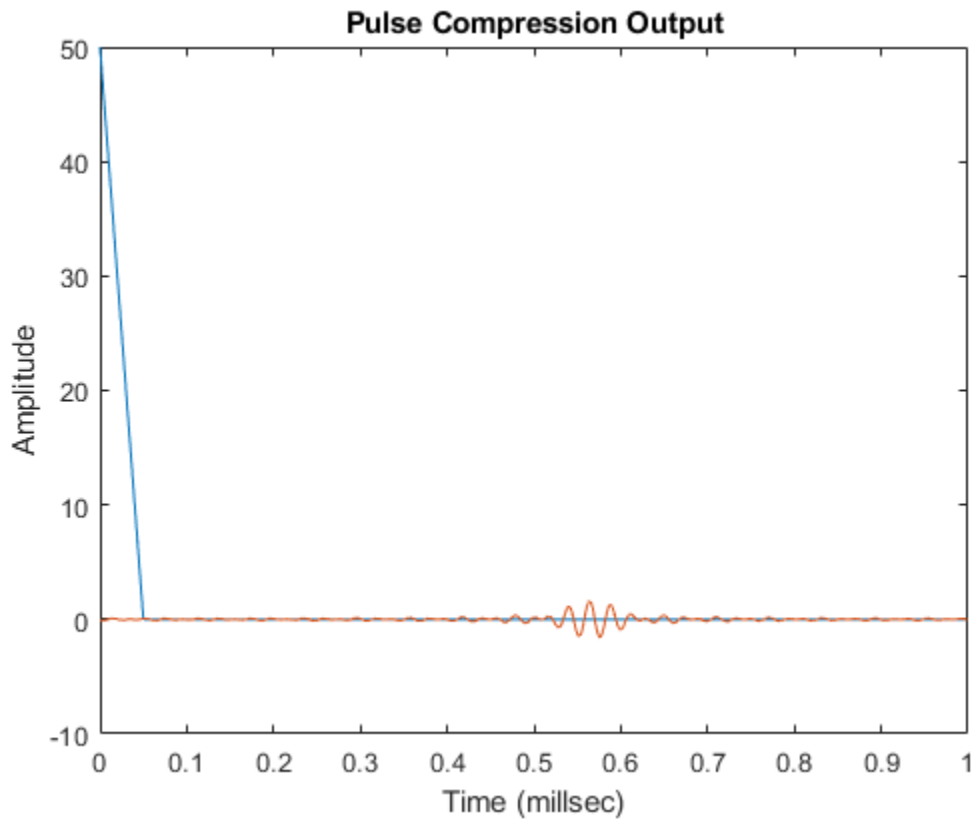
Create the compression processing library using the `pulseCompressionLibrary` System object™ with two processing specifications. The first processing specification is matched filtering and the second is stretch processing.

```
mf = getMatchedFilter(pulselib, 1);
procspec1 = {'MatchedFilter', 'Coefficients', mf};
procspec2 = {'StretchProcessor', 'ReferenceRange', 5000, ...
    'RangeSpan', 200, 'RangeWindow', 'Hamming'};
comprlib = pulseCompressionLibrary( ...,
    'WaveformSpecification', {waveform1, waveform2}, ...
    'ProcessingSpecification', {procspec1, procspec2}, ...
    'SampleRate', fs, 'PropagationSpeed', physconst('Lightspeed'));
```

Process both waveforms.

```
rect_out = comprlib(rectwav, 1);
lfm_out = comprlib(lfmwav, 2);
nsamp = fs/prf;
t = [0:(nsamp-1)]/fs;

plot(t*1000, real(rect_out))
hold on
plot(t*1000, real(lfm_out))
hold off
title('Pulse Compression Output')
xlabel('Time (millsec)')
ylabel('Amplitude')
```



### Range Response for Three Targets

Plot the range response of an LFM signal hitting three targets at ranges of 2000, 4000, and 5500 meters. Assuming the maximum range of the radar is 10 km, determine the pulse repetition interval from the maximum range.

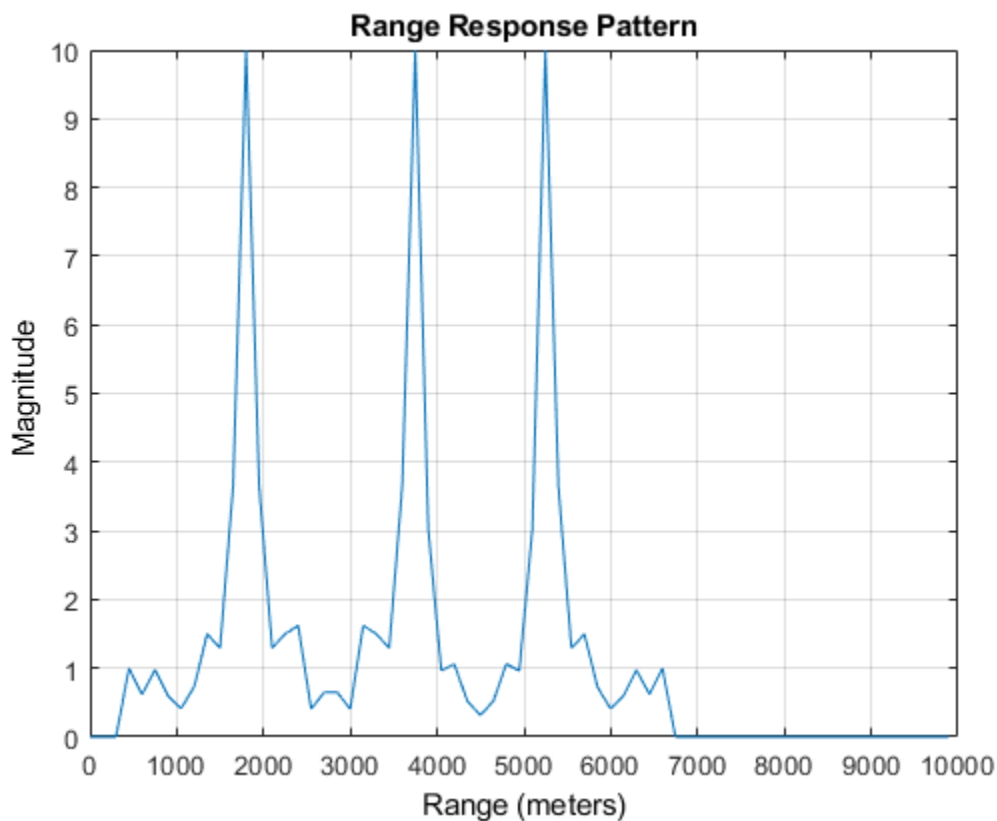
```
% Create the pulse waveform.
rmax = 10.0e3;
c = physconst('Lightspeed');
pri = 2*rmax/c;
fs = 1e6;
pri = ceil(pri*fs)/fs;
prf = 1/pri;
nsamp = pri*fs;
rxdata = zeros(nsamp,1);
t1 = 2*2000/c;
t2 = 2*4000/c;
t3 = 2*5500/c;
idx1 = floor(t1*fs);
idx2 = floor(t2*fs);
idx3 = floor(t3*fs);
lfm = phased.LinearFMWaveform('PulseWidth',10/fs,'PRF',prf, ...
    'SweepBandwidth',(30*fs)/40);
w = lfm();
%%
```

```

% Imbed the waveform part of the pulse into the received signal.
x = w(1:11);
rxdata(idx1:idx1+10) = x;
rxdata(idx2:idx2+10) = x;
rxdata(idx3:idx3+10) = x;

%%
% Create the pulse waveform library.
w1 = {'LinearFM','PulseWidth',10/fs,'PRF',prf,...
      'SweepBandwidth',(30*fs)/40};
wavlib = pulseWaveformLibrary('SampleRate',fs,'WaveformSpecification',{w1});
wav = wavlib(1);
%%
% Generate the range response signal.
p1 = {'MatchedFilter','Coefficients',getMatchedFilter(wavlib,1),'SpectrumWindow','None'};
idx = 1;
complib = pulseCompressionLibrary( ...
      'WaveformSpecification',{w1}, ...
      'ProcessingSpecification',{p1}, ...
      'SampleRate',fs, ...
      'PropagationSpeed',c);
y = complib(rxdata,1);
%%
% Plot range response of processed data
plotResponse(complib,rxdata,idx,'Unit','mag');

```



## More About

### Pulse Repetition Frequency Restrictions

The PRF property must satisfy these restrictions:

- The product of PRF and PulseWidth must be less than or equal to one. This condition expresses the requirement that the pulse width is less than one pulse repetition interval.
- The ratio of SampleRate to PRF must be an integer. This condition expresses the requirement that the number of samples in one pulse repetition interval is an integer.

### Chip Restrictions

The values of the ChipWidth and NumChips properties must satisfy these constraints:

- The product of PRF, ChipWidth, and NumChips must be less than or equal to one. This condition expresses the requirement that the sum of the durations of all chips is less than one pulse repetition interval.
- The product of SampleRate and ChipWidth must be an integer. This condition expresses the requirement that the number of samples in a chip must be an integer.

The table shows additional constraints on the number of chips for different code types.

If the Code Property Is ...	Then the NumChips Property Must Be...
'Frank', 'P1', or 'Px'	A perfect square.
'P2'	An even number that is a perfect square.
'Barker'	2, 3, 4, 5, 7, 11, or 13

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The plotResponse object function is not supported for code generation.

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Apps

**Pulse Waveform Analyzer**

### Objects

phased.LinearFMWaveform | phased.RectangularWaveform | phased.PhaseCodedWaveform  
| phased.SteppedFMWaveform | pulseWaveformLibrary | phased.RangeResponse |  
phased.RangeDopplerResponse | phased.MatchedFilter | phased.StretchProcessor

**Introduced in R2021a**

## plotResponse

Plot range response from pulse compression library

### Syntax

```
plotResponse(complib,X,idx)
plotResponse( ____,pulseidx)
plotResponse( ____, 'Unit',unit)
```

### Description

`plotResponse(complib,X,idx)` plots the range response of the input waveform, `X`, using the `idx` processing specification.

`plotResponse( ____, pulseidx)` also specifies the index, `pulseidx`, of the pulse to plot.

`plotResponse( ____, 'Unit', unit)` plots the response in the units specified by `unit`.

### Examples

#### Range Response for Three Targets

Plot the range response of an LFM signal hitting three targets at ranges of 2000, 4000, and 5500 meters. Assuming the maximum range of the radar is 10 km, determine the pulse repetition interval from the maximum range.

```
% Create the pulse waveform.
rmax = 10.0e3;
c = physconst('Lightspeed');
pri = 2*rmax/c;
fs = 1e6;
pri = ceil(pri*fs)/fs;
prf = 1/pri;
nsamp = pri*fs;
rxdata = zeros(nsamp,1);
t1 = 2*2000/c;
t2 = 2*4000/c;
t3 = 2*5500/c;
idx1 = floor(t1*fs);
idx2 = floor(t2*fs);
idx3 = floor(t3*fs);
lfm = phased.LinearFMWaveform('PulseWidth',10/fs,'PRF',prf, ...
    'SweepBandwidth',(30*fs)/40);
w = lfm();
%%
% Imbed the waveform part of the pulse into the received signal.
x = w(1:11);
rxdata(idx1:idx1+10) = x;
rxdata(idx2:idx2+10) = x;
rxdata(idx3:idx3+10) = x;
```

```

%%
% Create the pulse waveform library.
w1 = {'LinearFM','PulseWidth',10/fs,'PRF',prf,...
      'SweepBandwidth',(30*fs)/40};
wavlib = pulseWaveformLibrary('SampleRate',fs,'WaveformSpecification',{w1});
wav = wavlib(1);
%%
% Generate the range response signal.
p1 = {'MatchedFilter','Coefficients',getMatchedFilter(wavlib,1),'SpectrumWindow','None'};
idx = 1;
complib = pulseCompressionLibrary( ...
      'WaveformSpecification',{w1}, ...
      'ProcessingSpecification',{p1}, ...
      'SampleRate',fs, ...
      'PropagationSpeed',c);
y = complib(rxdata,1);
%%
% Plot range response of processed data
plotResponse(complib,rxdata,idx,'Unit','mag');

```



## Input Arguments

### **complib** — Pulse compression library

phased.PulseCompressionLibrary System object

Pulse compression library, specified as a `phased.PulseCompressionLibrary` System object .

**X – Input signal**

complex-valued  $K$ -by- $L$  matrix | complex-valued  $K$ -by- $N$  matrix | complex-valued  $K$ -by- $N$ -by- $L$  array

Input signal, specified as a complex-valued  $K$ -by- $L$  matrix, complex-valued  $K$ -by- $N$  matrix, or a complex-valued  $K$ -by- $N$ -by- $L$  array.  $K$  denotes the number of fast time samples,  $L$  the number of pulses, and  $N$  is the number of channels. Channels can be array elements or beams.

Data Types: `double`

**idx – Index of processing specification in pulse compression library**

positive integer

Index of processing specification in the pulse waveform library, specified as a positive integer.

Example: 3

Data Types: `double`

**pulseidx – Stepped FM waveform subpulse**

1 (default) | positive integer

Stepped FM waveform subpulse, specified as a positive integer. This index selects which subpulses of a stepped-FM waveform to plot. This argument only applies to stepped-FM waveforms.

Example: 5

Data Types: `double`

**unit – Plot units**

'db' (default) | 'mag' | 'pow'

Plot units, specified as 'db', 'mag', or 'pow'. who

- 'db' - plot the response power in dB.
- 'mag' - plot the magnitude of the response.
- 'pow' - plot the response power.

Example: 'mag'

Data Types: `char` | `string`

**Introduced in R2018b**



# pulseWaveformLibrary

Create library of pulse waveforms

## Description

The `pulseWaveformLibrary` System object creates a library of pulse waveforms. The waveforms in the library can be of different types or be of the same type with different parameters. You can use this library to transmit different kinds of pulses during a simulation.

To make a waveform library

- 1 Create the `pulseWaveformLibrary` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
pulseLib = pulseWaveformLibrary
pulseLib = pulseWaveformLibrary(Name,Value)
```

### Description

`pulseLib = pulseWaveformLibrary` System object creates a library of pulse waveforms, `pulseLib`, with default property values. The default consists of a rectangular waveform and a linear FM waveform.

`pulseLib = pulseWaveformLibrary(Name,Value)` creates a pulse waveform library with each property `Name` set to a specified `Value`. You can specify additional name-value pair arguments in any order as (`Name1,Value1,...,NameN,ValueN`). Enclose each property name in single quotes.

Example: `pulseLib = pulseWaveformLibrary('SampleRate',1e9,'WaveformSpecification',{{'Rectangular','PRF',1e4,'PulseWidth',100e-6},{'SteppedFM','PRF',1e4}})` creates a library containing one rectangular waveform and one stepped-FM waveform, both sampled at 1 GHz.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

**SampleRate — Waveform sample rate**

1e6 (default) | positive scalar

Waveform sample rate, specified as a positive scalar. All waveforms have the same sample rate. Units are in hertz.

Example: 100e3

Data Types: double

**WaveformSpecification — Pulse waveforms**

```
{{'Rectangular','PRF',10e3,'PulseWidth',100e-6},
{'LinearFM','PRF',1e4,'PulseWidth',50e-6,'SweepBandwidth',1e5,'SweepDirection',
'Up','SweepInterval','Positive'}} (default) | cell array
```

Pulse waveforms, specified as a cell array. Each cell of the array contains the specification of one waveform.

```
{{Waveform 1 Specification},{Waveform 2 Specification},{Waveform 3 Specification}, ...}
```

Each waveform specification is also a cell array containing the parameters of the waveform. The entries in a specification cell are the pulse identifier and a set of name-value pairs specific to that waveform.

```
{PulseIdentifier,Name1,Value1,Name2,Value2, ...}
```

This System object supports four built-in waveforms and also lets you specify custom waveforms. For the built-in waveforms, the waveform specifier consists of a waveform identifier followed by several name-value pairs setting the properties of the waveform. For the custom waveforms, the waveform specifier consists of a handle to a user-define waveform function and the functions input arguments.

**Waveform Types**

Waveform type	Waveform identifier	Waveform arguments
Linear FM	'LinearFM'	"Linear FM Waveform Arguments" on page 4-394
Phase coded	'PhaseCoded'	"Phase-Coded Waveform Arguments" on page 4-396
Rectangular	'Rectangular'	"Rectangular Waveform Arguments" on page 4-397
Stepped FM	'SteppedFM'	"Stepped FM Waveform Arguments" on page 4-398
Custom	<i>Function handle</i>	"Custom Waveform Arguments" on page 4-399

```
Example: {{'Rectangular','PRF',10e3,'PulseWidth',100e-6},
{'Rectangular','PRF',100e3,'PulseWidth',20e-6}}
```

Data Types: cell

**Linear FM Waveform Arguments**

Specify optional pairs of arguments as Name1=Value1, ..., NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

```
Example: {'LinearFM', 'PRF', 1e4, 'PulseWidth', 50e-6, 'SweepBandwidth', 1e5, ...
'SweepDirection', 'Up', 'SweepInterval', 'Positive'}
```

### **PRF — Pulse repetition frequency**

1e4 (default) | positive scalar

Pulse repetition frequency (PRF), specified as a positive scalar. Units are in hertz. See “Pulse Repetition Frequency Restrictions” on page 4-406 for restrictions on the PRF.

Example: 20e3

Data Types: double

### **PulseWidth — Pulse duration**

5e-5 (default) | positive scalar

Pulse duration, specified as a positive scalar. Units are in seconds. You cannot specify both `PulseWidth` and `DutyCycle`.

Example: 100e-6

Data Types: double

### **DutyCycle — Pulse duty cycle**

0.5 | positive scalar

Pulse duty cycle, specified as a positive scalar greater than zero and less than or equal to one. You cannot specify both `PulseWidth` and `DutyCycle`.

Example: 0.7

Data Types: double

### **SweepBandwidth — Bandwidth of the FM sweep**

1e5 (default) | positive scalar

Bandwidth of the FM sweep, specified as a positive scalar. Units are in hertz.

Example: 100e3

Data Types: double

### **SweepDirection — Bandwidth of the FM sweep**

'Up' (default) | 'Down'

Direction of the FM sweep, specified as 'Up' or 'Down'. 'Up' corresponds to increasing frequency. 'Down' corresponds to decreasing frequency.

Data Types: char

### **SweepInterval — FM sweep interval**

'Positive' (default) | 'Symmetric'

FM sweep interval, specified as 'Positive' or 'Symmetric'. If you set this property value to 'Positive', the waveform sweeps the interval between 0 and  $B$ , where  $B$  is the `SweepBandwidth` argument value. If you set this property value to 'Symmetric', the waveform sweeps the interval between  $-B/2$  and  $B/2$ .

Example: 'Symmetric'

Data Types: char

### **Envelope — Envelope function**

'Rectangular' (default) | 'Gaussian'

Envelope function, specified as 'Rectangular' or 'Gaussian'.

Example: 'Gaussian'

Data Types: char

### **FrequencyOffset — Frequency offset of pulse**

0 (default) | scalar

Frequency offset of pulse, specified as a scalar. The frequency offset shifts the frequency of the generated pulse waveform. Units are in hertz.

Example: 100e3

Data Types: double

### **Phase-Coded Waveform Arguments**

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: {'PhaseCoded', 'PRF', 1e4, 'Code', 'Zadoff-Chu',  
'SequenceIndex', 3, 'ChipWidth', 5e-6, 'NumChips', 8}

### **PRF — Pulse repetition frequency**

1e4 (default) | positive scalar

Pulse repetition frequency (PRF), specified as a positive scalar. Units are in hertz. See “Pulse Repetition Frequency Restrictions” on page 4-406 for restrictions on the PRF.

Example: 20e3

Data Types: double

### **Code — Type of phase modulation code**

'Frank' (default) | 'P1' | 'P2' | 'Px' | 'Zadoff-Chu' | 'P3' | 'P4' | 'Barker'

Type of phase modulation code, specified as 'Frank', 'P1', 'P2', 'Px', 'Zadoff-Chu', 'P3', 'P4', or 'Barker'.

Example: 'P1'

Data Types: char

### **SequenceIndex — Zadoff-Chu sequence index**

1 (default) | positive integer

Sequence index used for the Zadoff-Chu code, specified as a positive integer. The value of SequenceIndex must be relatively prime to the value of NumChips.

Example: 3

### Dependencies

To enable this name-value pair, set the Code property to 'Zadoff-Chu'.

Data Types: double

### ChipWidth — Chip duration

1e-5 (default) | positive scalar

Chip duration, specified as a positive scalar. Units are in seconds. See “Chip Restrictions” on page 4-407 for restrictions on chip sizes.

Example: 30e-3

Data Types: double

### NumChips — Number of chips in waveform

4 (default) | positive integer

Number of chips in waveform, specified as a positive integer. See “Chip Restrictions” on page 4-407 for restrictions on chip sizes.

Example: 3

Data Types: double

### FrequencyOffset — Frequency offset of pulse

0 (default) | scalar

Frequency offset of pulse, specified as a scalar. The frequency offset shifts the frequency of the generated pulse waveform. Units are in hertz.

Example: 100e3

Data Types: double

### Rectangular Waveform Arguments

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: {'Rectangular', 'PRF', 10e3, 'PulseWidth', 100e-6}

### PRF — Pulse repetition frequency

1e4 (default) | positive scalar

Pulse repetition frequency (PRF), specified as a positive scalar. Units are in hertz. See “Pulse Repetition Frequency Restrictions” on page 4-406 for restrictions on the PRF.

Example: 20e3

Data Types: double

### PulseWidth — Pulse duration

5e-5 (default) | positive scalar

Pulse duration, specified as a positive scalar. Units are in seconds. You cannot specify both `PulseWidth` and `DutyCycle`.

Example: `100e-6`

Data Types: `double`

### **DutyCycle — Pulse duty cycle**

`0.5` | positive scalar

Pulse duty cycle, specified as a positive scalar greater than zero and less than or equal to one. You cannot specify both `PulseWidth` and `DutyCycle`.

Example: `0.7`

Data Types: `double`

### **FrequencyOffset — Frequency offset of pulse**

`0` (default) | scalar

Frequency offset of pulse, specified as a scalar. The frequency offset shifts the frequency of the generated pulse waveform. Units are in hertz.

Example: `100e3`

Data Types: `double`

### **Stepped FM Waveform Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `{'SteppedFM', 'PRF', 10e-4}`

### **PRF — Pulse repetition frequency**

`1e4` (default) | positive scalar

Pulse repetition frequency (PRF), specified as a positive scalar. Units are in hertz. See “Pulse Repetition Frequency Restrictions” on page 4-406 for restrictions on the PRF.

Example: `20e3`

Data Types: `double`

### **PulseWidth — Pulse duration**

`5e-5` (default) | positive scalar

Pulse duration, specified as a positive scalar. Units are in seconds. You cannot specify both `PulseWidth` and `DutyCycle`.

Example: `100e-6`

Data Types: `double`

### **DutyCycle — Pulse duty cycle**

`0.5` | positive scalar

Pulse duty cycle, specified as a positive scalar greater than zero and less than or equal to one. You cannot specify both `PulseWidth` and `DutyCycle`.

Example: 0.7

Data Types: double

### **NumSteps — Number of frequency steps in waveform**

5 (default) | positive integer

Number of frequency steps in waveform, specified as a positive integer.

Example: 3

Data Types: double

### **FrequencyStep — Linear frequency step size**

20e3 (default) | positive scalar

Linear frequency step size, specified as a positive scalar.

Example: 100.0

Data Types: double

### **FrequencyOffset — Frequency offset of pulse**

0 (default) | scalar

Frequency offset of pulse, specified as a scalar. The frequency offset shifts the frequency of the generated pulse waveform. Units are in hertz.

Example: 100e3

Data Types: double

## **Custom Waveform Arguments**

You can create a custom waveform from a user-defined function. The first input argument of the function must be the sample rate. For example, specify a hyperbolic waveform function,

```
function wav = HyperbolicFM(fs,prf,pw,freq,bw,fcent),
```

where `fs` is the sample rate and `prf`, `pw`, `freq`, `bw`, and `fcent` are other waveform arguments. The function must have at least one output argument, `wav`, to return the samples of each pulse. This output must be a column vector. There can be other outputs returned following the waveform samples.

Then, create a waveform specification using a function handle instead of the waveform identifier. The first cell in the waveform specification must be a function handle. The remaining cells contain all function input arguments except the sample rate. Specify all input arguments in the order they are passed into the function.

```
waveformspec = {@HyperbolicFM,prf,pw,freq,bw,fcent}
```

See “Add Custom Waveform to Pulse Waveform Library” on page 4-405 for an example that uses a custom waveform.

## Usage

### Syntax

```
waveform = pulselib(idx)
```

### Description

`waveform = pulselib(idx)` returns samples of a waveform, `waveform`, specified by its index, `idx`, in the library.

### Input Arguments

#### **idx** — Index of the waveform in the waveform library

positive integer

Index of the waveform in the waveform library, specified as a positive integer.

Example: 2

Data Types: `double`

### Output Arguments

#### **waveform** — Waveform samples

complex-valued vector

Waveform samples, returned as a complex-valued vector.

Data Types: `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `pulseWaveformLibrary`

`getMatchedFilter` Matched filter coefficients for pulse waveform  
`plot` Plot waveform from waveform library

### Common to All System Objects

`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## Examples



## Obtain and Plot Phase-Coded Waveform from Waveform Library

Construct a waveform library consisting of three waveforms. The library contains a rectangular, a linear FM, and a phase-coded waveform. Then, obtain and plot the real and imaginary parts of the phase-coded waveform.

```

waveform1 = {'Rectangular','PRF',1e4,'PulseWidth',50e-6};
waveform2 = {'LinearFM','PRF',1e4,'PulseWidth',50e-6, ...
    'SweepBandwidth',1e5,'SweepDirection','Up',...
    'SweepInterval','Positive'};
waveform3 = {'PhaseCoded','PRF',1e4,'Code','Zadoff-Chu', ...
    'SequenceIndex',3,'ChipWidth',5e-6,'NumChips',8};
fs = 1e6;
wavlib = pulseWaveformLibrary('SampleRate',fs, ...
    'WaveformSpecification',{waveform1,waveform2,waveform3});

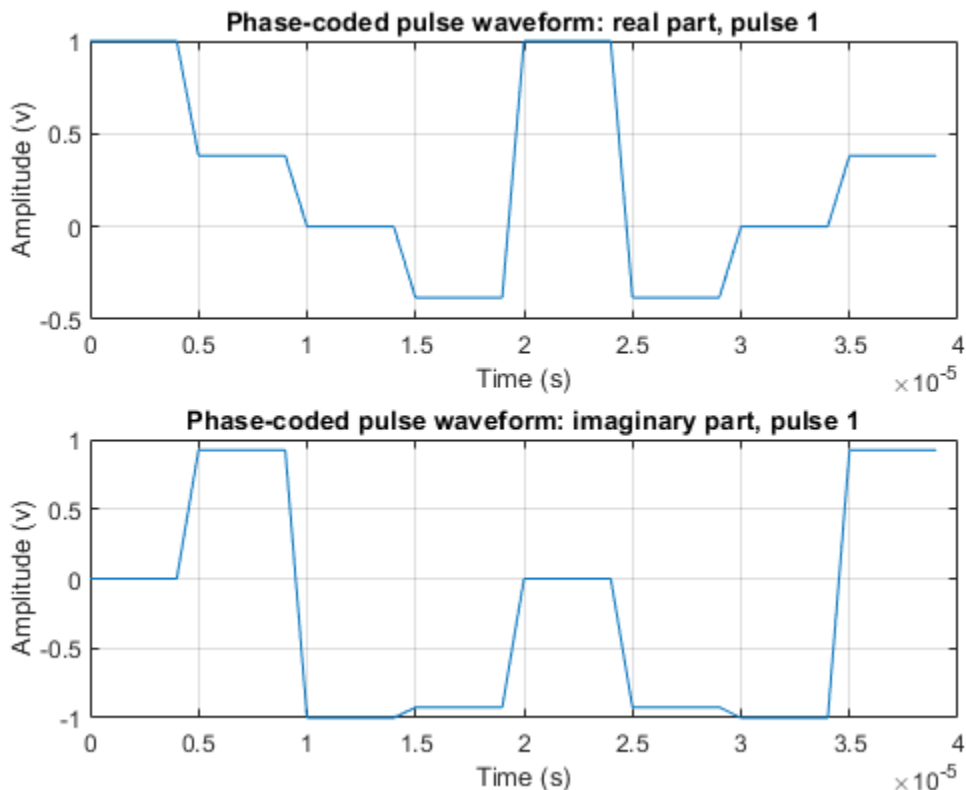
```

Extract the waveform from the library.

```
wav3 = wavlib(3);
```

Plot the waveform using the plot method.

```
plot(wavlib,3,'PlotType','complex')
```



### Plot Stepped FM Waveform

Construct a waveform library consisting of three waveforms. The library contains one rectangular, one linear FM, and one stepped-FM waveforms. Then, plot the real parts of the first three pulses of the stepped-fm waveform.

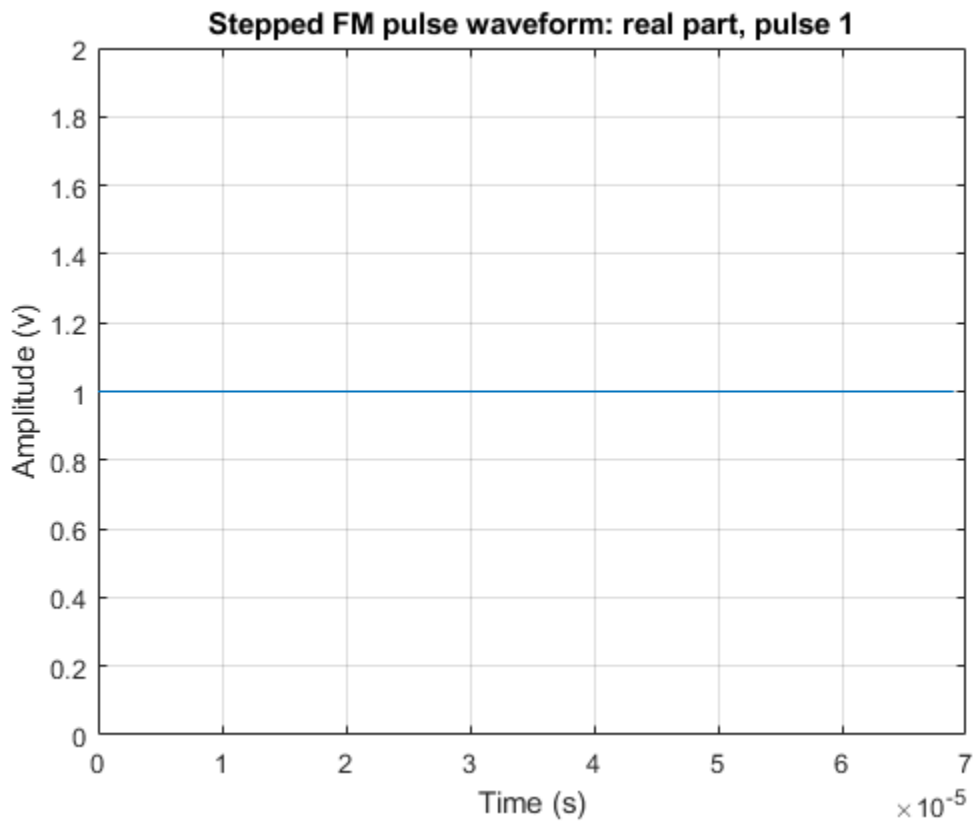
```

waveform1 = {'Rectangular', 'PRF', 1e4, 'PulseWidth', 70e-6};
waveform2 = {'LinearFM', 'PRF', 1e4, 'PulseWidth', 70e-6, ...
            'SweepBandwidth', 1e5, 'SweepDirection', 'Up', ...
            'SweepInterval', 'Positive'};
waveform3 = {'SteppedFM', 'PRF', 1e4, 'PulseWidth', 70e-6, 'NumSteps', 5, ...
            'FrequencyStep', 50000, 'FrequencyOffset', 0};
fs = 1e6;
wavlib = pulseWaveformLibrary('SampleRate', fs, ...
                              'WaveformSpecification', {waveform1, waveform2, waveform3});

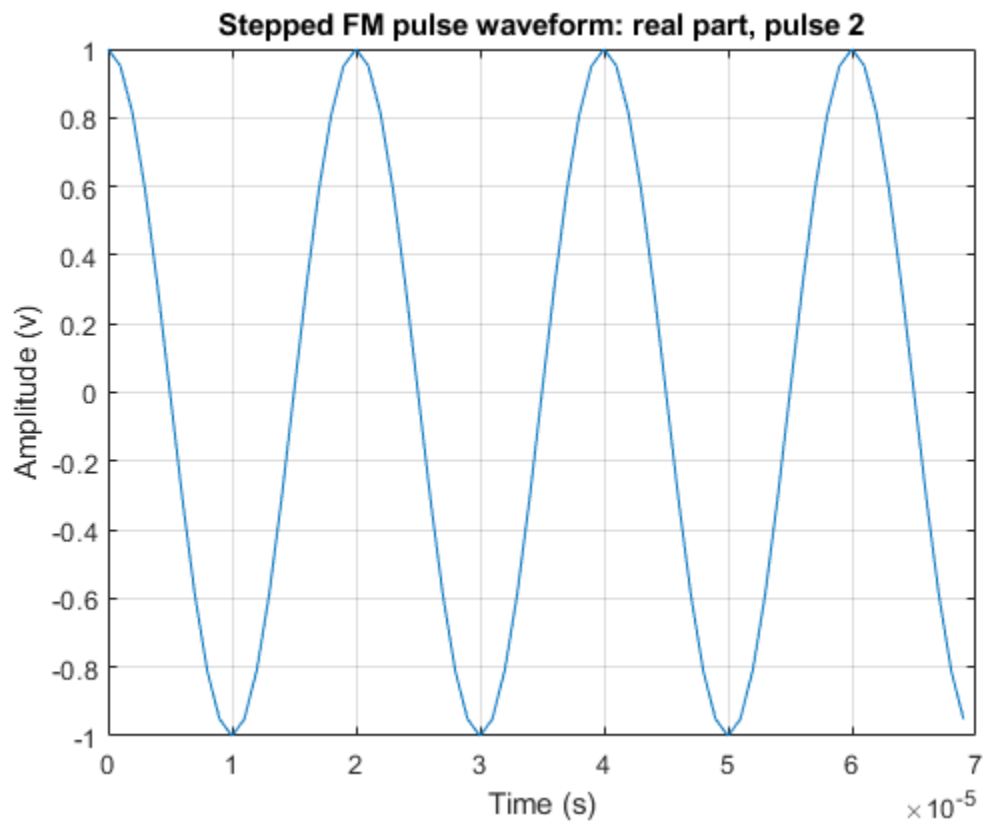
```

Plot the first three pulses of the waveform using the `plot` method.

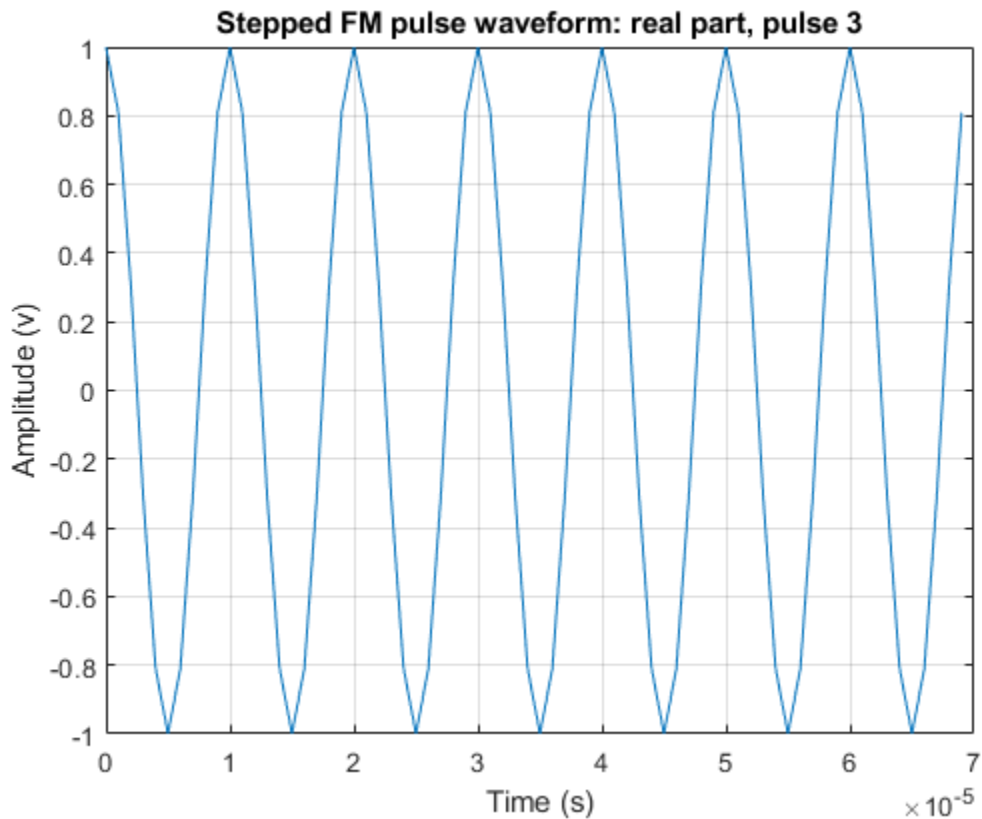
```
plot(wavlib, 3, 'PulseIdx', 1)
```



```
plot(wavlib, 3, 'PulseIdx', 2)
```



```
plot(wavlib,3,'PulseIdx',3)
```



### Plot Matched Filter Coefficients of Two Pulses

This example shows how to put two waveforms into a waveform library and how to extract and plot their matched filter coefficients.

Create a pulse library consisting of a rectangular and a linear FM waveform.

```

waveform1 = {'Rectangular', 'PRF', 10e3 'PulseWidth', 50e-6};
waveform2 = {'LinearFM', 'PRF', 10e3, 'PulseWidth', 50e-6, 'SweepBandwidth', 1e5, ...
    'SweepDirection', 'Up', 'SweepInterval', 'Positive'};
pulsesib = pulseWaveformLibrary('SampleRate', 1e6, ...
    'WaveformSpecification', {waveform1, waveform2});

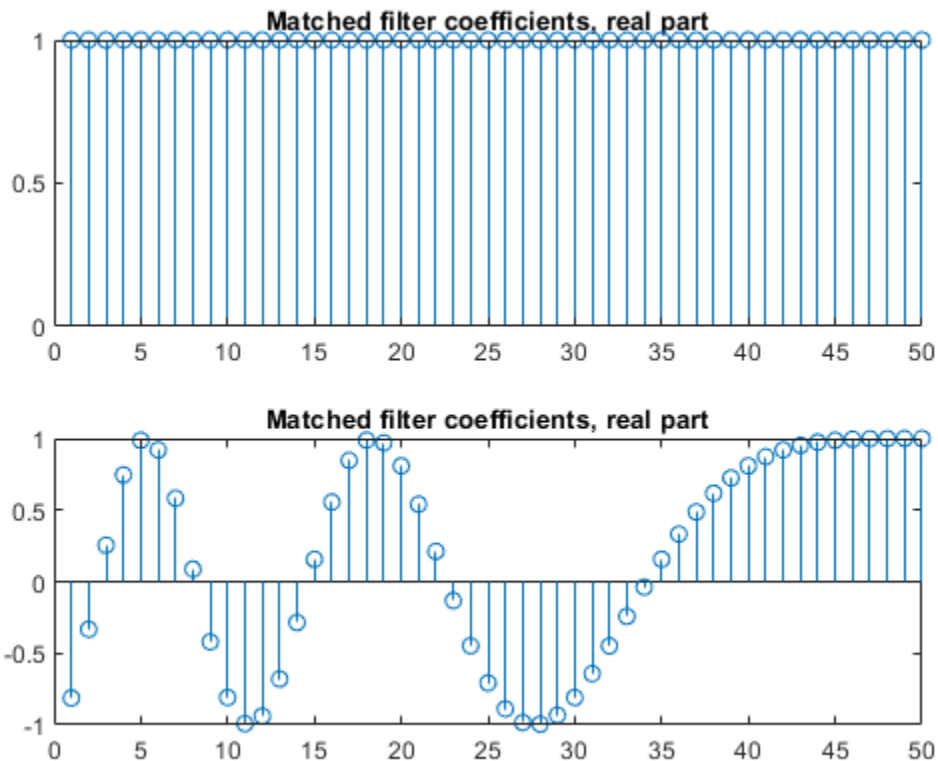
```

Retrieve the matched filter coefficients for each waveform and plot their real parts.

```

coeff1 = getMatchedFilter(pulsesib, 1, 1);
subplot(2, 1, 1)
stem(real(coeff1))
title('Matched filter coefficients, real part')
coeff2 = getMatchedFilter(pulsesib, 2, 1);
subplot(2, 1, 2)
stem(real(coeff2))
title('Matched filter coefficients, real part')

```



### Add Custom Waveform to Pulse Waveform Library

Define a custom hyperbolic FM waveform and add it to a `pulseWaveformLibrary` System object together with a linear FM waveform. Plot the hyperbolic waveform.

Specify the hyperbolic FM waveform parameters. The pulse width is 75 ms and the pulse repetition interval is 100 ms. The center frequency is 500 Hz and the bandwidth is 400 Hz.

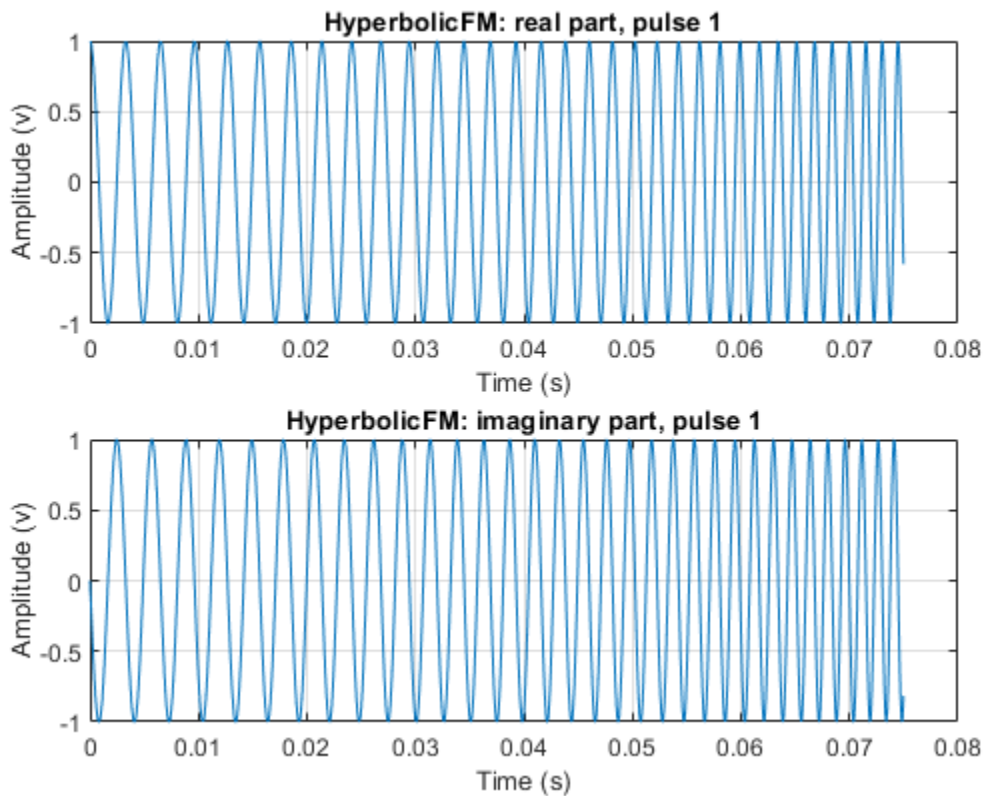
```
fs = 50e3;
pri = 0.1;
prf = 1/pri;
pw = 0.075;
bw = 400.0;
fcent = 500.0;
```

Create a pulse waveform library consisting of a hyperbolic FM waveform and a linear FM waveform.

```
pulselib = pulseWaveformLibrary('SampleRate',fs, ...
    'WaveformSpecification',{@HyperbolicFM,prf,pw,bw,fcent}, ...
    {'LinearFM','PRF',prf,'PulseWidth',pw, ...
    'SweepBandwidth',bw,'SweepDirection','Up',...
    'SweepInterval','Positive'});
```

Plot the complex hyperbolic FM waveform.

```
plot(pulselib,1,'PlotType','complex')
```



Define the Hyperbolic FM waveform function.

```
function y = HyperbolicFM(fs,prf,pw,bw,fcent)
pri = 1/prf;
t = [0:1/fs:pri]';
idx = find(t <= pw);
fl = fcent - bw/2;
fh = fcent + bw/2;
y = zeros(size(t));
arg = 2*pi*fl*fh/bw*pw*log(1.0 - bw*t(idx)/fh/pw);
y(idx) = exp(1i*arg);
end
```

## More About

### Pulse Repetition Frequency Restrictions

The PRF property must satisfy these restrictions:

- The product of PRF and PulseWidth must be less than or equal to one. This condition expresses the requirement that the pulse width is less than one pulse repetition interval.
- The ratio of SampleRate to PRF must be an integer. This condition expresses the requirement that the number of samples in one pulse repetition interval is an integer.

## Chip Restrictions

The values of the `ChipWidth` and `NumChips` properties must satisfy these constraints:

- The product of `PRF`, `ChipWidth`, and `NumChips` must be less than or equal to one. This condition expresses the requirement that the sum of the durations of all chips is less than one pulse repetition interval.
- The product of `SampleRate` and `ChipWidth` must be an integer. This condition expresses the requirement that the number of samples in a chip must be an integer.

The table shows additional constraints on the number of chips for different code types.

If the Code Property Is ...	Then the NumChips Property Must Be...
'Frank', 'P1', or 'Px'	A perfect square
'P2'	An even number that is a perfect square
'Barker'	2, 3, 4, 5, 7, 11, or 13

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

The `plot` object function is not supported.

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Apps

**Pulse Waveform Analyzer**

### Objects

`phased.LinearFMWaveform` | `phased.RectangularWaveform` | `phased.PhaseCodedWaveform`  
| `phased.SteppedFMWaveform` | `pulseCompressionLibrary`

**Introduced in R2021a**

## getMatchedFilter

Matched filter coefficients for pulse waveform

### Syntax

```
coeff = getMatchedFilter(pulselib,idx)
coeff = getMatchedFilter(pulselib,idx,pidx)
```

### Description

`coeff = getMatchedFilter(pulselib,idx)` returns matched filter coefficients, `coeff`, for the waveform specified by the index, `idx`, in the waveform library, `pulselib`.

`coeff = getMatchedFilter(pulselib,idx,pidx)` also specifies the pulse index, `pidx`, of a stepped FM waveform.

### Examples

#### Plot Matched Filter Coefficients of Two Pulses

This example shows how to put two waveforms into a waveform library and how to extract and plot their matched filter coefficients.

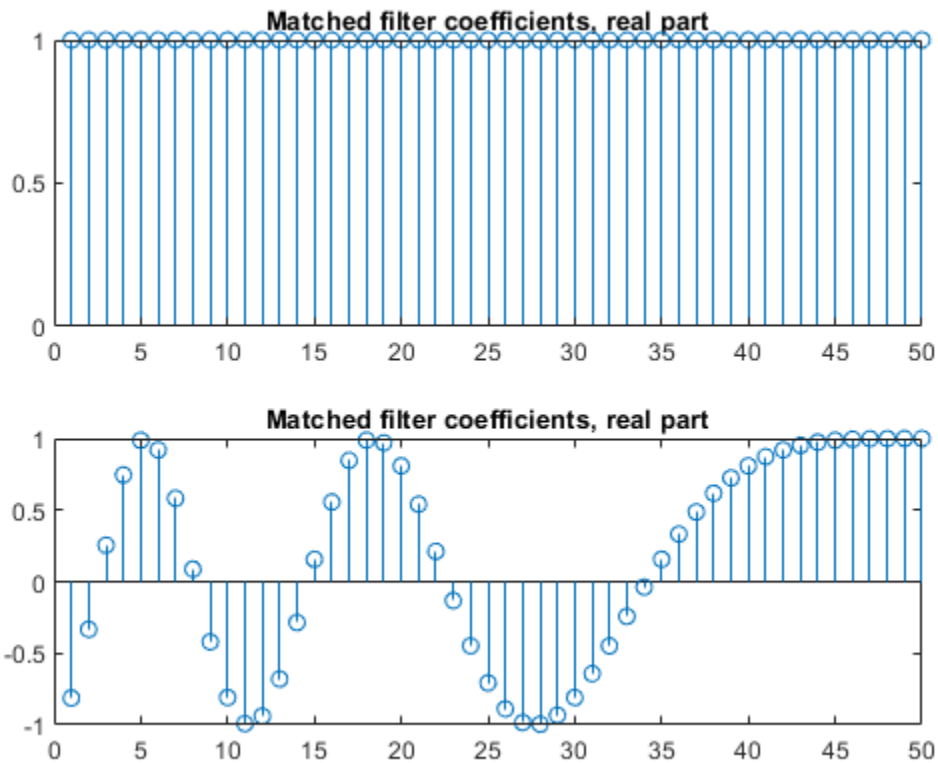
Create a pulse library consisting of a rectangular and a linear FM waveform.

```
waveform1 = {'Rectangular','PRF',10e3,'PulseWidth',50e-6};
waveform2 = {'LinearFM','PRF',10e3,'PulseWidth',50e-6,'SweepBandwidth',1e5, ...
            'SweepDirection','Up','SweepInterval','Positive'};
pulsesib = pulseWaveformLibrary('SampleRate',1e6,...
                                'WaveformSpecification',{waveform1,waveform2});
```

Retrieve the matched filter coefficients for each waveform and plot their real parts.

```
coeff1 = getMatchedFilter(pulsesib,1,1);
subplot(2,1,1)
stem(real(coeff1))
title('Matched filter coefficients, real part')
coeff2 = getMatchedFilter(pulsesib,2,1);
subplot(2,1,2)
stem(real(coeff2))
title('Matched filter coefficients, real part')
```





## Input Arguments

### **pulseLib** — Waveform library

phased.PulseWaveformLibrary System object

Pulse waveform library, specified as a `phased.PulseWaveformLibrary` System object.

### **idx** — Waveform index

1 (default) | positive integer

Waveform index, specified as a positive integer. The index specifies which waveform coefficients to return.

Data Types: double

### **pid** — Pulse index

1 (default) | positive integer

Pulse index, specified as a positive integer. The index specifies which pulse matched-filter coefficients to return. This argument applies only to stepped FM waveforms.

Data Types: double

## Output Arguments

### **coeff** — Matched filter coefficients

complex-valued vector | complex-valued matrix

Matched filter coefficients, specified as a complex-valued vector or complex-valued matrix. For the stepped FM pulse, the output is a complex-valued matrix. Each matrix column corresponds to a step in the waveform. For all other waveforms, the output is a column vector.

Data Types: `double`

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

**Introduced in R2021a**

# plot

Plot waveform from waveform library

## Syntax

```
plot(pulseLib,idx)
plot(pulseLib,idx,'PlotType',Type)
plot(____,'PulseIdx',pidX)
plot(____,LineStyle)
hdl = plot(____)
```

## Description

`plot(pulseLib,idx)` plots the real part of the waveform specified by `idx` belonging to the pulse waveform library, `pulseLib`.

`plot(pulseLib,idx,'PlotType',Type)` also specifies whether to plot the real and/or imaginary parts of the waveform using the ('PlotType',Type) name-value pair argument.

`plot(____,'PulseIdx',pidX)` also specifies the index, `pidX`, of the pulse to plot using the ('PulseIdx',pidX) name-value pair argument.

`plot(____,LineStyle)` specifies the line color, line style, or marker options. These options are the same options found in the MATLAB `plot` function. When both real and imaginary plots are specified, the `LineStyle` applies to both subplots. This argument is always the last input to the method.

`hdl = plot(____)` returns the line handle, `hdl`, in the figure.

## Examples

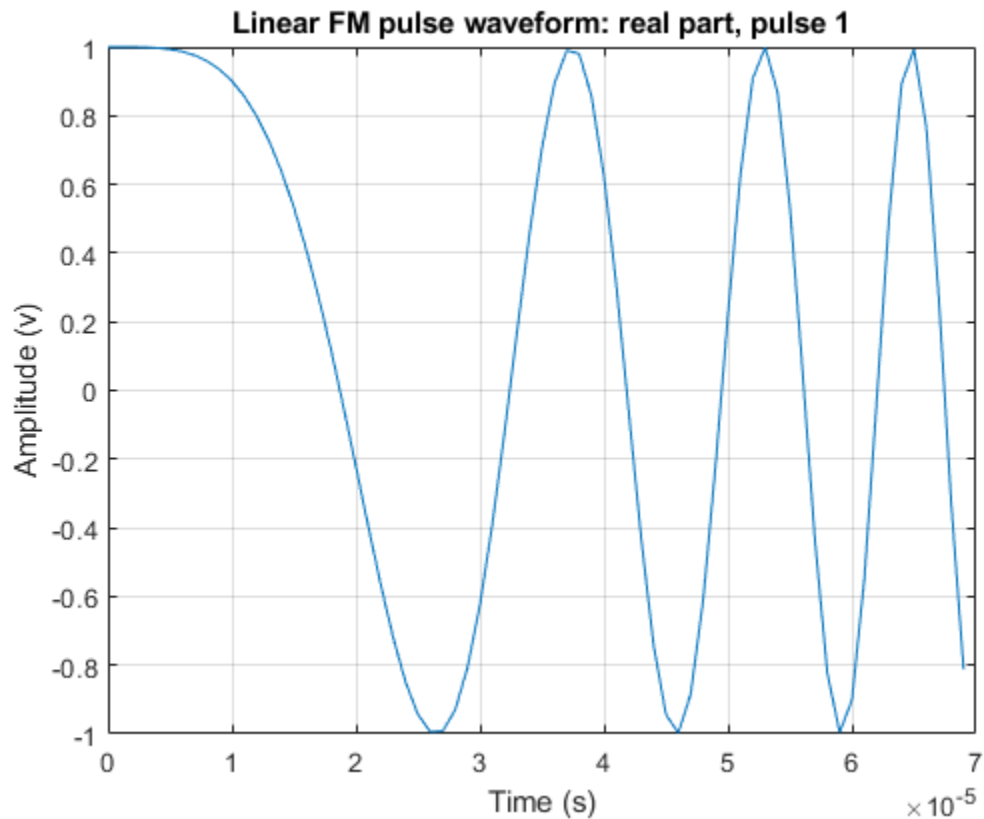
### Plot Linear FM Waveform

Construct a waveform library consisting of three waveforms. The library contains one rectangular waveform, one linear FM waveform, and one stepped-FM waveform.

```
waveform1 = {'Rectangular','PRF',1e4,'PulseWidth',70e-6};
waveform2 = {'LinearFM','PRF',1e4,'PulseWidth',70e-6, ...
    'SweepBandwidth',1e5,'SweepDirection','Up', ...
    'SweepInterval','Positive'};
waveform3 = {'SteppedFM','PRF',1e4,'PulseWidth',70e-6,'NumSteps',5, ...
    'FrequencyStep',50000,'FrequencyOffset',0};
fs = 1e6;
wavlib = pulseWaveformLibrary('SampleRate',fs, ...
    'WaveformSpecification',{waveform1,waveform2,waveform3});
```

Plot the linear FM waveform using the `plot` method.

```
plot(wavlib,2)
```



### Obtain and Plot Phase-Coded Waveform from Waveform Library

Construct a waveform library consisting of three waveforms. The library contains a rectangular, a linear FM, and a phase-coded waveform. Then, obtain and plot the real and imaginary parts of the phase-coded waveform.

```

waveform1 = {'Rectangular', 'PRF', 1e4, 'PulseWidth', 50e-6};
waveform2 = {'LinearFM', 'PRF', 1e4, 'PulseWidth', 50e-6, ...
    'SweepBandwidth', 1e5, 'SweepDirection', 'Up', ...
    'SweepInterval', 'Positive'};
waveform3 = {'PhaseCoded', 'PRF', 1e4, 'Code', 'Zadoff-Chu', ...
    'SequenceIndex', 3, 'ChipWidth', 5e-6, 'NumChips', 8};
fs = 1e6;
wavlib = pulseWaveformLibrary('SampleRate', fs, ...
    'WaveformSpecification', {waveform1, waveform2, waveform3});

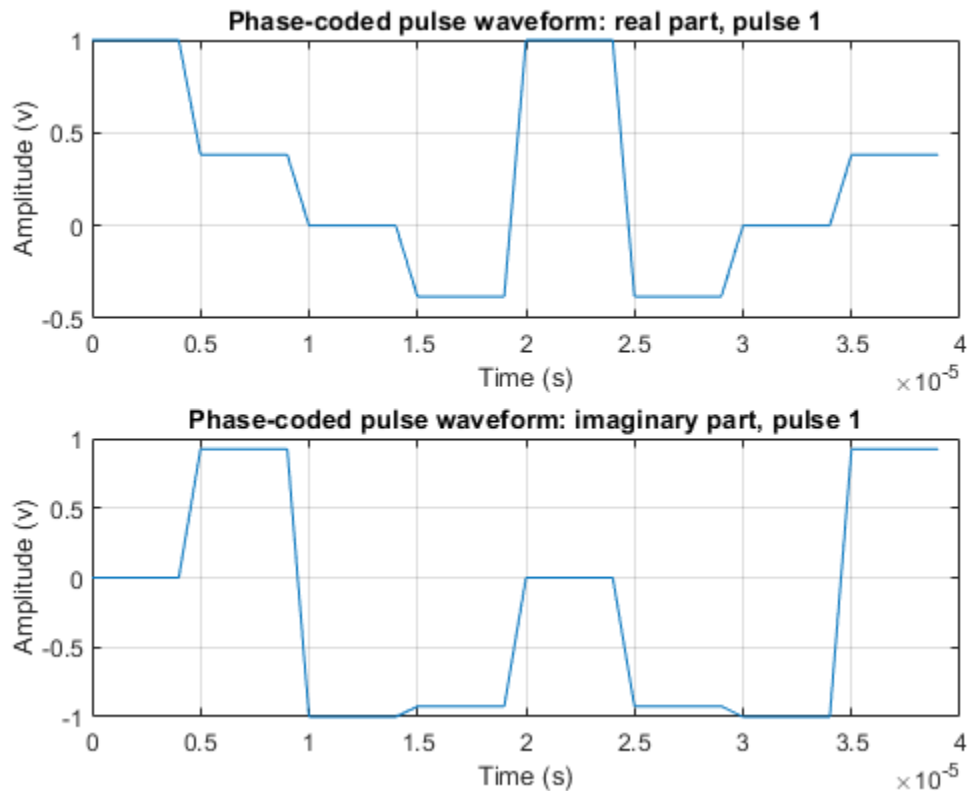
```

Extract the waveform from the library.

```
wav3 = wavlib(3);
```

Plot the waveform using the plot method.

```
plot(wavlib, 3, 'PlotType', 'complex')
```



### Plot Stepped FM Waveform

Construct a waveform library consisting of three waveforms. The library contains one rectangular, one linear FM, and one stepped-FM waveforms. Then, plot the real parts of the first three pulses of the stepped-fm waveform.

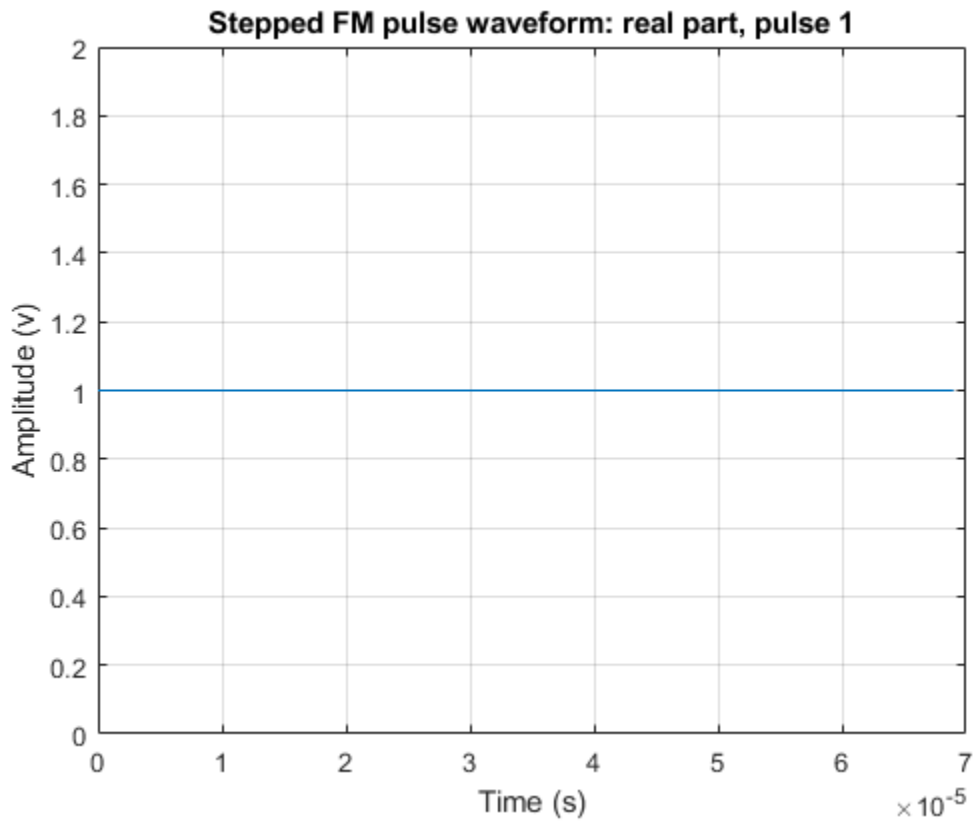
```

waveform1 = {'Rectangular', 'PRF', 1e4, 'PulseWidth', 70e-6};
waveform2 = {'LinearFM', 'PRF', 1e4, 'PulseWidth', 70e-6, ...
            'SweepBandwidth', 1e5, 'SweepDirection', 'Up', ...
            'SweepInterval', 'Positive'};
waveform3 = {'SteppedFM', 'PRF', 1e4, 'PulseWidth', 70e-6, 'NumSteps', 5, ...
            'FrequencyStep', 50000, 'FrequencyOffset', 0};
fs = 1e6;
wavlib = pulseWaveformLibrary('SampleRate', fs, ...
                              'WaveformSpecification', {waveform1, waveform2, waveform3});

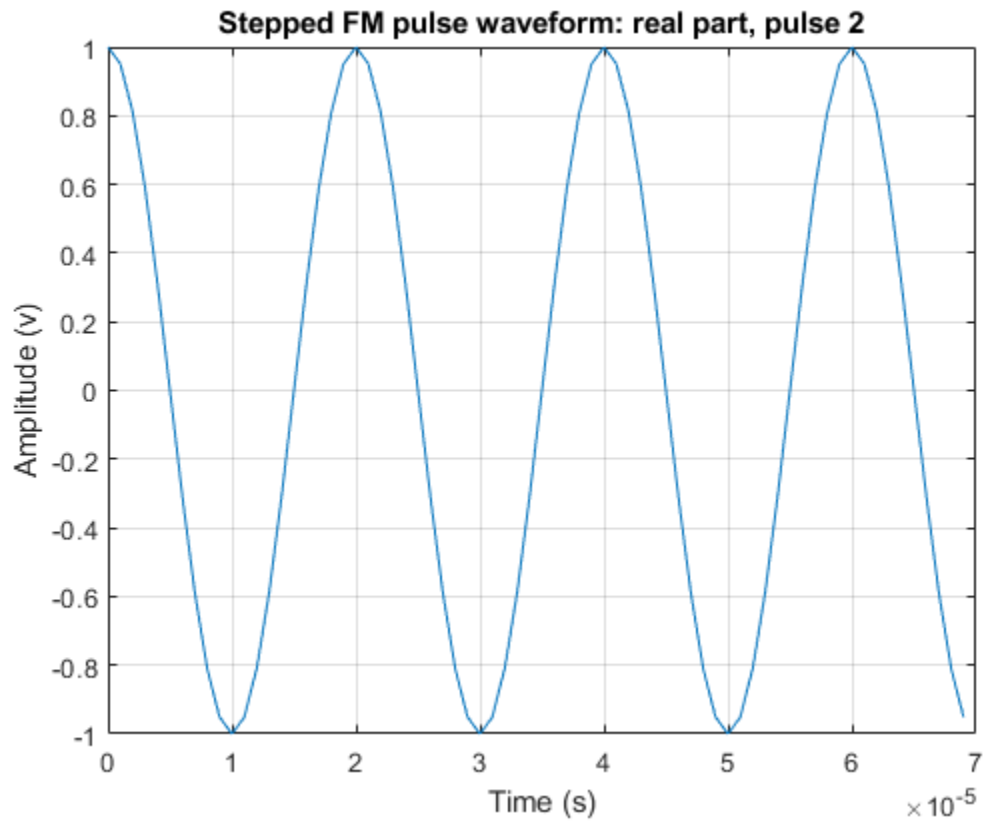
```

Plot the first three pulses of the waveform using the `plot` method.

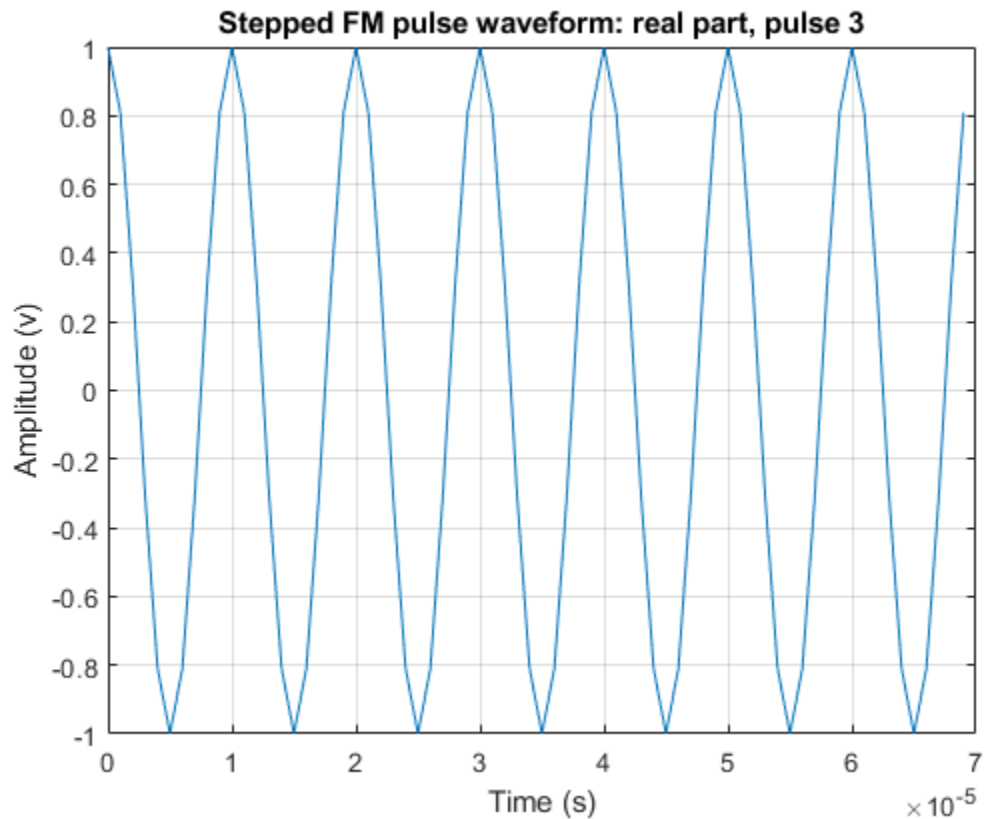
```
plot(wavlib, 3, 'PulseIdx', 1)
```



```
plot(wavlib,3, 'PulseIdx',2)
```



```
plot(wavlib,3, 'PulseIdx',3)
```



### Plot Linear FM Waveform With Dotted Lines

Construct a waveform library consisting of three waveforms. The library contains one rectangular, one linear FM, and one stepped-FM waveforms. Then, plot the linear FM waveform.

```

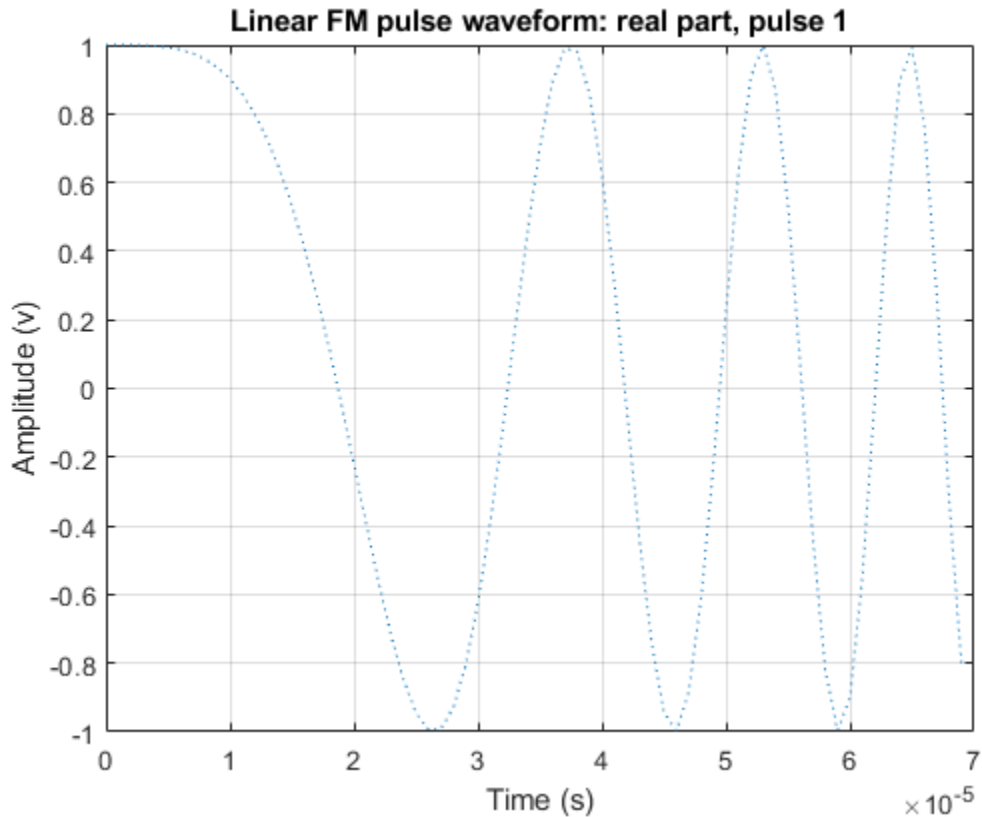
waveform1 = {'Rectangular', 'PRF', 1e4, 'PulseWidth', 70e-6};
waveform2 = {'LinearFM', 'PRF', 1e4, 'PulseWidth', 70e-6, ...
            'SweepBandwidth', 1e5, 'SweepDirection', 'Up', ...
            'SweepInterval', 'Positive'};
waveform3 = {'SteppedFM', 'PRF', 1e4, 'PulseWidth', 70e-6, 'NumSteps', 5, ...
            'FrequencyStep', 50000, 'FrequencyOffset', 0};
fs = 1e6;
wavlib = pulseWaveformLibrary('SampleRate', fs, ...
                              'WaveformSpecification', {waveform1, waveform2, waveform3});

```

Plot the waveform using the plot method.

```
plot(wavlib, 2, ':')
```





## Input Arguments

### **pulseLib** — Waveform library

pulseWaveformLibrary object System object

Waveform library, specified as a pulseWaveformLibrary System object.

### **idx** — Index of waveform in pulse waveform library

positive integer

Index of waveform in pulse waveform library, specified as a positive integer.

Example: 3

Data Types: double

### **Type** — Plot type

'real' (default) | 'imag' | 'complex'

Plot type, specified as 'real', 'imag', or 'complex'. Use this argument in the 'Type' name-value pair.

Data Types: char | string

### **pidx** — Index of plot to pulse

1 (default) | positive integer

Index of plot to pulse, specified as a positive integer. Use this argument in the 'PulseIdx' name-value pair. This argument only affects the stepped-FM waveform.

Data Types: `double`

### **LineStyle — Line color, style, and marker options**

'b' (default) | character vector

Line color, style, and marker options, specified as a character vector. These options are the same as for the MATLAB `plot` function. If you specify a `PlotType` value of 'complex', then `LineStyle` applies to both the real and imaginary subplots.

Example: 'ko'

Data Types: `char`

### **Name-Value Pair Arguments**

Example: 'PlotType','imag'

### **PlotType — Plot real or imaginary components of waveform**

'real' (default) | 'imag' | 'complex'

Components of waveform, specified as 'real', 'imag', or 'complex'.

Example: 'complex'

Data Types: `char`

### **PulseIdx — Plot stepped FM waveform subpulse**

1 (default) | positive integer

Plot stepped FM waveform subpulse, specified as a positive integer. This argument only affects the stepped-FM waveform.

Example: 5

Data Types: `double`

## **Output Arguments**

### **hdl — Handles of lines in figure**

scalar | 2-by-1 real-valued vector

Handle of lines in figure, returned as a scalar or 2-by-1 real-valued vector. For the case when both real and imaginary plots are specified, the vector includes handles to the lines in both subplots, in the form of `[RealLineHandle;ImagLineHandle]`.

## **See Also**

`plot`

**Introduced in R2021a**

# barrageJammer

Barrage jammer

## Description

The barrageJammer object implements a white Gaussian noise jammer.

To obtain the jamming signal:

- 1 Define and set up your barrage jammer. See “Construction” on page 4-419.
- 2 Call `step` to compute the jammer output according to the properties of `barrageJammer`. The behavior of `step` is specific to each object in the toolbox.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

## Construction

`H = barrageJammer` creates a barrage jammer System object, `H`. This object generates a complex white Gaussian noise jamming signal.

`H = barrageJammer(Name, Value)` creates object, `H`, with each specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as `(Name1, Value1, ..., NameN, ValueN)`.

`H = barrageJammer(E, Name, Value)` creates a barrage jammer object, `H`, with the `ERP` property set to `E` and other specified property `Names` set to the specified `Values`.

## Properties

### ERP

Effective radiated power

Specify the effective radiated power (ERP) (in watts) of the jamming signal as a positive scalar.

**Default:** 5000

### SamplesPerFrameSource

Source of number of samples per frame

Specify whether the number of samples of the jamming signal comes from the `SamplesPerFrame` property of this object or from an input argument in `step`. Values of this property are:

'Property'	The <code>SamplesPerFrame</code> property of this object specifies the number of samples of the jamming signal.
'Input port'	An input argument in each invocation of <code>step</code> specifies the number of samples of the jamming signal.

**Default:** 'Property'

### SamplesPerFrame

Number of samples per frame

Specify the number of samples in the output jamming signal as a positive integer. This property applies when you set the `SamplesPerFrameSource` property to 'Property'.

**Default:** 100

### SeedSource

Source of seed for random number generator

Specify how the object generates random numbers. Values of this property are:

'Auto'	The default MATLAB random number generator produces the random numbers. Use 'Auto' if you are using this object with Parallel Computing Toolbox™ software.
'Property'	The object uses its own private random number generator to produce random numbers. The <code>Seed</code> property of this object specifies the seed of the random number generator. Use 'Property' if you want repeatable results and are not using this object with Parallel Computing Toolbox software.

**Default:** 'Auto'

### Seed

Seed for random number generator

Specify the seed for the random number generator as a scalar integer between 0 and  $2^{32}-1$ . This property applies when you set the `SeedSource` property to 'Property'.

**Default:** 0

### Methods

- `reset`     Reset random number generator for noise generation
- `step`     Generate noise jamming signal

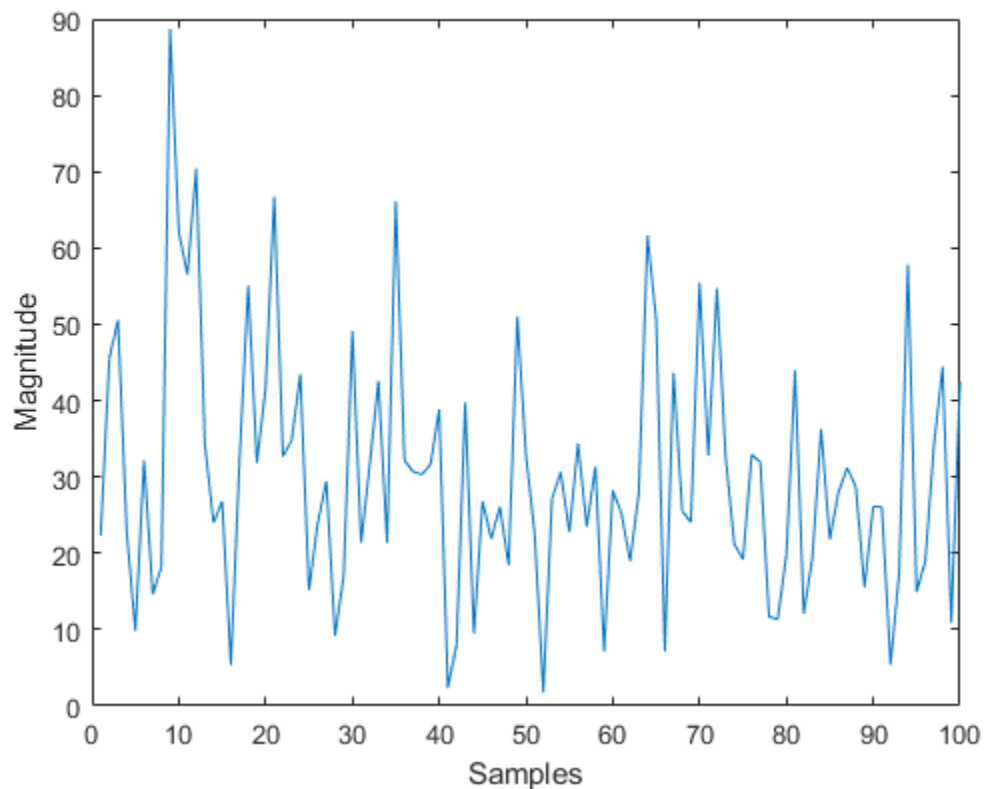
Common to All System Objects	
<code>release</code>	Allow System object property value changes

## Examples

### Plot Barrage Jammer Output

Create a barrage jammer with an effective radiated power of 1000W. Then plot the magnitude of the jammer output. barrageJammer uses a random number generator. Plots can vary from run-to-run.

```
jammer = barrageJammer('ERP',1000);  
plot(abs(jammer()))  
xlabel('Samples')  
ylabel('Magnitude')
```



## References

- [1] Ward, J. "Space-Time Adaptive Processing for Airborne Radar Data Systems," *Technical Report 1015*, MIT Lincoln Laboratory, December, 1994.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### **See Also**

`phased.Platform` | `phased.RadarTarget`

**Introduced in R2021a**

# reset

**System object:** barrageJammer

Reset random number generator for noise generation

## Syntax

reset(H)

## Description

reset(H) resets the states of the barrageJammer object, H. This method resets the random number generator state if the SeedSource property is set to 'Property'.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## step

**System object:** barrageJammer

Generate noise jamming signal

### Syntax

```
Y = step(H)
Y = step(H,N)
```

### Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

`Y = step(H)` returns a column vector, `Y`, that is a complex white Gaussian noise jamming signal. The power of the jamming signal is specified by the `ERP` property. The length of the jamming signal is specified by the `SamplesPerFrame` property. This syntax is available when the `SamplesPerFrameSource` property is 'Property'.

`Y = step(H,N)` returns the jamming signal with length `N`. This syntax is available when the `SamplesPerFrameSource` property is 'Input port'.

---

**Note** The object performs an initialization the first time the object is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

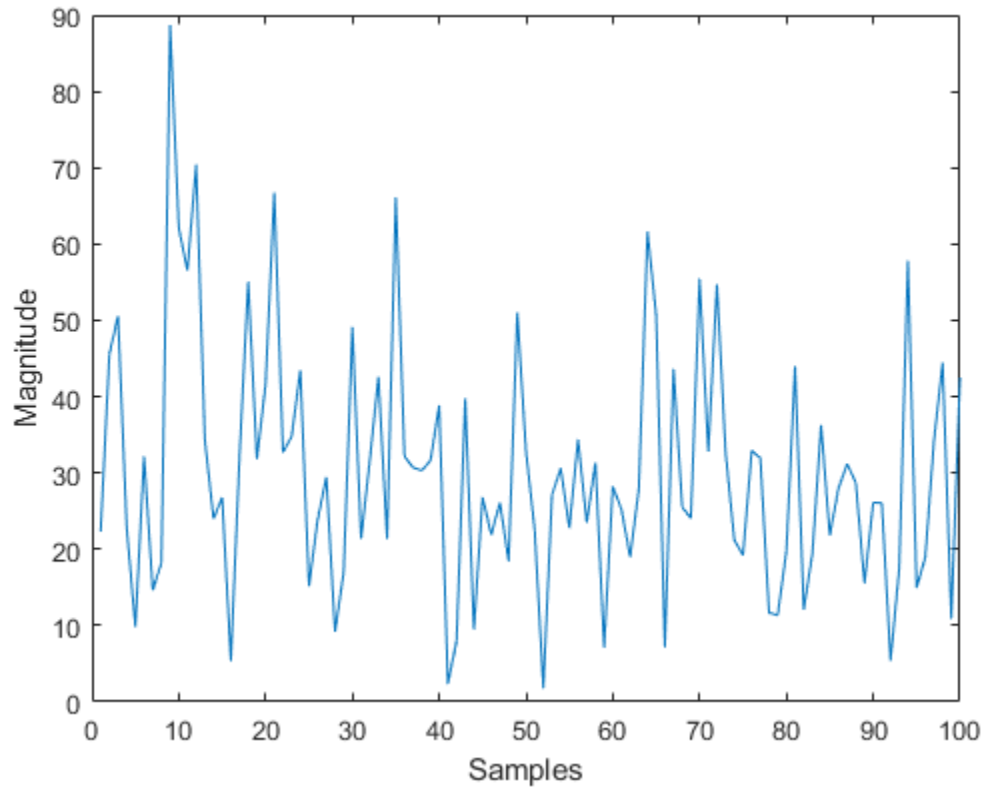
## Examples

### Plot Barrage Jammer Output

Create a barrage jammer with an effective radiated power of 1000W. Then plot the magnitude of the jammer output. `barrageJammer` uses a random number generator. Plots can vary from run-to-run.

```
jammer = barrageJammer('ERP',1000);
plot(abs(jammer()))
xlabel('Samples')
ylabel('Magnitude')
```





## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

**Introduced in R2021a**

## backscatterBicyclist

Backscatter radar signals from bicyclist

### Description

The `backscatterBicyclist` object simulates backscattered radar signals reflected from a moving bicyclist. The bicyclist consists of both the bicycle and its rider. The object models the motion of the bicyclist and computes the sum of all reflected signals from multiple discrete scatterers on the bicyclist. The model ignores internal occlusions within the bicyclist. The reflected signals are based on a multi-scatterer model developed from a 77 GHz radar system.

Scatterers are located on five major bicyclist components:

- Bicycle frame and rider
- Bicycle pedals
- Upper and lower legs of the rider
- Front wheel
- Back wheel

Excluding the wheels, there are 114 scatterers on the bicyclist. The wheels contain scatterers on the rim and spokes. The number of scatterers on the wheels depends on the number of spokes per wheel. The number of spokes is specified using the `NumWheelSpokes` property.

You can obtain the current bicyclist position and velocity by calling the `move` object function. Calling this function also updates the position and velocity for the next time epoch. To obtain the reflected signal, call the `reflect` object function. You can plot the instantaneous position of the bicyclist using the `plot` object function.

### Creation

#### Syntax

```
bicyclist = backscatterBicyclist  
bicyclist = backscatterBicyclist(Name,Value,...)
```

#### Description

`bicyclist = backscatterBicyclist` creates a `backscatterBicyclist` object, `bicyclist`, having default property values.

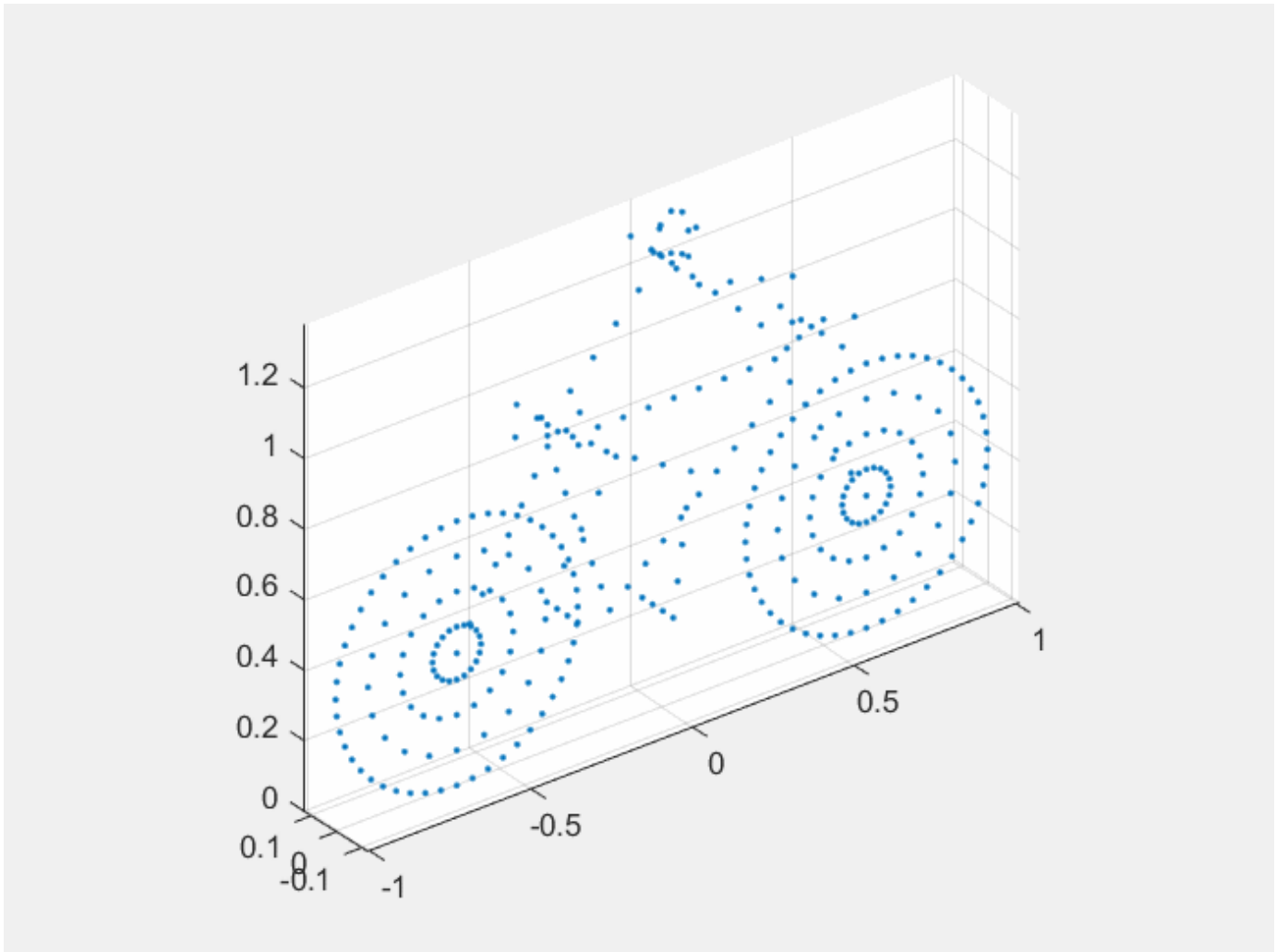
`bicyclist = backscatterBicyclist(Name,Value,...)` creates a `backscatterBicyclist` object, `bicyclist`, with each specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as `(Name1,Value1,...,NameN,ValueN)`. Any unspecified properties take default values. For example,

```
bicyclist = backscatterBicyclist( ...  
    'NumWheelSpokes',18,'Speed',10.0, ...
```

```
'InitialPosition',[0;0;0],'InitialHeading',90, ...
'GearTransmissionRatio',5.5);
```

models a bicyclist with 18 spokes on each wheel that is moving along the positive  $y$ -axis at 10 meters per second. The gear transmission ratio of 5.5 indicates that there are 5.5 wheel rotations for each pedal rotation. The bicyclist is heading along the  $y$ -axis.

This figure illustrates a bicyclist starting to turn left.



## Properties

### **NumWheelSpokes** — Number of spokes per wheel

20 (default) | positive integer

Number of spokes per wheel of the bicycle, specified as a positive integer from 3 to 50, inclusive.

Data Types: double

### **GearTransmissionRatio** — Ratio of wheel rotations to pedal rotations

1.5 (default) | positive scalar

Ratio of wheel rotations to pedal rotations, specified as a positive scalar. The gear ratio must be in the range from 0.5 through 6. Units are dimensionless.

Data Types: `double`

#### **OperatingFrequency — Carrier frequency of narrowband signals**

`77e9` (default) | positive scalar

Carrier frequency of the narrowband incident signals, specified as a positive scalar. Units are in Hz.

Example: `900e6`

Data Types: `double`

#### **InitialPosition — Initial position of bicyclist**

`[0;0;0]` (default) | 3-by-1 real-valued vector

Initial position of the bicyclist, specified as a 3-by-1 real-valued vector in the form of  $[x;y;z]$  in global coordinates. Units are in meters. The initial position corresponds to the location of the origin of the bicycle coordinates. The origin is at the center of mass of the scatterers of the default bicyclist configuration projected onto the ground.

Data Types: `double`

#### **InitialHeading — Initial heading of bicyclist**

`0` (default) | scalar

Initial heading of bicyclist, specified as a scalar. Heading is measured in the  $xy$ -plane from the  $x$ -axis towards the  $y$ -axis. Heading is with respect to global coordinates. Units are in degrees.

Data Types: `double`

#### **Speed — Speed of bicyclist**

`4` (default) | nonnegative scalar

Speed of bicyclist, specified as a nonnegative scalar. The motion model limits the speed to a maximum of 60 m/s (216 kph). Speed is defined with respect to global coordinates. Units are in meters per second.

Data Types: `double`

#### **Coast — Set bicycle coasting state**

`false` (default) | `true`

Set bicycle coasting state, specified as `false` or `true`. If set to `true`, the bicyclist is not pedaling, but the wheels are still rotating (freewheeling). If set to `false`, the bicyclist is pedaling, and the `GearTransmissionRatio` determines the wheel rotations to pedal rotations.

Data Types: `logical`

#### **PropagationSpeed — Signal propagation speed**

`physconst('LightSpeed')` (default) | positive scalar

Signal propagation speed, specified as a positive scalar. Units are in meters per second. The default propagation speed is the value returned by `physconst('LightSpeed')`. See `physconst` for more information.

Example: `3e8`

Data Types: double

### **AzimuthAngles — Radar cross-section azimuth angles**

$[-180:180]$  (default) | 1-by- $P$  real-valued row vector |  $P$ -by-1 real-valued column vector

Radar cross-section azimuth angles, specified as a 1-by- $P$  or  $P$ -by-1 real-valued vector. This property defines the azimuth coordinates of each column of the radar cross-section matrix specified by the `RCSPattern` property.  $P$  must be greater than two. Angle units are in degrees.

Example: `[-45:0.1:45]`

Data Types: double

### **ElevationAngles — Radar cross-section elevation angles**

0 (default) | scalar | 1-by- $Q$  real-valued row vector |  $Q$ -by-1 real-valued column vector

Radar cross-section elevation angles, specified as a 1-by- $Q$  or  $Q$ -by-1 real-valued vector. This property defines the elevation coordinates of each row of the radar cross-section matrix specified by the `RCSPattern` property.  $Q$  must be greater than two. Angle units are in degrees.

Example: `[-30:0.1:30]`

Data Types: double

### **RCSPattern — Radar cross-section pattern**

1-by-361 real-valued matrix (default) |  $Q$ -by- $P$  real-valued vector | 1-by- $P$  real-valued vector

Radar cross-section (RCS) pattern, specified as a  $Q$ -by- $P$  real-valued matrix or a 1-by- $P$  real-valued vector. Matrix rows represent constant elevation, and columns represent constant azimuth.  $Q$  is the length of the vector defined by the `ElevationAngles` property.  $P$  is the length of the vector defined by the `AzimuthAngles` property. Units are in square meters.

You can also specify the pattern as a 1-by- $P$  real-valued vector of azimuth angles for a single elevation.

The default value of this property is a 1-by-361 matrix containing values derived from 77 GHz radar measurements of a bicyclist. The default values of `AzimuthAngles` and `ElevationAngles` correspond to the default RCS matrix.

Example: `[1, .5; .5, 1]`

Data Types: double

## **Object Functions**

### **Specific to This Object**

<code>getNumScatterers</code>	Number of scatterers on bicyclist
<code>move</code>	Position, velocity, and orientation of moving bicyclist
<code>plot</code>	Display locations of scatterers on bicyclist
<code>reflect</code>	Reflected signal from moving bicyclist

### **Common to All System Objects**

<code>step</code>	Run System object algorithm
-------------------	-----------------------------

release Release resources and allow changes to System object property values and input characteristics  
reset Reset internal states of System object

## Examples

### Radar Signal Backscattered by Bicyclist

Compute the backscattered radar signal from a bicyclist moving along the x-axis at 5 m/s away from a radar. Assume that the radar is located at the origin. The radar transmits an LFM signal at 24 GHz with a 300 MHz bandwidth. A signal is reflected at the moment the bicyclist starts to move and then one second later.

#### Initialize Bicyclist, Waveform, and Propagation Channel Objects

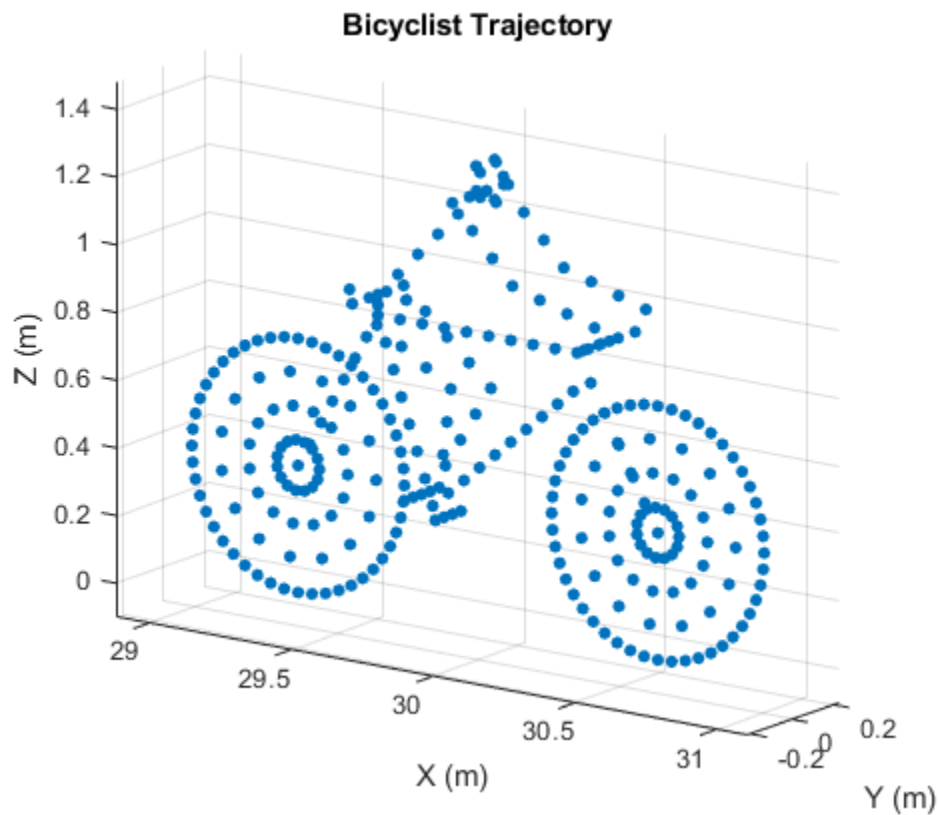
Initialize the `backscatterBicyclist`, `phased.LinearFMWaveform`, and `phased.FreeSpace` objects. Assume a 300 MHz sampling frequency. The initial position of the bicyclist lies on the x-axis 30 meters from the radar.

```
bw = 300e6;
fs = bw;
fc = 24e9;
radarpos = [0;0;0];
bpos = [30;0;0];
bicyclist = backscatterBicyclist( ...
    'OperatingFrequency',fc,'NumWheelSpokes',15, ...
    'InitialPosition',bpos,'Speed',5.0, ...
    'InitialHeading',0.0);
lfmwav = phased.LinearFMWaveform( ...
    'SampleRate',fs, ...
    'SweepBandwidth',bw);
sig = lfmwav();
chan = phased.FreeSpace( ...
    'OperatingFrequency',fc, ...
    'SampleRate',fs, ...
    'TwoWayPropagation',true);
```

#### Plot Initial Bicyclist Position

Using the `move` object function, obtain the initial scatterer positions, velocities and the orientation of the bicyclist. Plot the initial position of the bicyclist. The `dt` argument of the `move` object function determines that the next call to `move` returns the bicyclist state of motion `dt` seconds later.

```
dt = 1.0;
[bpos,bvel,bax] = move(bicyclist,dt,0);
plot(bicyclist)
```



### Obtain First Reflected Signal

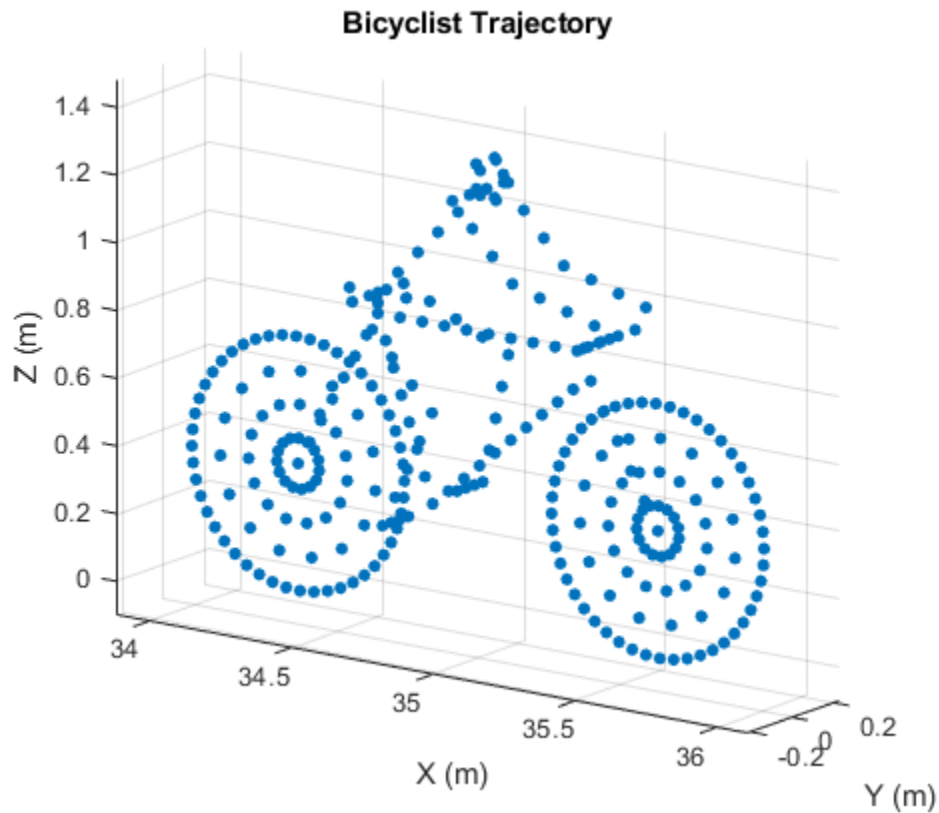
Propagate the signal to all scatterers and obtain the cumulative reflected return signal.

```
N = getNumScatterers(bicyclist);
sigtrns = chan(repmat(sig,1,N),radarpos,bpos,[0;0;0],bvel);
[rngs,ang] = rangeangle(radarpos,bpos,bax);
y0 = reflect(bicyclist,sigtrns,ang);
```

### Plot Bicyclist Position After Position Update

After the bicyclist has moved, obtain the scatterer positions and velocities and then move the bicycle along its trajectory for another second.

```
[bpos,bvel,bax] = move(bicyclist,dt,0);
plot(bicyclist)
```



### Obtain Second Reflected Signal

Propagate the signal to all scatterers at their new positions and obtain the cumulative reflected return signal.

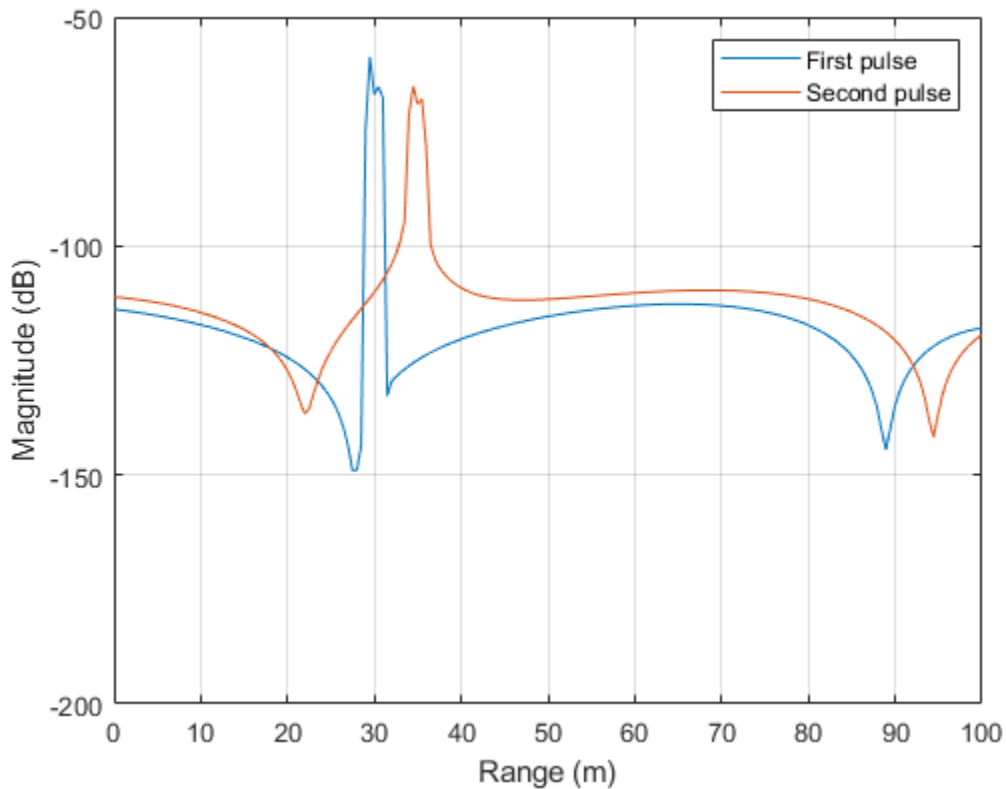
```
sigtrns = chan(repmat(sig,1,N),radarpos,bpos,[0;0;0],bvel);
[~,ang] = rangeangle(radarpos,bpos,bax);
y1 = reflect(bicyclist,sigtrns,ang);
```

### Match Filter Reflected Signals

Match filter the reflected signals and plot them together.

```
mfsig = getMatchedFilter(lfmwav);
nsamp = length(mfsig);
mf = phased.MatchedFilter('Coefficients',mfsig);
ymf = mf([y0 y1]);
fdelay = (nsamp-1)/fs;
t = (0:size(ymf,1)-1)/fs - fdelay;
c = physconst('LightSpeed');
plot(c*t/2,mag2db(abs(ymf)))
ylim([-200 -50])
xlabel('Range (m)')
ylabel('Magnitude (dB)')
ax = axis;
axis([0,100,ax(3),ax(4)])
grid
legend('First pulse','Second pulse')
```





Compute the difference in range between the maxima of the two pulses.

```
[maxy,idx] = max(abs(ymf));
dpeaks = t(1,idx(2)) - t(1,idx(1));
drng = c*dpeaks/2
```

```
drng = 4.9965
```

The range difference is 5 m, as expected given the bicyclist speed.

### Display Micro-Doppler Shift from Moving Bicyclist

Display a spectrogram showing the micro-Doppler effect on radar signals reflected from the scatterers on a moving bicyclist target. A stationary radar transmits 1000 pulses of an FMCW radar wave with a bandwidth of 250 MHz and of 1  $\mu$ sec duration. The radar operates at 24 GHz. The bicyclist starts 5 m from the radar and moves away at 4 m/s.

Set up the waveform, channel, transmitter, receiver, and platform System objects.

```
bw = 250e6;
fs = 2*bw;
fc = 24e9;
c = physconst('Lightspeed');
tm = 1e-6;
wav = phased.FMCWWaveform('SampleRate',fs,'SweepTime',tm, ...
```

```

    'SweepBandwidth',bw);
chan = phased.FreeSpace('PropagationSpeed',c,'OperatingFrequency',fc, ...
    'TwoWayPropagation',true,'SampleRate',fs);
radarplt = phased.Platform('InitialPosition',[0;0;0], ...
    'OrientationAxesOutputPort',true);
trx = phased.Transmitter('PeakPower',1,'Gain',25);
rcvx = phased.ReceiverPreamp('Gain',25,'NoiseFigure',10);

```

Create a bicyclist object moving at 4 meters/second.

```

bicyclistSpeed = 4;
bicyclist = backscatterBicyclist('InitialPosition',[5;0;0],'Speed',bicyclistSpeed, ...
    'PropagationSpeed',c,'OperatingFrequency',fc,'InitialHeading',0.0);
lambda = c/fc;
fmax = 2*bicyclist.GearTransmissionRatio*bicyclistSpeed/lambda;
tsamp = 1/(2*fmax);

```

Loop over 1000 pulses. Find the angle of incidence of the radar. Propagate the wave to each scatterer, and then reflect the wave from the scatterers back to the radar.

```

npulse = 1000;
xr = complex(zeros(round(fs*tm),npulse));
for m = 1:npulse
    [posr,velr,axr] = radarplt(tsamp);
    [post,velt,axt] = move(bicyclist,tsamp,0);
    [~,angrt] = rangeangle(posr,post,axt);
    x = trx(wav());
    xt = chan(repmat(x,1,size(post,2)),posr,post,velr,velt);
    xr(:,m) = rcvx(reflect(bicyclist,xt,angrt));
end

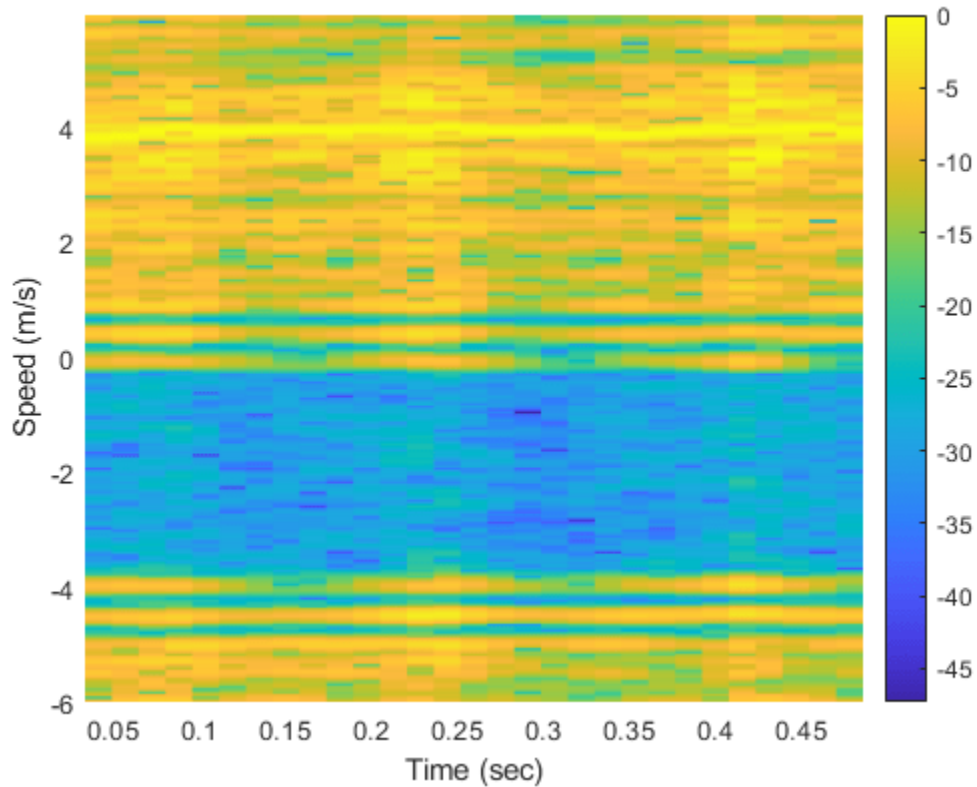
```

Process the arriving signals. First, dechirp the signal and then pass the signal into a Kaiser-windowed short-time Fourier transform.

```

xd = conj(dechirp(xr,x));
M = 128;
beta = 6;
w = kaiser(M,beta);
R = floor(1.7*(M-1)/(beta+1));
noverlap = M - R;
[S,F,T] = stft(sum(xd),1/tsamp,'Window',w,'FFTLength',M*2, ...
    'OverlapLength',noverlap);
maxval = max(10*log10(abs(S)));
pcolor(T,-F*lambda/2,10*log10(abs(S))-maxval);
shading flat;
colorbar
xlabel('Time (sec)')
ylabel('Speed (m/s)')

```

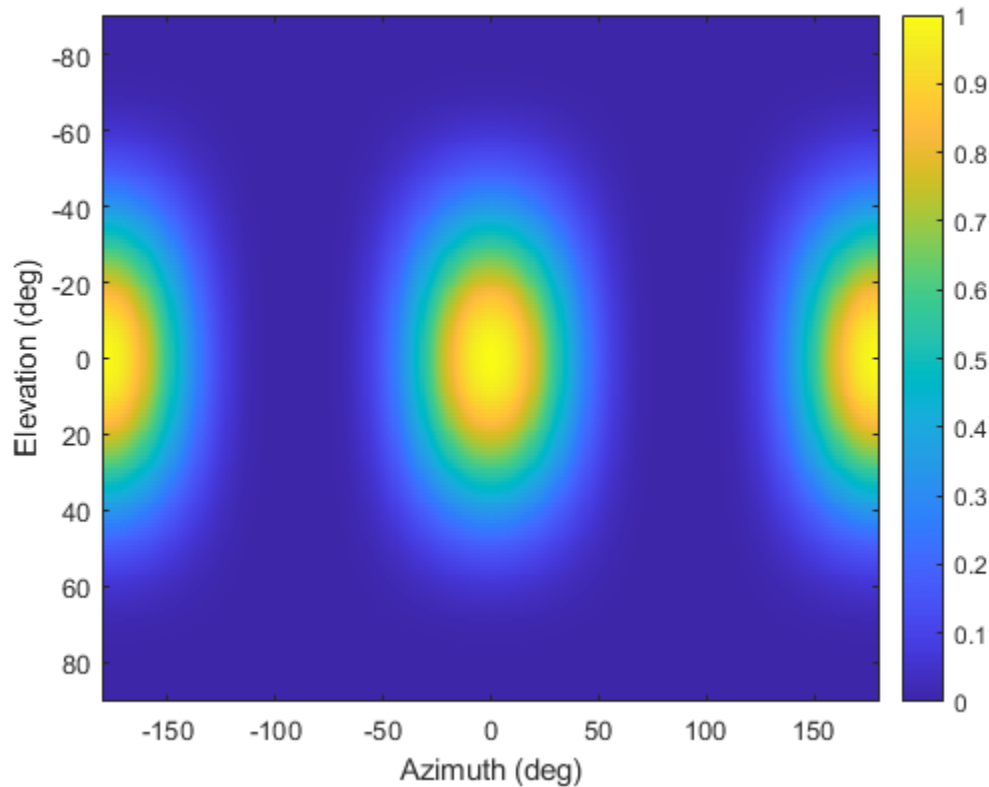


### Backscatter Bicyclist With Custom RCS Pattern

Create a custom RCS pattern to use with the `backscatterBicyclist` object.

The RCS pattern is computed from cosines raised to the fourth power. Plot the pattern.

```
az = [-180:180];  
el = [-90:90];  
caz = cosd(az').^4;  
cel = cosd(el).^4;  
rcs = (caz*cel)';  
imagesc(az,el,rcs)  
xlabel('Azimuth (deg)')  
ylabel('Elevation (deg)')  
colorbar
```



```
bicyclist = backscatterBicyclist( ...
    'NumWheelSpokes',18,'Speed',10.0, ...
    'InitialPosition',[0;0;0],'InitialHeading',90, ...
    'GearTransmissionRatio',5.5,'AzimuthAngles',az, ...
    'ElevationAngles',el,'RCSPattern',rcs);
```

## Algorithms

### Bicycle Model

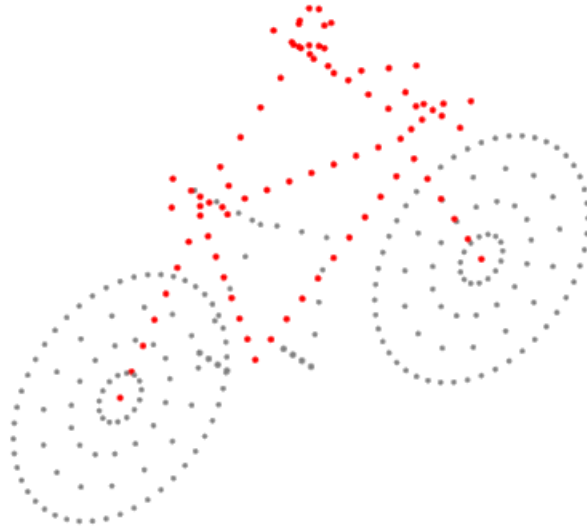
The bicyclist consists of five primary components: bicycle frame and rider, pedals, rider legs, front wheel, and rear wheel. Each component contains many scatterers. All components move with a velocity determined by the specified speed and heading properties. In addition, the legs, pedals, and wheels undergo cyclical motion determined by the speed.

#### Motion of Scatterers on Frame and Rider

Scatterers on the frame and rider are fixed with respect to the bicyclist and move with the ego velocity

where  $v$  is the speed of the bicyclist specified by the `Speed` property and  $H$  is the heading specified by the `InitialHeading` property. These properties can be changed by calling the `move` function.

This figure shows the location of the scatterers on the bicycle frame and rider.



### Motion of Scatterers on Pedals

Scatterers on the pedals move with the bicyclist but can also revolve around the crank spindle with a radius of rotation  $R_{\text{ped}}$ . There are two possible motions of the pedals depending upon whether the bicycle is coasting (freewheeling) or not coasting:

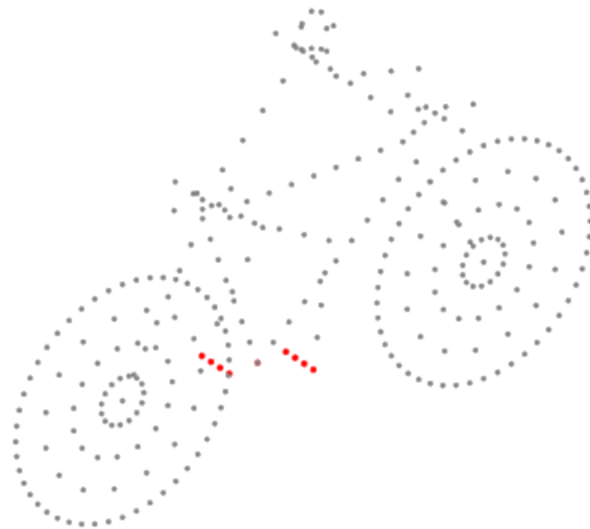
- When the bicycle is coasting, the pedals do not revolve around the crank spindle and the velocity of the pedal scatterers equals the bicyclist velocity. Their positions relative to the bicyclist are fixed. Coasting is turned on by setting the `Coast` property to `true` or by setting the `coast` argument of the `move` object function to `true`. The speed of the pedal is
- When the bicycle is not coasting, the rider is pedaling. The angular velocity of the pedals is related to the angular velocity of the wheels by

where  $G$  is the gear ratio defined by the `GearTransmissionRatio` property. The speed of a pedal scatterer equals the rotational speed of the pedal multiplied by the distance from pedal to crank spindle. The vector form of this relationship is:

The velocity of the pedal with respect to the bicyclist is then

Coasting is turned off by setting the `Coast` property to `false` or by setting the `coast` argument of the `move` object function to `false`.

This figure shows the locations of the pedal scatterers.



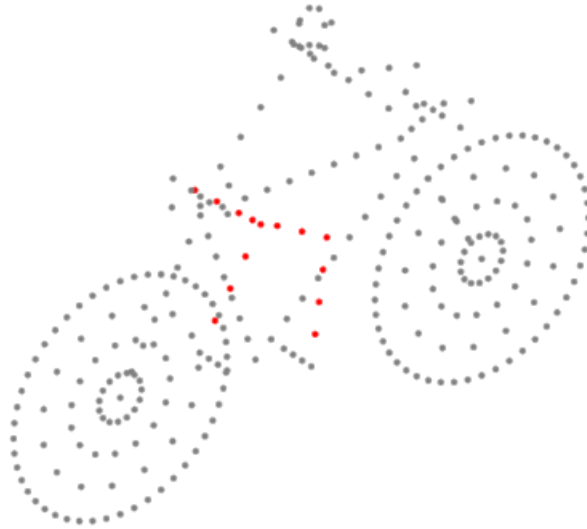
#### **Motion of Scatterers on Riders Legs**

Scatterers on the upper and lower legs of the rider move with the bicycle with an added cyclical motion. There are two possible motions of the legs depending upon whether the bicycle is coasting or not coasting:

- When the bicycle is coasting, the legs are not moving with the respect to the bicycle and the scatterers move with the velocity of the bicyclist. Coasting is turned on by setting the `Coast` property to `true` or by setting the `coast` argument of the `move` object function to `true`.
- When the bicycle is not coasting, the upper and lower legs execute reciprocating motion. The upper legs partially rotate around the hip of the rider. The foot is attached to the pedal and rotates with the pedal. The knee connects the lower and upper legs. The locations of the foot and hips of the rider determine the locations of the knees and the motion of the scatterers on the legs.

Coasting is turned off by setting the `Coast` property to `false` or by setting the `coast` argument of the `move` object function to `false`.

This figure shows the locations of the scatterers on the upper and lower legs of the rider.

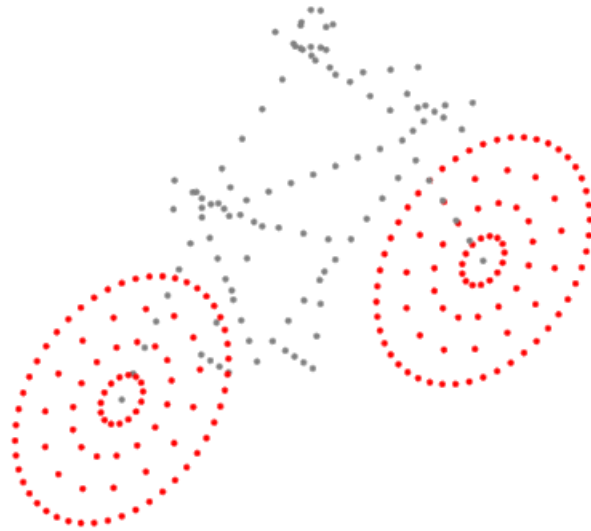


### **Motion of Scatterers on Bicycle Wheels**

Scatterers are on the spokes and rims of the wheels and revolve around the wheel axle at varying distances,  $r_{\text{spk}}$ , from the axle. The velocity of the scatterers in the bicyclist frame of reference is

The absolute velocity of a spoke or rim scatterer is

This figure shows the locations of the scatterers on the wheel rims and spokes.



### **Radar Cross-Section**

The value of the radar cross-section (RCS) of a scatterer generally depends upon the incident angle of the reflected radiation. The `backscatterBicyclist` object uses a simplified RCS model: the RCS pattern of an individual scatterer equals the total bicyclist pattern divided by the number of scatterers. The value of the RCS is computed from the RCS pattern evaluated at an average over all scatterers of the azimuth and elevation incident angles. Therefore, the RCS value is the same for all scatterers. You can specify the RCS pattern using the `RCSPattern` property of the `backscatterBicyclist` object or use the default value.

### **References**

- [1] Stolz, M. et al. "Multi-Target Reflection Point Model of Cyclists for Automotive Radar." *2017 European Radar Conference (EURAD)*, Nuremberg, 2017, pp. 94-97.
- [2] Chen, V., D. Tahmoush, and W. J. Miceli. *Radar Micro-Doppler Signatures: Processing and Applications*. The Institution of Engineering and Technology: London, 2014.
- [3] Belgiovane, D., and C. C. Chen. "Bicycles and Human Rider Backscattering at 77 GHz for Automotive Radar." *2016 10<sup>th</sup> European Conference on Antennas and Propagation (EuCAP)*, Davos, 2016, pp. 1-5.
- [4] Victor Chen, *The Micro-Doppler Effect in Radar*. Norwood, MA: Artech House, 2011.



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`getNumScatterers` | `move` | `plot` | `reflect` | `phased.BackscatterSonarTarget` | `phased.BackscatterRadarTarget` | `phased.WidebandBackscatterRadarTarget` | `phased.RadarTarget` | `backscatterPedestrian`

### Introduced in R2021a

## getNumScatterers

Number of scatterers on bicyclist

### Syntax

```
N = getNumScatterers(bicyclist)
```

### Description

`N = getNumScatterers(bicyclist)` returns the number of scatterers, `N`, on the `bicyclist`.

### Examples

#### Find Number of Bicyclist Scatterers

Use the `getNumScatterers` object function to find the number of scatterers on a bicyclist with 25 spokes. Create the `backscatterBicyclist` object and then call `getNumScatterers`.

```
fc = 77e9;
bicyclist = backscatterBicyclist( ...
    'OperatingFrequency',fc,'NumWheelSpokes',25, ...
    'InitialPosition',[5;0;0]);
N = getNumScatterers(bicyclist)

N = 359
```

### Input Arguments

#### **bicyclist** – Bicyclist target

`backscatterBicyclist` object

Bicyclist, specified as a `backscatterBicyclist` object.

### Output Arguments

#### **N** – Number of scatterers

positive integer

Number of scatterers on `bicyclist`, returned as a positive integer.

### Extended Capabilities

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### See Also

`move` | `plot` | `reflect`

**Introduced in R2019b**

## move

Position, velocity, and orientation of moving bicyclist

### Syntax

```
[bpos,bvel,bax] = move(bicyclist,T,angh)
[bpos,bvel,bax] = move(bicyclist,T,angh,speed)
[bpos,bvel,bax] = move(bicyclist,T,angh,speed,coast)
```

### Description

[bpos,bvel,bax] = move(bicyclist,T,angh) returns the current positions, bpos, and current velocities, bvel, of the scatterers and the current orientation axes, bax, of the bicyclist. The positions, velocities, and axes are then updated for the next time interval T. ang specifies the heading angle of the bicyclist.

[bpos,bvel,bax] = move(bicyclist,T,angh,speed) also specifies the speed of the bicyclist.

[bpos,bvel,bax] = move(bicyclist,T,angh,speed,coast) also specifies the coasting state, coast, of the bicyclist.

### Examples

#### Display Bicyclist Scatterer Positions

Plot the positions of all bicyclist scatterers. Assume there are 15 spokes per wheel.

Create a backscatterBicyclist object for a radar system operating at 77 GHz and having a bandwidth of 300 MHz. The sampling rate is twice the bandwidth. The bicyclist is initially 5 meters away from the radar.

```
bw = 300e6;
fs = 2*bw;
fc = 77e9;
rpos = [0;0;0];
bpos = [5;0;0];
bicyclist = backscatterBicyclist( ...
    'OperatingFrequency',fc,'NumWheelSpokes',15, ...
    'InitialPosition',bpos);
```

Obtain the initial position of the scatterers and advance the motion by 1 second.

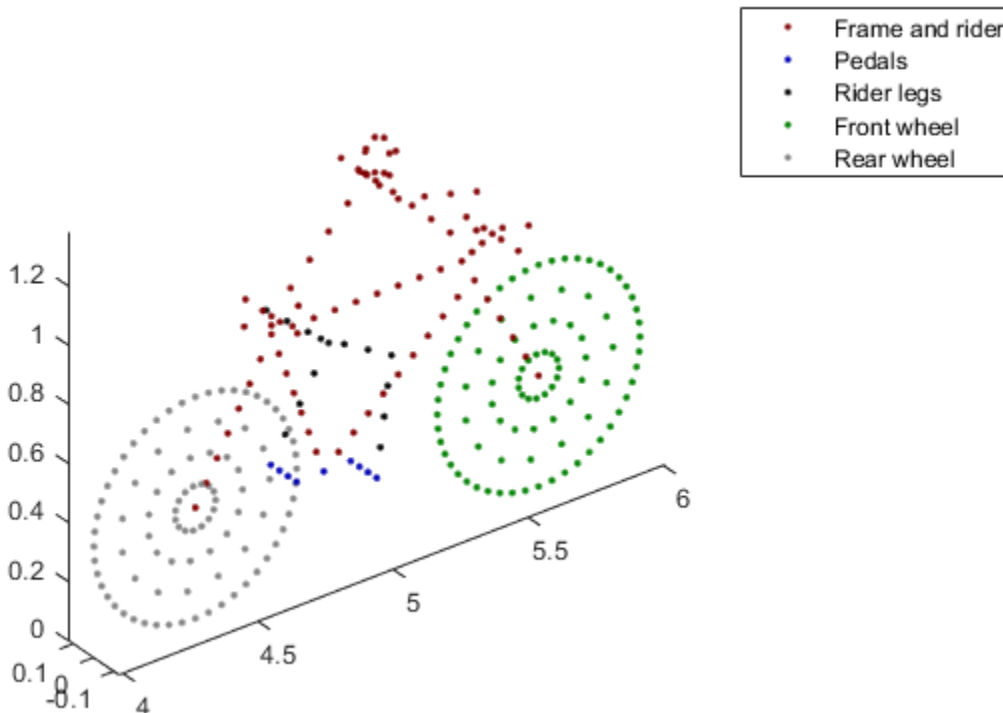
```
[bpos,bvel,bax] = move(bicyclist,1,0);
```

Obtain the number of scatterers and the indices of the wheel scatterers.

```
N = getNumScatterers(bicyclist);
Nsw = (N-114+1)/2;
idxfrontwheel = (114:(114 + Nsw - 1));
idxrearwheel = (114 + Nsw):N;
```

Plot the locations of the scatterers.

```
plot3(bpos(1,1:90),bpos(2,1:90),bpos(3,1:90), ...
      'LineStyle','none','Color',[0.5,0,0],'Marker','.')
axis equal
hold on
plot3(bpos(1,91:99),bpos(2,91:99),bpos(3,91:99), ...
      'LineStyle','none','Color',[0,0,0.7],'Marker','.')
plot3(bpos(1,100:113),bpos(2,100:113),bpos(3,100:113), ...
      'LineStyle','none','Color',[0,0,0.5],'Marker','.')
plot3(bpos(1,idxfrontwheel),bpos(2,idxfrontwheel),bpos(3,idxfrontwheel), ...
      'LineStyle','none','Color',[0,0.5,0],'Marker','.')
plot3(bpos(1,idxrearwheel),bpos(2,idxrearwheel),bpos(3,idxrearwheel), ...
      'LineStyle','none','Color',[0.5,0.5,0.5],'Marker','.')
hold off
legend('Frame and rider','Pedals','Rider legs','Front wheel','Rear wheel')
```



### Model Bicyclist Moving along Arc

Display an animation of a bicyclist riding in a quarter circle. Use the default property values of the `backscatterBicyclist` object. The motion is updated at 30 millisecond intervals for 500 steps.

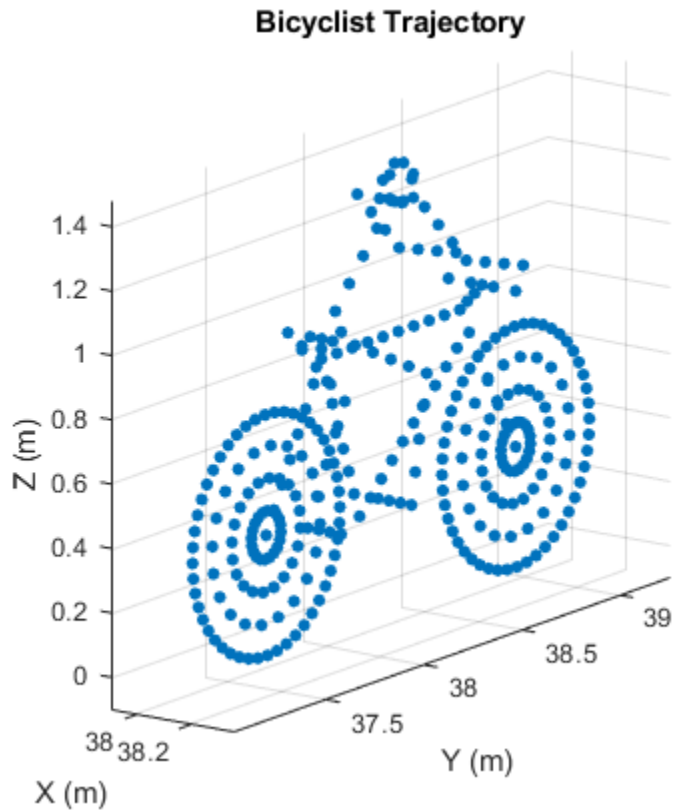
```
dt = 0.03;
M = 500;
angstep = 90/M;
```

```

bicycle = backscatterBicyclist;

for m = 1:M
    [bpos,bvel,bang] = move(bicycle,dt,angstep*m);
    plot(bicycle)
end

```



## Input Arguments

### **bicyclist** – Bicyclist target

backscatterBicyclist object

Bicyclist, specified as a backscatterBicyclist object.

### **T** – Duration of next motion interval

scalar

Duration of next motion interval, specified as a positive scalar. The scatterer positions and velocities and bicyclist orientation are updated over this time duration. Units are in seconds.

Example: 0.75

Data Types: double

### **angh** – Bicyclist heading

0.0 | scalar

Heading of the bicyclist, specified as a scalar. Heading is measured in the  $xy$ -plane from the  $x$ -axis towards the  $y$ -axis. Units are in degrees.

Example: -34

Data Types: `double`

### **speed — Bicyclist speed**

value Speed property (default) | nonnegative scalar

Bicyclist speed, specified as a nonnegative scalar. The motion model limits the speed to 60 m/s. Units are in meters per second. Alternatively, you can specify the bicyclist speed using the `Speed` property of the `backscatterBicyclist` object.

Example: 8

Data Types: `double`

### **coast — Set bicyclist coasting state**

value of `Coast` property (default) | `false` | `true`

Set bicyclist coasting state, specified as `false` or `true`. If set to `true`, the bicyclist is not pedaling, but the wheels are still rotating (freewheeling). If set to `false`, the bicyclist is pedaling, and the `GearTransmissionRatio` determines the ratio of wheel rotations to pedal rotations. Alternatively, you can specify the bicyclist coasting state using the `Coast` property of the `backscatterBicyclist` object.

Data Types: `logical`

## **Output Arguments**

### **bpos — Positions of bicyclist scatterers**

real-valued 3-by- $N$  matrix

Positions of bicyclist scatterers, returned as a real-valued 3-by- $N$  matrix. Each column represents the Cartesian position,  $[x;y;z]$ , of one of the bicyclist scatterers.  $N$  represents the number of scatterers and can be obtained using the `getNumScatterers` object function. Units are in meters. See “Bicycle Scatterer Indices” on page 4-448 for the column representing the position of each scatterer.

Data Types: `double`

### **bvel — Velocities of bicyclist scatterers**

real-valued 3-by- $N$  matrix

Velocities of bicyclist scatterers, returned as a real-valued 3-by- $N$  matrix. Each column represents the Cartesian velocity,  $[vx;vy;vz]$ , of one of the bicyclist scatterers.  $N$  represents the number of scatterers and can be obtained using the `getNumScatterers` object function. Units are in meters per second. See “Bicycle Scatterer Indices” on page 4-448 for the column representing the velocity of each scatterer.

Data Types: `double`

### **bax — Orientation axes of bicyclist**

real-valued 3-by-3 matrix

Orientation axes of bicyclist, returned as a real-valued 3-by-3 matrix. Units are dimensionless.

Data Types: `double`

## More About

### Bicycle Scatterer Indices

Bicyclist scatterer indices define which columns in the scatterer position or velocity matrices contain the position and velocity data for a specific scatterer. For example, column 92 of `bpos` specifies the 3-D position of one of the scatterers on a pedal.

The wheel scatterers are equally divided between the wheels. You can determine the total number of wheel scatterers,  $N$ , by subtracting 113 from the output of the `getNumScatterers` function. The number of scatterers per wheel is  $N_{sw} = N/2$ .

### Bicyclist Scatterer Indices

Bicyclist Component	Bicyclist Scatterer Index
Frame and rider	1 ... 90
Pedals	91 ... 99
Rider legs	100 ... 113
Front wheel	114 ... 114 + $N_{sw}$ - 1
Rear wheel	114 + $N_{sw}$ ... 114 + $N$ - 1

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`getNumScatterers` | `plot` | `reflect`

**Introduced in R2021a**



## plot

Display locations of scatterers on bicyclist

### Syntax

```
plot(bicyclist)
fhndl = plot(bicyclist)
fhndl = plot(bicyclist, 'Parent', ax)
```

### Description

`plot(bicyclist)` displays the positions of all scatterers on a `bicyclist` at the current time. To display the current position of the bicyclist, call the `plot` object function after calling the `move` object function. Calling `plot` before any call to `move` displays the bicyclist at the origin.

`fhndl = plot(bicyclist)` returns the figure handle of the display window.

`fhndl = plot(bicyclist, 'Parent', ax)` also specifies the plot axes for the bicyclist plot.

### Examples

#### Radar Signal Backscattered by Bicyclist

Compute the backscattered radar signal from a bicyclist moving along the  $x$ -axis at 5 m/s away from a radar. Assume that the radar is located at the origin. The radar transmits an LFM signal at 24 GHz with a 300 MHz bandwidth. A signal is reflected at the moment the bicyclist starts to move and then one second later.

#### Initialize Bicyclist, Waveform, and Propagation Channel Objects

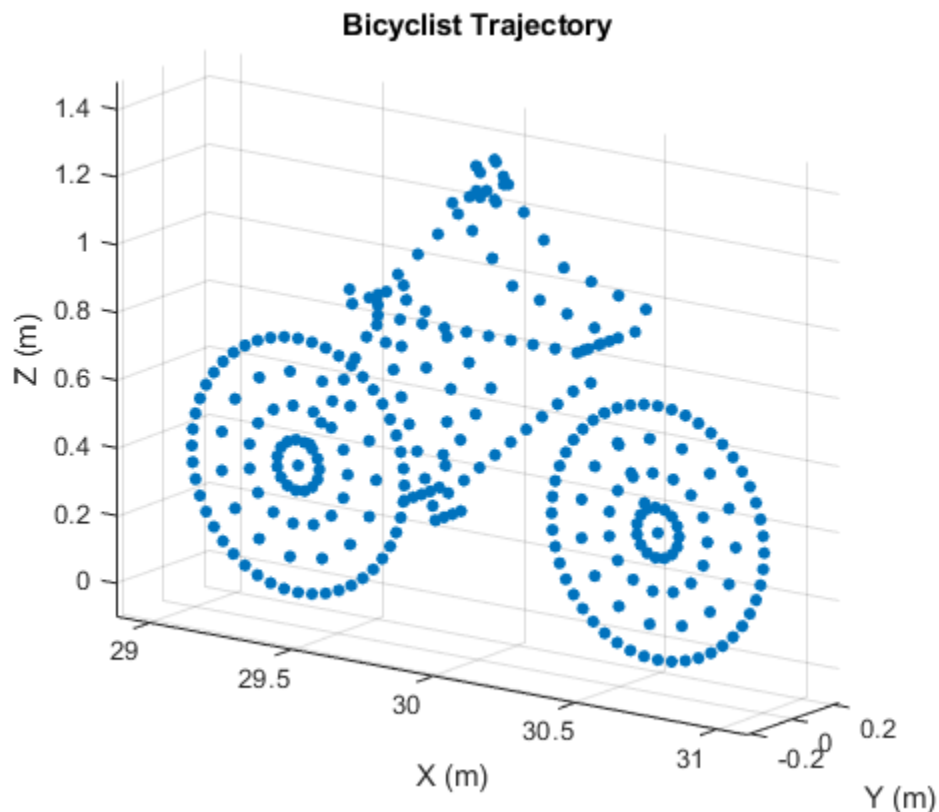
Initialize the `backscatterBicyclist`, `phased.LinearFMWaveform`, and `phased.FreeSpace` objects. Assume a 300 MHz sampling frequency. The initial position of the bicyclist lies on the  $x$ -axis 30 meters from the radar.

```
bw = 300e6;
fs = bw;
fc = 24e9;
radarpos = [0;0;0];
bpos = [30;0;0];
bicyclist = backscatterBicyclist( ...
    'OperatingFrequency',fc,'NumWheelSpokes',15, ...
    'InitialPosition',bpos,'Speed',5.0, ...
    'InitialHeading',0.0);
lfmwav = phased.LinearFMWaveform( ...
    'SampleRate',fs, ...
    'SweepBandwidth',bw);
sig = lfmwav();
chan = phased.FreeSpace( ...
    'OperatingFrequency',fc, ...
    'SampleRate',fs, ...
    'TwoWayPropagation',true);
```

### Plot Initial Bicyclist Position

Using the move object function, obtain the initial scatterer positions, velocities and the orientation of the bicyclist. Plot the initial position of the bicyclist. The `dt` argument of the move object function determines that the next call to move returns the bicyclist state of motion `dt` seconds later.

```
dt = 1.0;
[bpos,bvel,bax] = move(bicyclist,dt,0);
plot(bicyclist)
```



### Obtain First Reflected Signal

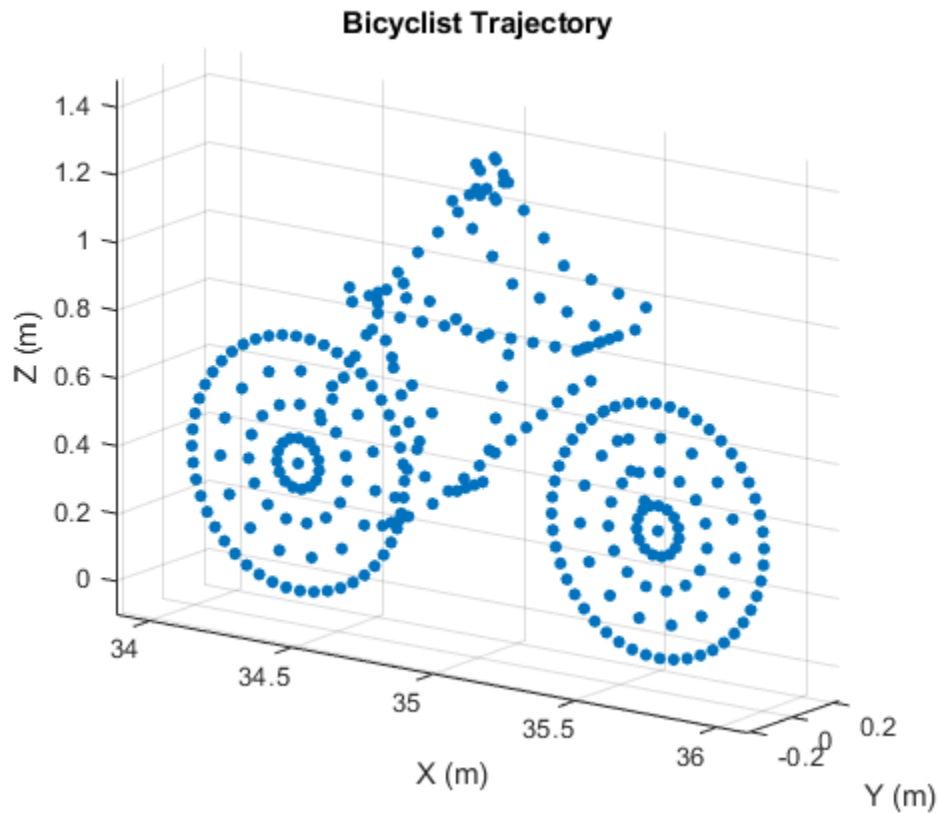
Propagate the signal to all scatterers and obtain the cumulative reflected return signal.

```
N = getNumScatterers(bicyclist);
sigtrns = chan(repmat(sig,1,N),radarpos,bpos,[0;0;0],bvel);
[rngs,ang] = rangeangle(radarpos,bpos,bax);
y0 = reflect(bicyclist,sigtrns,ang);
```

### Plot Bicyclist Position After Position Update

After the bicyclist has moved, obtain the scatterer positions and velocities and then move the bicycle along its trajectory for another second.

```
[bpos,bvel,bax] = move(bicyclist,dt,0);
plot(bicyclist)
```



### Obtain Second Reflected Signal

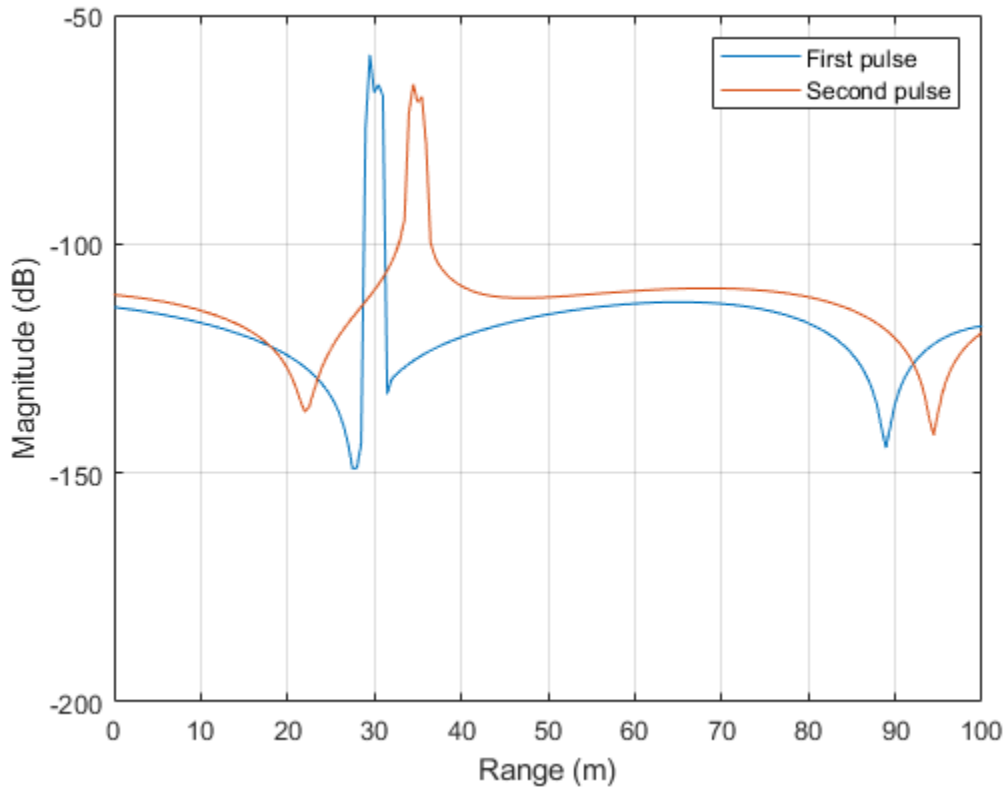
Propagate the signal to all scatterers at their new positions and obtain the cumulative reflected return signal.

```
sigtrns = chan(repmat(sig,1,N),radarpos,bpos,[0;0;0],bvel);
[~,ang] = rangeangle(radarpos,bpos,bax);
y1 = reflect(bicyclist,sigtrns,ang);
```

### Match Filter Reflected Signals

Match filter the reflected signals and plot them together.

```
mfsig = getMatchedFilter(lfmwav);
nsamp = length(mfsig);
mf = phased.MatchedFilter('Coefficients',mfsig);
ymf = mf([y0 y1]);
fdelay = (nsamp-1)/fs;
t = (0:size(ymf,1)-1)/fs - fdelay;
c = physconst('LightSpeed');
plot(c*t/2,mag2db(abs(ymf)))
ylim([-200 -50])
xlabel('Range (m)')
ylabel('Magnitude (dB)')
ax = axis;
axis([0,100,ax(3),ax(4)])
grid
legend('First pulse','Second pulse')
```



Compute the difference in range between the maxima of the two pulses.

```
[maxy,idx] = max(abs(ymf));
dpeaks = t(1,idx(2)) - t(1,idx(1));
drng = c*dpeaks/2
```

```
drng = 4.9965
```

The range difference is 5 m, as expected given the bicyclist speed.

## Input Arguments

### **bicyclist** – Bicyclist target

backscatterBicyclist object

Bicyclist, specified as a backscatterBicyclist object.

### **ax** – Plot axes

axes handle

Plot axes, specified as an axes handle.

Data Types: double

## Output Arguments

**fhndl** — figure handle

figure handle

Figure handle of plot window.

## See Also

getNumScatterers | move | reflect

**Introduced in R2019b**

## reflect

Reflected signal from moving bicyclist

### Syntax

```
Y = reflect(bicyclist,X,ang)
```

### Description

`Y = reflect(bicyclist,X,ang)` returns the total reflected signal, `Y`, from a bicyclist. The total reflected signal is the sum of all reflected signals from the bicyclist scatterers. `X` represents the incident signals at each scatterer. `ang` defines the directions of the incident and reflected signals with respect to the each scatterers.

The reflected signal strength depends on the value of the radar cross-section at the incident angle. This simplified model uses the same value for all scatterers.

### Examples

#### Radar Signal Backscattered by Bicyclist

Compute the backscattered radar signal from a bicyclist moving along the `x`-axis at 5 m/s away from a radar. Assume that the radar is located at the origin. The radar transmits an LFM signal at 24 GHz with a 300 MHz bandwidth. A signal is reflected at the moment the bicyclist starts to move and then one second later.

#### Initialize Bicyclist, Waveform, and Propagation Channel Objects

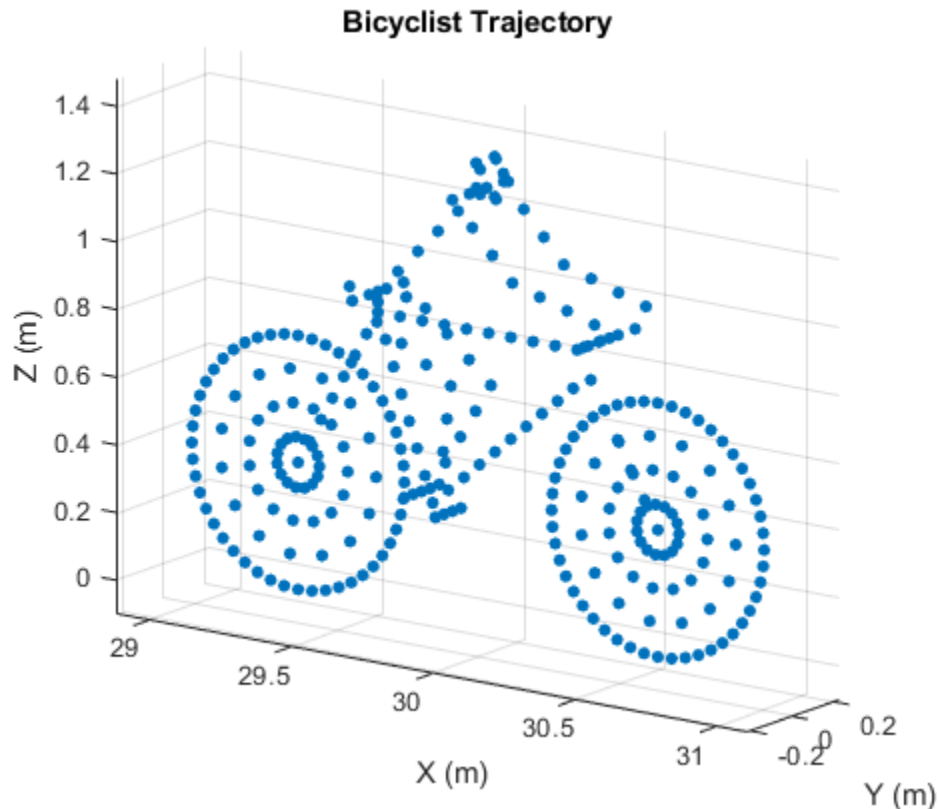
Initialize the `backscatterBicyclist`, `phased.LinearFMWaveform`, and `phased.FreeSpace` objects. Assume a 300 MHz sampling frequency. The initial position of the bicyclist lies on the `x`-axis 30 meters from the radar.

```
bw = 300e6;
fs = bw;
fc = 24e9;
radarpos = [0;0;0];
bpos = [30;0;0];
bicyclist = backscatterBicyclist( ...
    'OperatingFrequency',fc,'NumWheelSpokes',15, ...
    'InitialPosition',bpos,'Speed',5.0, ...
    'InitialHeading',0.0);
lfmwav = phased.LinearFMWaveform( ...
    'SampleRate',fs, ...
    'SweepBandwidth',bw);
sig = lfmwav();
chan = phased.FreeSpace( ...
    'OperatingFrequency',fc, ...
    'SampleRate',fs, ...
    'TwoWayPropagation',true);
```

### Plot Initial Bicyclist Position

Using the move object function, obtain the initial scatterer positions, velocities and the orientation of the bicyclist. Plot the initial position of the bicyclist. The `dt` argument of the move object function determines that the next call to move returns the bicyclist state of motion `dt` seconds later.

```
dt = 1.0;
[bpos,bvel,bax] = move(bicyclist,dt,0);
plot(bicyclist)
```



### Obtain First Reflected Signal

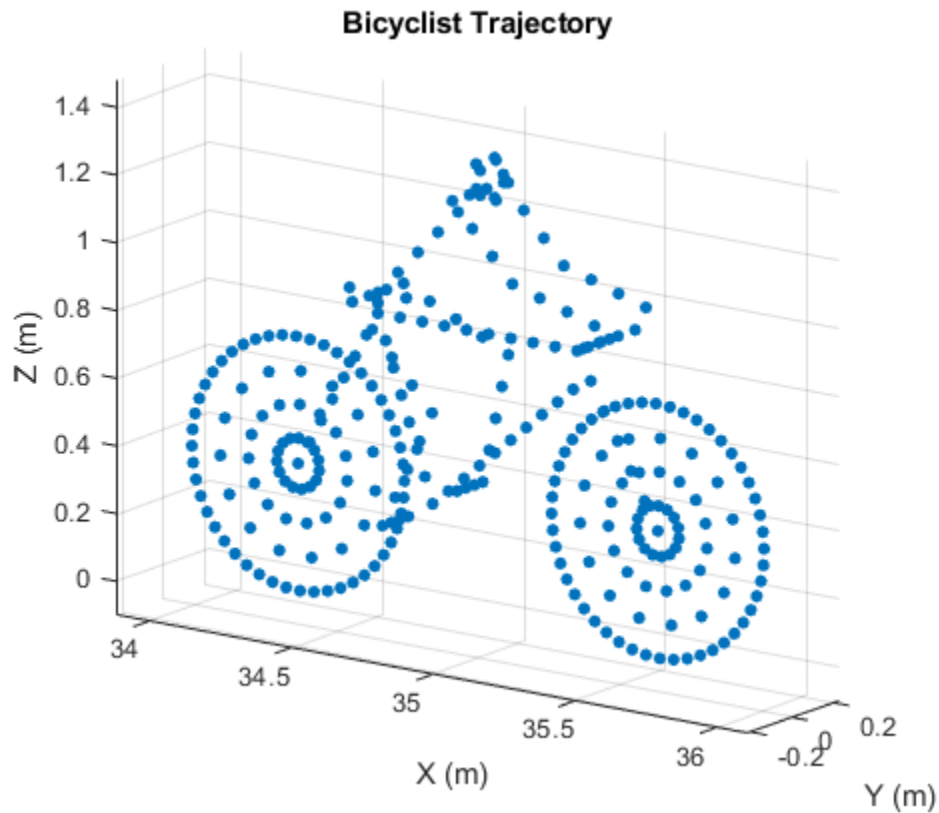
Propagate the signal to all scatterers and obtain the cumulative reflected return signal.

```
N = getNumScatterers(bicyclist);
sigtrns = chan(repmat(sig,1,N),radarpos,bpos,[0;0;0],bvel);
[rngs,ang] = rangeangle(radarpos,bpos,bax);
y0 = reflect(bicyclist,sigtrns,ang);
```

### Plot Bicyclist Position After Position Update

After the bicyclist has moved, obtain the scatterer positions and velocities and then move the bicycle along its trajectory for another second.

```
[bpos,bvel,bax] = move(bicyclist,dt,0);
plot(bicyclist)
```



### Obtain Second Reflected Signal

Propagate the signal to all scatterers at their new positions and obtain the cumulative reflected return signal.

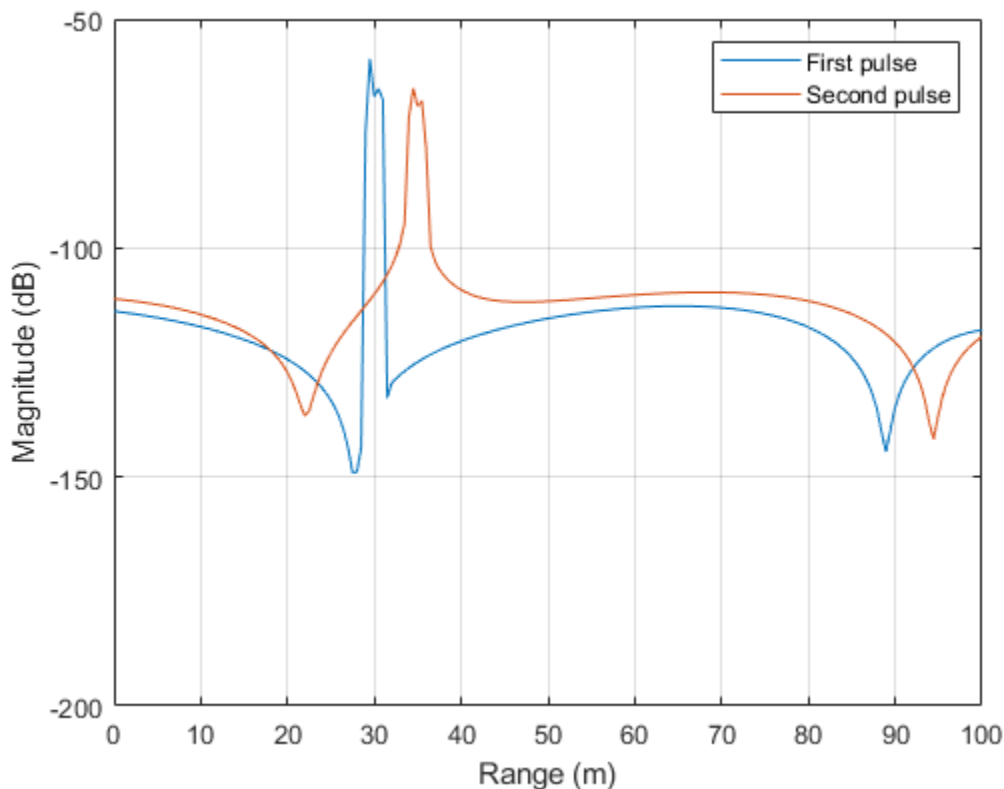
```
sigtrns = chan(repmat(sig,1,N),radarpos,bpos,[0;0;0],bvel);
[~,ang] = rangeangle(radarpos,bpos,bax);
y1 = reflect(bicyclist,sigtrns,ang);
```

### Match Filter Reflected Signals

Match filter the reflected signals and plot them together.

```
mfsig = getMatchedFilter(lfmwav);
nsamp = length(mfsig);
mf = phased.MatchedFilter('Coefficients',mfsig);
ymf = mf([y0 y1]);
fdelay = (nsamp-1)/fs;
t = (0:size(ymf,1)-1)/fs - fdelay;
c = physconst('LightSpeed');
plot(c*t/2,mag2db(abs(ymf)))
ylim([-200 -50])
xlabel('Range (m)')
ylabel('Magnitude (dB)')
ax = axis;
axis([0,100,ax(3),ax(4)])
grid
legend('First pulse','Second pulse')
```





Compute the difference in range between the maxima of the two pulses.

```
[maxy,idx] = max(abs(ymf));
dpeaks = t(1,idx(2)) - t(1,idx(1));
drng = c*dpeaks/2
```

```
drng = 4.9965
```

The range difference is 5 m, as expected given the bicyclist speed.

## Input Arguments

### **bicyclist** – Bicyclist target

backscatterBicyclist object

Bicyclist, specified as a backscatterBicyclist object.

### **X** – Incident radar signals

complex-valued  $M$ -by- $N$  matrix

Incident radar signals on each bicyclist scatterer, specified as a complex-valued  $M$ -by- $N$  matrix.  $M$  is the number of samples in the signal.  $N$  is the number of point scatterers on the bicyclist and is determined partly from the number of spokes in each wheel,  $N_{ws}$ . See “Bicycle Scatterer Indices” on page 4-458 for the column representing the incident signal at each scatterer.

The size of the first dimension of the input matrix can vary to simulate a changing signal length. A size change can occur, for example, in the case of a pulse waveform with variable pulse repetition frequency.

Data Types: `double`

Complex Number Support: Yes

### **ang** — Directions of incident signals

real-valued 2-by- $P$  matrix

Directions of incident signals on the bicyclist scatterers, specified as a real-valued 2-by- $N$  matrix.  $N$  equals the number of columns in  $X$ . Each column of  $Ang$  specifies the incident direction of the signal to a scatterer taking the form of an azimuth-elevation pair,  $[AzimuthAngle;ElevationAngle]$ . Units are in degrees. See “Bicycle Scatterer Indices” on page 4-458 for the column representing the incident direction at each scatterer.

Data Types: `double`

## Output Arguments

### **Y** — Total reflected radar signals

complex-valued  $M$ -by-1 column vector

Total reflected radar signals, returned as a complex-valued  $M$ -by-1 column vector.  $M$  equals the number of samples in the input signal,  $X$ .

Data Types: `double`

Complex Number Support: Yes

## More About

### Bicycle Scatterer Indices

Bicyclist scatterer indices define which columns in the scatterer position or velocity matrices contain the position and velocity data for a specific scatterer. For example, column 92 of `bpos` specifies the 3-D position of one of the scatterers on a pedal.

The wheel scatterers are equally divided between the wheels. You can determine the total number of wheel scatterers,  $N$ , by subtracting 113 from the output of the `getNumScatterers` function. The number of scatterers per wheel is  $N_{sw} = N/2$ .

### Bicyclist Scatterer Indices

Bicyclist Component	Bicyclist Scatterer Index
Frame and rider	1 ... 90
Pedals	91 ... 99
Rider legs	100 ... 113
Front wheel	114 ... 114 + $N_{sw}$ - 1
Rear wheel	114 + $N_{sw}$ ... 114 + $N$ - 1

## Algorithms

### Radar Cross-Section

The value of the radar cross-section (RCS) of a scatterer generally depends upon the incident angle of the reflected radiation. The `backscatterBicyclist` object uses a simplified RCS model: the RCS pattern of an individual scatterer equals the total bicyclist pattern divided by the number of scatterers. The value of the RCS is computed from the RCS pattern evaluated at an average over all scatterers of the azimuth and elevation incident angles. Therefore, the RCS value is the same for all scatterers. You can specify the RCS pattern using the `RCSPattern` property of the `backscatterBicyclist` object or use the default value.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

#### See Also

`getNumScatterers` | `move` | `plot`

**Introduced in R2021a**

## backscatterPedestrian

Backscatter radar signals from pedestrian

### Description

`backscatterPedestrian` creates an object that simulates signals reflected from a walking pedestrian. The pedestrian walking model coordinates the motion of 16 body segments to simulate natural motion. The model also simulates the radar reflectivity of each body segment. From this model, you can obtain the position and velocity of each segment and the total backscattered radiation as the body moves.

After creating the pedestrian, you can move the pedestrian by calling the `move` object function. To obtain the reflected signal, call the `reflect` object function. You can plot the instantaneous position of the body segments using the `plot` object function.

### Creation

#### Syntax

```
pedestrian = backscatterPedestrian
pedestrian = backscatterPedestrian(Name,Value,...)
```

#### Description

`pedestrian = backscatterPedestrian` creates a pedestrian target model object, `pedestrian`. The pedestrian model includes 16 body segments - left and right feet, left and right lower legs, left and right upper legs, left and right hip, left and right lower arms, left and right upper arms, left and right shoulders, neck, and head.

`pedestrian = backscatterPedestrian(Name,Value,...)` creates a pedestrian object, `pedestrian`, with each specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as `(Name1,Value1,...,NameN,ValueN)`. Any unspecified properties take default values. For example,

```
pedestrian = backscatterPedestrian( ...
    'Height',2,'WalkingSpeed',0.5, ...
    'InitialPosition',[0;0;0],'InitialHeading',90);
```

models a two-meter tall woman or man moving along the positive *y*-axis at one-half meter per second.

### Properties

#### Height — Height of pedestrian

1.65 (default) | positive scalar

Height of pedestrian, specified as a positive scalar. Units are in meters.

Data Types: double

**WalkingSpeed — Walking speed of pedestrian**

1.4 times pedestrian height (default) | non-negative scalar

Walking speed of pedestrian, specified as a non-negative scalar. The motion model limits the walking speed to 1.4 times the pedestrian height set in the `Height` property. Units are in meters per second.

Data Types: double

**PropagationSpeed — Signal propagation speed**

physconst('LightSpeed') (default) | positive scalar

Signal propagation speed, specified as a positive scalar. Units are in meters per second. The default propagation speed is the value returned by `physconst('LightSpeed')`. See `physconst` for more information.

Example: 3e8

Data Types: double

**OperatingFrequency — Carrier frequency**

300e6 (default) | positive scalar

Carrier frequency of narrowband incident signals, specified as a positive scalar. Units are in Hz.

Example: 1e9

Data Types: double

**InitialPosition — Initial position of pedestrian**

[0;0;0] (default) | 3-by-1 real-valued vector

Initial position of the pedestrian, specified as a 3-by-1 real-valued vector in the form of `[x;y;z]`. Units are in meters.

Data Types: double

**InitialHeading — Initial heading of pedestrian**

0 (default) | scalar

Initial heading of pedestrian, specified as a scalar. Heading is measured in the  $xy$ -plane from the  $x$ -axis towards  $y$ -axis. Units are in degrees.

Data Types: double

**Object Functions****Specific to This Object**

<code>move</code>	Position and velocity of walking pedestrian
<code>reflect</code>	Reflected signal from walking pedestrian
<code>plot</code>	Display stick figure showing the positions of all body segments of pedestrian

**Common to All System Objects**

<code>step</code>	Run System object algorithm
<code>release</code>	Release resources and allow changes to System object property values and input characteristics

reset     Reset internal states of System object

## Examples

### Reflected Signal from Moving Pedestrian

Compute the reflected radar signal from a pedestrian moving along the  $x$ -axis away from the origin. The radar operates at 24 GHz and is located at the origin. The pedestrian is initially 100 meters from the radar. Transmit a linear FM waveform having a 300 MHz bandwidth. The reflected signal is captured at the moment the pedestrian starts to move and at two seconds into the motion.

Create a linear FM waveform and a free space channel to propagate the waveform.

```
c = physconst('Lightspeed');
bw = 300.0e6;
fs = bw;
fc = 24.0e9;
wav = phased.LinearFMWaveform('SampleRate',fs,'SweepBandwidth',bw);
x = wav();
channel = phased.FreeSpace('OperatingFrequency',fc,'SampleRate',fs, ...
    'TwoWayPropagation',true);
```

Create the pedestrian object. Set the initial position of the pedestrian to 100 m on the  $x$ -axis with initial heading along the positive  $x$ -direction. The pedestrian height is 1.8 m and the pedestrian is walking at 0.5 meters per second.

```
pedest = backscatterPedestrian('Height',1.8, ...
    'OperatingFrequency',fc,'InitialPosition',[100;0;0], ...
    'InitialHeading',0,'WalkingSpeed',0.5);
```

The first call to the `move` function returns the initial position, initial velocity, and initial orientation of all body segments and then advances the pedestrian motion two seconds ahead.

```
[bpos,bvel,bpax] = move(pedest,2,0);
```

Transmit the first pulse to the pedestrian. Create 16 replicas of the signal and propagate them to the positions of the pedestrian body segments. Use the `rangeangle` function to compute the arrival angle of each replica at the corresponding body segment. Then use the `reflect` function to return the coherent sum of all the reflected signals from the body segments at the pedestrian initial position.

```
radarpos = [0;0;0];
xp = channel(repmat(x,1,16),radarpos,bpos,[0;0;0],bvel);
[~,ang] = rangeangle(radarpos,bpos,bpax);
y0 = reflect(pedest,xp,ang);
```

Obtain the position, velocity, and orientation of each body segment then advance the pedestrian motion another two seconds.

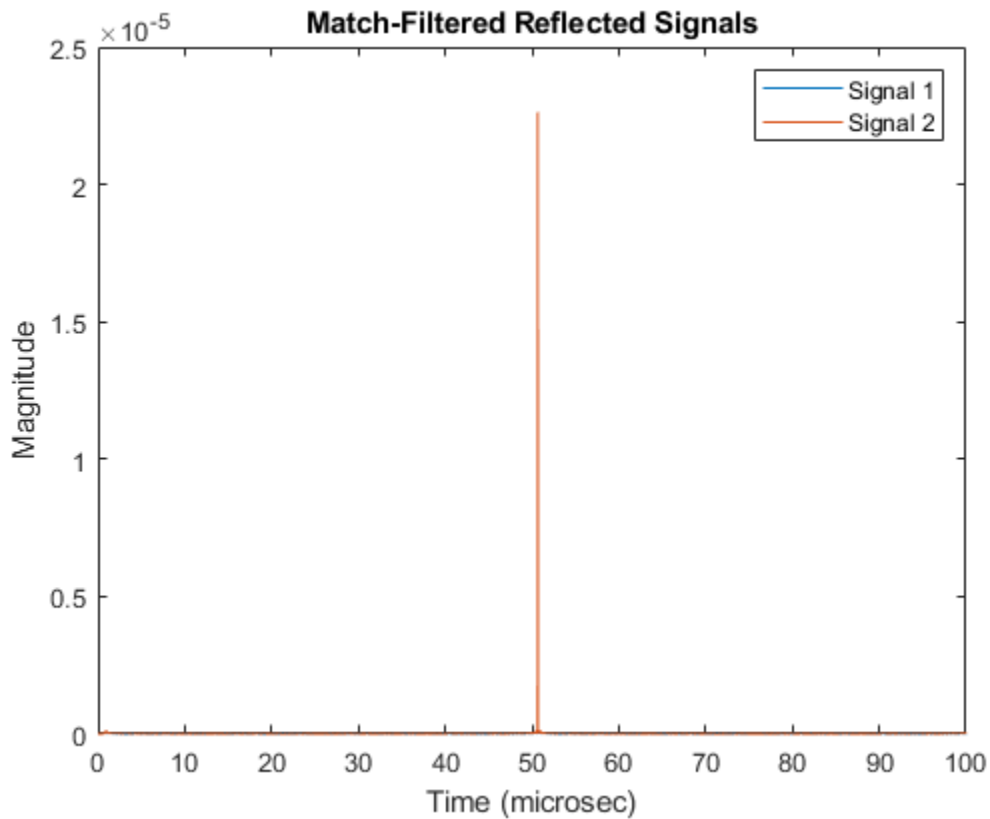
```
[bpos,bvel,bpax] = move(pedest,2,0);
```

Transmit and propagate the second pulse to the new position of the pedestrian.

```
radarpos = [0;0;0];
xp = channel(repmat(x,1,16),radarpos,bpos,[0;0;0],bvel);
[~,ang] = rangeangle(radarpos,bpos,bpax);
y1 = reflect(pedest,xp,ang);
```

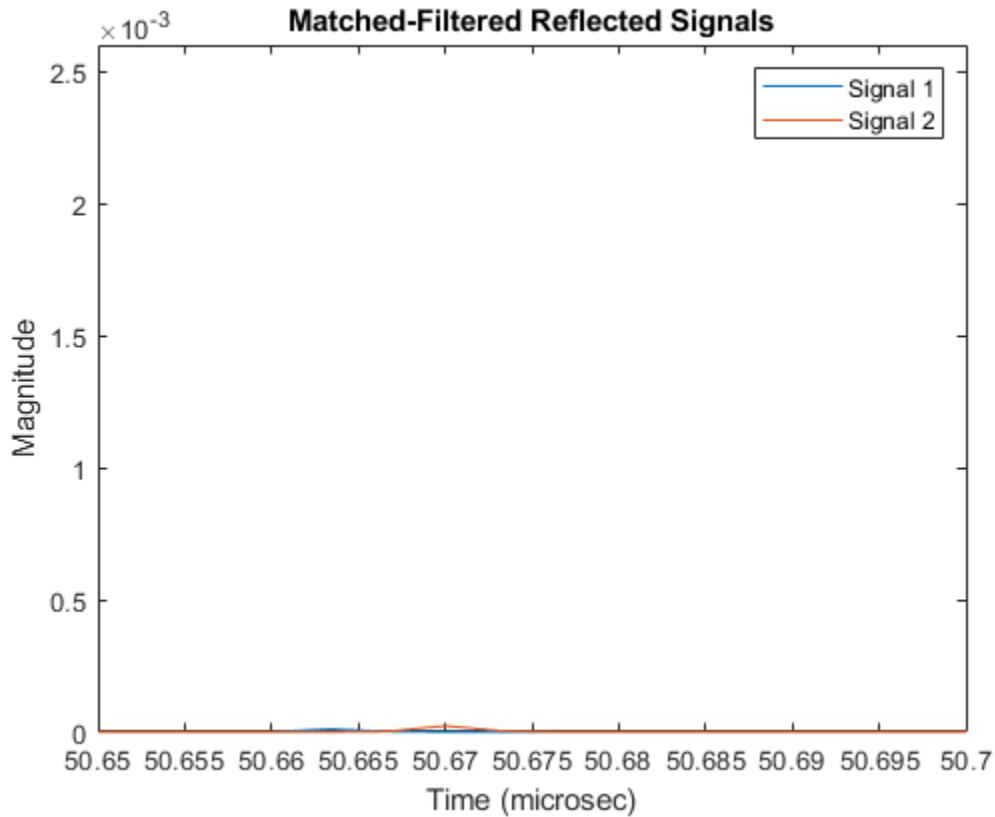
Match-filter and plot both of the reflected pulses. The plot shows the increased delay of the matched filter output as the pedestrian walks away.

```
filter = phased.MatchedFilter('Coefficients',getMatchedFilter(wav));
ymf = filter([y0 y1]);
t = (0:size(ymf,1)-1)/fs;
plot(t*1e6,abs(ymf))
xlabel('Time (microsec)')
ylabel('Magnititude')
title('Match-Filtered Reflected Signals')
legend('Signal 1','Signal 2')
```



Zoom in and show the time delays for each signal.

```
plot(t*1e6,abs(ymf))
xlabel('Time (microsec)')
ylabel('Magnititude')
title('Matched-Filtered Reflected Signals')
axis([50.65 50.7 0 .0026])
legend('Signal 1','Signal 2')
```



### Plot Arm Motion of Walking Pedestrian

Create a pedestrian object. Set the initial position of the pedestrian to 100 m on the x-axis with initial heading along the positive x-direction. The pedestrian height is 1.8 m and the pedestrian is walking at 1.5 meters per second.

```
fc = 24.0e9;
pedest = backscatterPedestrian( 'Height',1.8, ...
    'OperatingFrequency',fc,'InitialPosition',[100;0;0], ...
    'InitialHeading',0,'WalkingSpeed',1.5);
```

Obtain and plot the detailed motion of the right and left lower arms of the pedestrian by capturing their positions every 1/10th of a second.

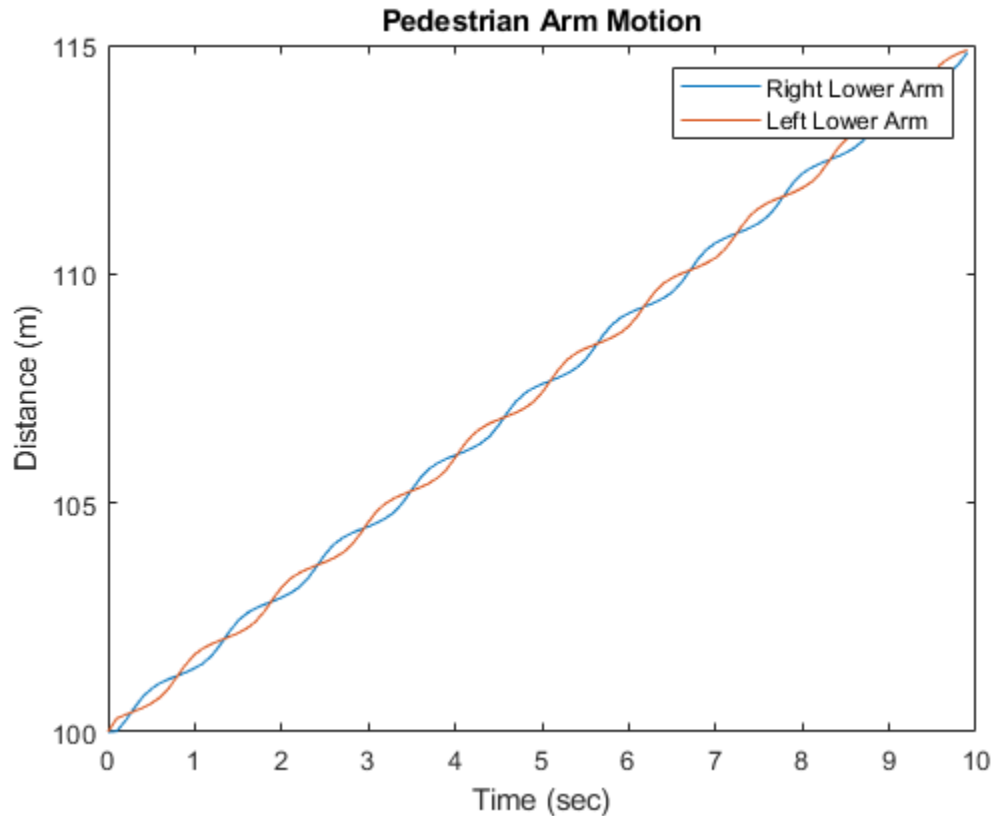
```
blla = zeros(3,100);
brla = blla;
t = zeros(1,100);
T = .1;
for k = 1:100
    [bppos,bpvel,bpax] = move(pedest,T,0);
    blla(:,k) = bppos(:,9);
    brla(:,k) = bppos(:,10);
    t(k) = T*(k-1);
end
plot(t,brla(1,:),t,blla(1,:))
```



```

title('Pedestrian Arm Motion')
xlabel('Time (sec)')
ylabel('Distance (m)')
legend('Right Lower Arm','Left Lower Arm')

```



### Plot Pedestrian Motion

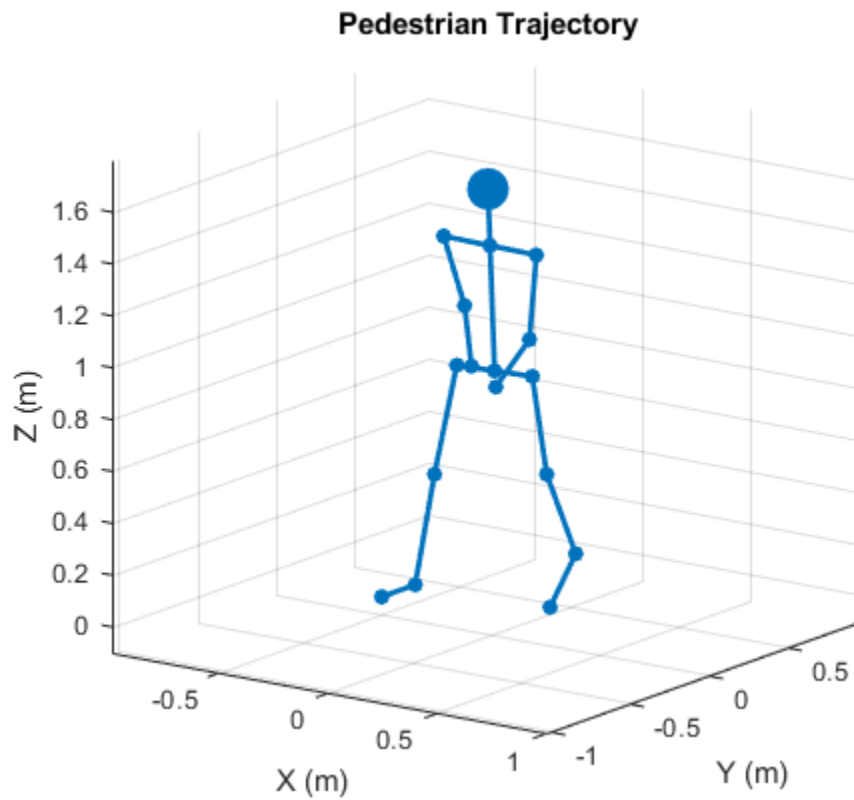
Display the motion of a pedestrian walking a square path. Create the pedestrian using a `backscatterPedestrian` object with default values except for height which is 1.7 meters. Advance and display the pedestrian position every 3 milliseconds. First, the pedestrian moves along the positive x-axis, then along the positive y-axis, along the negative x-axis, and finally along the negative y-axis to return to the starting point.

```

ped = backscatterPedestrian('Height',1.7);
dt = 0.003;
N = 3600;
for m = 1:N
    if (m < N/4)
        angstep = 0.0;
    end
    if (m >= N/4)
        angstep = 90.0;
    end
    if (m >= N/2)

```

```
        angstep = 180.0;
    end
    if (m >= 3*N/4)
        angstep = 270.0;
    end
    move(ped,dt,angstep);
    plot(ped)
end
```



## References

- [1] Victor Chen, *The Micro-Doppler Effect in Radar*, Artech House, 2011.
- [2] Ronan Boulic, Nadia Magnenat-Thalmann, Daniel Thalmann, A Global Human Walking Model with Real-time Kinematic Personification, *The Visual Computer: International Journal of Computer Graphics*, Vol. 6, Issue 6, Dec 1990.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

**See Also**

move | reflect | plot | phased.BackscatterSonarTarget |  
phased.BackscatterRadarTarget | phased.WidebandBackscatterRadarTarget |  
phased.RadarTarget

**Introduced in R2019a**

## move

Position and velocity of walking pedestrian

### Syntax

```
[BPPOS, BPVEL, BPAX] = move(pedestrian, T, ANGH)
```

### Description

[BPPOS, BPVEL, BPAX] = move(pedestrian, T, ANGH) returns the position, BPPOS, velocity, BPVEL, and orientation axes, BPAX, of body segments of a moving pedestrian. The object then simulates the walking motion for the next duration, specified in T. ANGH specifies the current heading angle.

### Examples

#### Reflected Signal from Moving Pedestrian

Compute the reflected radar signal from a pedestrian moving along the  $x$ -axis away from the origin. The radar operates at 24 GHz and is located at the origin. The pedestrian is initially 100 meters from the radar. Transmit a linear FM waveform having a 300 MHz bandwidth. The reflected signal is captured at the moment the pedestrian starts to move and at two seconds into the motion.

Create a linear FM waveform and a free space channel to propagate the waveform.

```
c = physconst('Lightspeed');
bw = 300.0e6;
fs = bw;
fc = 24.0e9;
wav = phased.LinearFMWaveform('SampleRate', fs, 'SweepBandwidth', bw);
x = wav();
channel = phased.FreeSpace('OperatingFrequency', fc, 'SampleRate', fs, ...
    'TwoWayPropagation', true);
```

Create the pedestrian object. Set the initial position of the pedestrian to 100 m on the  $x$ -axis with initial heading along the positive  $x$ -direction. The pedestrian height is 1.8 m and the pedestrian is walking at 0.5 meters per second.

```
pedest = backscatterPedestrian('Height', 1.8, ...
    'OperatingFrequency', fc, 'InitialPosition', [100; 0; 0], ...
    'InitialHeading', 0, 'WalkingSpeed', 0.5);
```

The first call to the `move` function returns the initial position, initial velocity, and initial orientation of all body segments and then advances the pedestrian motion two seconds ahead.

```
[bppos, bpvel, bpax] = move(pedest, 2, 0);
```

Transmit the first pulse to the pedestrian. Create 16 replicas of the signal and propagate them to the positions of the pedestrian body segments. Use the `rangeangle` function to compute the arrival angle of each replica at the corresponding body segment. Then use the `reflect` function to return the coherent sum of all the reflected signals from the body segments at the pedestrian initial position.

```

radarpos = [0;0;0];
xp = channel(repmat(x,1,16), radarpos, bppos, [0;0;0], bpvel);
[~,ang] = rangeangle(radarpos, bppos, bpax);
y0 = reflect(pedest, xp, ang);

```

Obtain the position, velocity, and orientation of each body segment then advance the pedestrian motion another two seconds.

```

[bppos, bpvel, bpax] = move(pedest, 2, 0);

```

Transmit and propagate the second pulse to the new position of the pedestrian.

```

radarpos = [0;0;0];
xp = channel(repmat(x,1,16), radarpos, bppos, [0;0;0], bpvel);
[~,ang] = rangeangle(radarpos, bppos, bpax);
y1 = reflect(pedest, xp, ang);

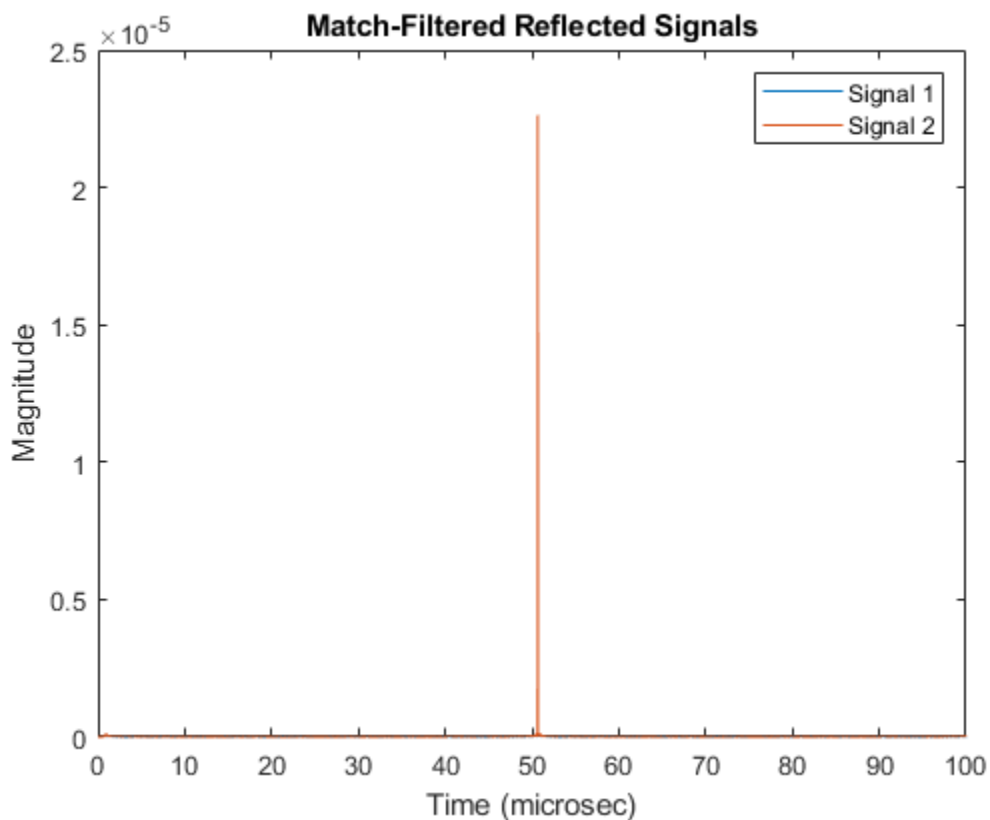
```

Match-filter and plot both of the reflected pulses. The plot shows the increased delay of the matched filter output as the pedestrian walks away.

```

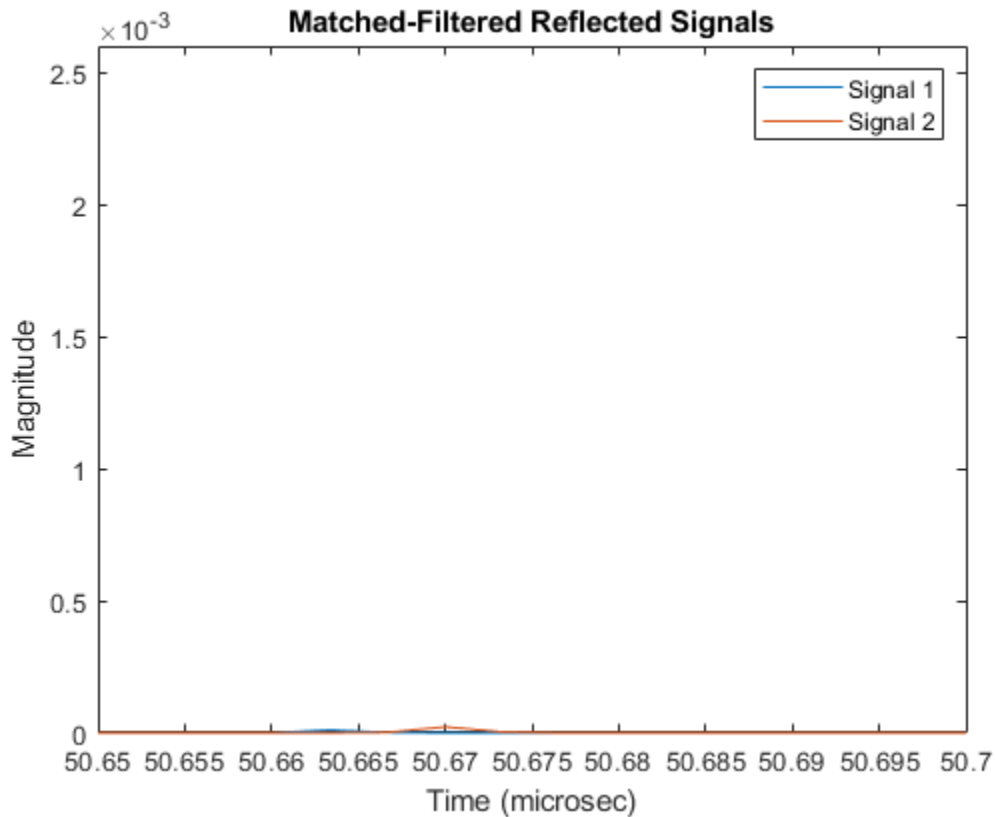
filter = phased.MatchedFilter('Coefficients', getMatchedFilter(wav));
ymf = filter([y0 y1]);
t = (0:size(ymf,1)-1)/fs;
plot(t*1e6, abs(ymf))
xlabel('Time (microsec)')
ylabel('Magnitude')
title('Match-Filtered Reflected Signals')
legend('Signal 1', 'Signal 2')

```



Zoom in and show the time delays for each signal.

```
plot(t*1e6,abs(ymf))
xlabel('Time (microsec)')
ylabel('Magnitude')
title('Matched-Filtered Reflected Signals')
axis([50.65 50.7 0 .0026])
legend('Signal 1','Signal 2')
```



## Input Arguments

### pedestrian — Pedestrian target

backscatterPedestrian object

Pedestrian target model, specified as a backscatterPedestrian object.

### T — Duration of next walking interval

scalar

Duration of next walking interval, specified as a positive scalar. Units are in seconds.

Example: 0.75

Data Types: double

### ANGH — Pedestrian heading

scalar

Heading of the pedestrian, specified as a scalar. Heading is measured in the  $xy$ -plane from the  $x$ -axis towards the  $y$ -axis. Units are in degrees.

Example: -34

Data Types: double

## Output Arguments

### **BPP0S — Positions of body segments**

real-valued 3-by-16 matrix

Positions of body segments, returned as a real-valued 3-by-16 matrix. Each column represents the Cartesian position,  $[x; y; z]$ , of one of 16 body segments. Units are in meters. See “Body Segment Indices” on page 2-16 for the column representing the position of each body segment.

Data Types: double

### **BPVEL — Velocity of body segments**

real-valued 3-by-16 matrix

Velocity of body segments, returned as a real-valued 3-by-16 matrix. Each column represents the Cartesian velocity vector,  $[vx; vy; vz]$ , of one of 16 body segments. Units are in meters per second. See “Body Segment Indices” on page 2-16 for the column representing the velocity of each body segment.

Data Types: double

### **BPAX — Orientation of body segments**

real-valued 3-by-3-by-16 array

Orientation axes of body segments, returned as a real-valued 3-by-3-by-16 array. Each page represents the 3-by-3 orientation axes of one of 16 body segments. Units are dimensionless. See “Body Segment Indices” on page 2-16 for the page representing the orientation of each body segment.

Data Types: double

## More About

### **Body Segment Indices**

Body segment indices define which columns in BPP0S and BPVEL contain the position and velocity data for a specific body segment. The indices also point to the page of BPAX containing the orientation matrix for a specific body segment. For example, column three of BPP0S contains the 3-D position of the left lower leg. Page three of BPAX contains the orientation matrix of the left lower leg.

Body Segment	Index	
Left foot	1	
Right foot	2	
Left lower leg	3	
Right lower leg	4	
Left upper leg	5	
Right upper leg	6	
Left hip	7	
Right hip	8	
Left lower arm	9	
Right lower arm	10	
Left upper arm	11	
Right upper arm	12	
Left shoulder	13	
Right shoulder	14	
Head	15	
Torso	16	

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`backscatterPedestrian` | `reflect` | `plot`

**Introduced in R2019a**



# plot

Display stick figure showing the positions of all body segments of pedestrian

## Syntax

```
plot(pedestrian)
fhndl = plot(pedestrian)
```

## Description

`plot(pedestrian)` displays a stick figure showing the positions of all body segments of a pedestrian. The lines of the figure represent body segments while the dots represent the joints connecting body segments.

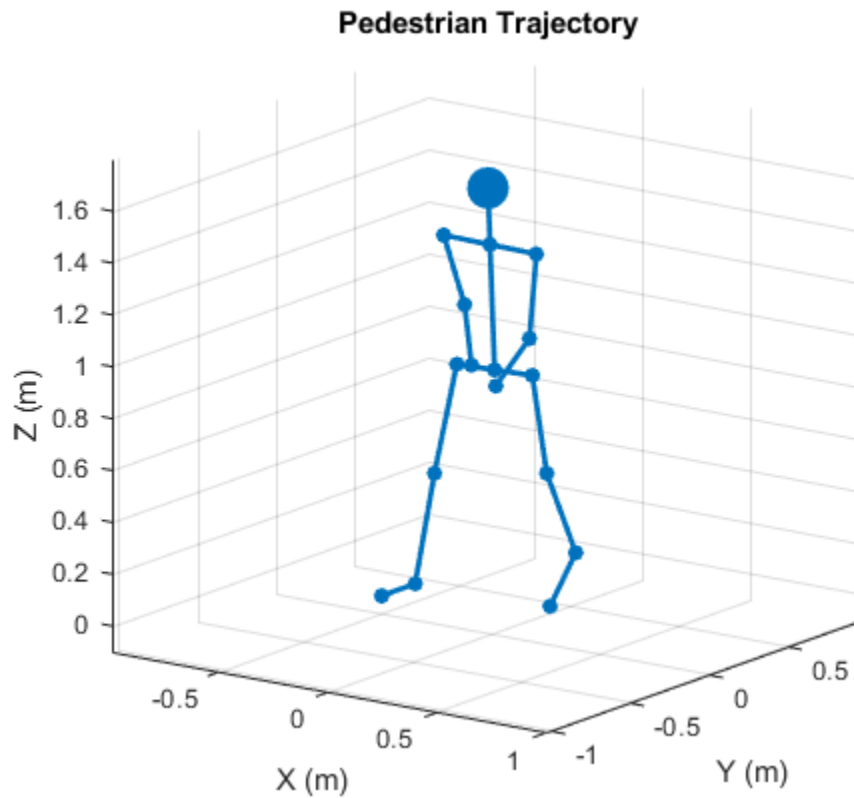
`fhndl = plot(pedestrian)` returns the figure handle of the display window.

## Examples

### Plot Pedestrian Motion

Display the motion of a pedestrian walking a square path. Create the pedestrian using a `backscatterPedestrian` object with default values except for height which is 1.7 meters. Advance and display the pedestrian position every 3 milliseconds. First, the pedestrian moves along the positive  $x$ -axis, then along the positive  $y$ -axis, along the negative  $x$ -axis, and finally along the negative  $y$ -axis to return to the starting point.

```
ped = backscatterPedestrian('Height',1.7);
dt = 0.003;
N = 3600;
for m = 1:N
    if (m < N/4)
        angstep = 0.0;
    end
    if (m >= N/4)
        angstep = 90.0;
    end
    if (m >= N/2)
        angstep = 180.0;
    end
    if (m >= 3*N/4)
        angstep = 270.0;
    end
    move(ped,dt,angstep);
    plot(ped)
end
```



## Input Arguments

### **pedestrian** – Pedestrian target

backscatterPedestrian object

Pedestrian target, specified as a backscatterPedestrian object.

## Output Arguments

### **fhndl** – figure handle

figure handle

Figure handle of plot window

## See Also

backscatterPedestrian | move | reflect

## Topics

“Reflected Signal from Moving Pedestrian” on page 4-462

**Introduced in R2019b**

# reflect

Reflected signal from walking pedestrian

## Syntax

```
Y = reflect(pedestrian,X,ANG)
```

## Description

`Y = reflect(pedestrian,X,ANG)` returns the reflected signal, `Y`, from incident signals, `X`, on a pedestrian. The reflected signal is the sum of signals from all body segments. `ANG` defines the directions of the incident and reflected signals with respect to the body segments.

## Examples

### Reflected Signal from Moving Pedestrian

Compute the reflected radar signal from a pedestrian moving along the  $x$ -axis away from the origin. The radar operates at 24 GHz and is located at the origin. The pedestrian is initially 100 meters from the radar. Transmit a linear FM waveform having a 300 MHz bandwidth. The reflected signal is captured at the moment the pedestrian starts to move and at two seconds into the motion.

Create a linear FM waveform and a free space channel to propagate the waveform.

```
c = physconst('Lightspeed');
bw = 300.0e6;
fs = bw;
fc = 24.0e9;
wav = phased.LinearFMWaveform('SampleRate',fs,'SweepBandwidth',bw);
x = wav();
channel = phased.FreeSpace('OperatingFrequency',fc,'SampleRate',fs, ...
    'TwoWayPropagation',true);
```

Create the pedestrian object. Set the initial position of the pedestrian to 100 m on the  $x$ -axis with initial heading along the positive  $x$ -direction. The pedestrian height is 1.8 m and the pedestrian is walking at 0.5 meters per second.

```
pedest = backscatterPedestrian('Height',1.8, ...
    'OperatingFrequency',fc,'InitialPosition',[100;0;0], ...
    'InitialHeading',0,'WalkingSpeed',0.5);
```

The first call to the `move` function returns the initial position, initial velocity, and initial orientation of all body segments and then advances the pedestrian motion two seconds ahead.

```
[bppos,bpvel,bpax] = move(pedest,2,0);
```

Transmit the first pulse to the pedestrian. Create 16 replicas of the signal and propagate them to the positions of the pedestrian body segments. Use the `rangeangle` function to compute the arrival angle of each replica at the corresponding body segment. Then use the `reflect` function to return the coherent sum of all the reflected signals from the body segments at the pedestrian initial position.

```

radarpos = [0;0;0];
xp = channel(repmat(x,1,16), radarpos, bppos, [0;0;0], bpvel);
[~,ang] = rangeangle(radarpos, bppos, bpax);
y0 = reflect(pedest, xp, ang);

```

Obtain the position, velocity, and orientation of each body segment then advance the pedestrian motion another two seconds.

```
[bppos, bpvel, bpax] = move(pedest, 2, 0);
```

Transmit and propagate the second pulse to the new position of the pedestrian.

```

radarpos = [0;0;0];
xp = channel(repmat(x,1,16), radarpos, bppos, [0;0;0], bpvel);
[~,ang] = rangeangle(radarpos, bppos, bpax);
y1 = reflect(pedest, xp, ang);

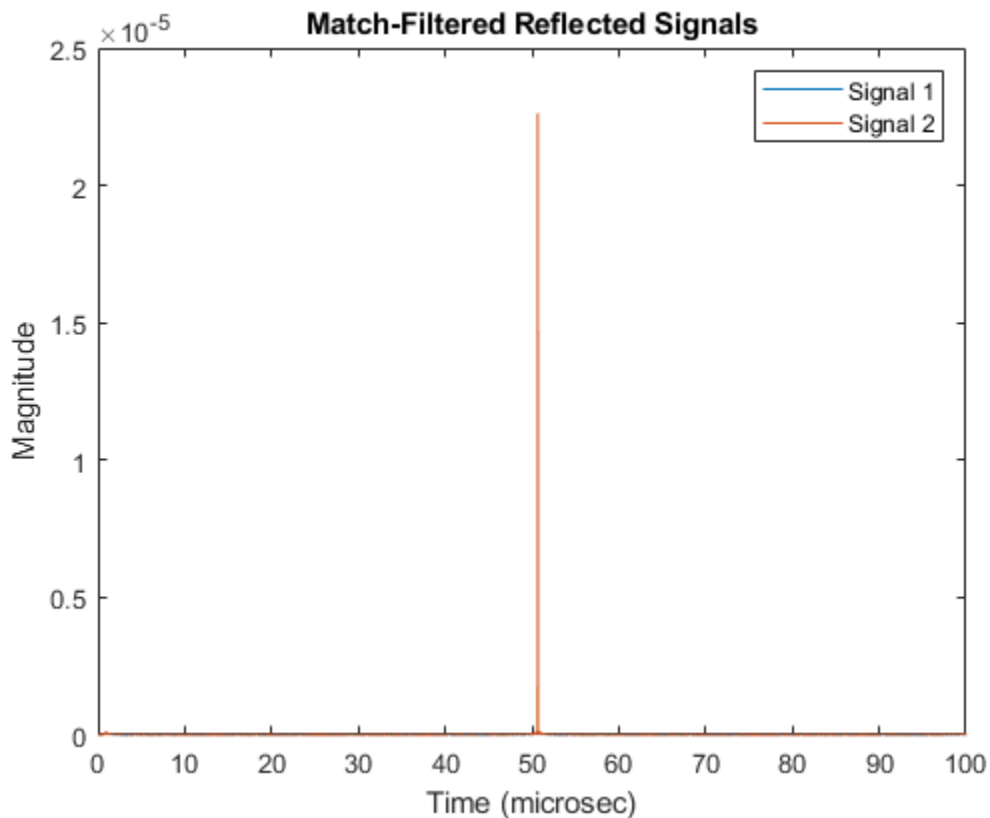
```

Match-filter and plot both of the reflected pulses. The plot shows the increased delay of the matched filter output as the pedestrian walks away.

```

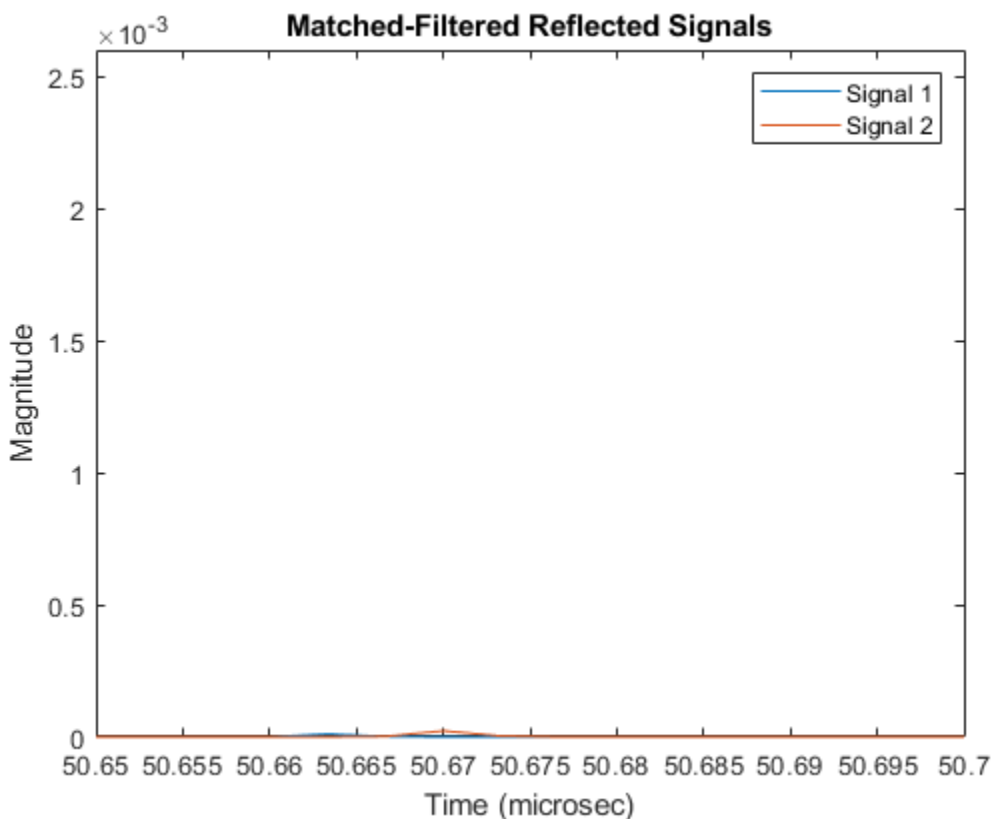
filter = phased.MatchedFilter('Coefficients', getMatchedFilter(wav));
ymf = filter([y0 y1]);
t = (0:size(ymf,1)-1)/fs;
plot(t*1e6, abs(ymf))
xlabel('Time (microsec)')
ylabel('Magnitude')
title('Match-Filtered Reflected Signals')
legend('Signal 1', 'Signal 2')

```



Zoom in and show the time delays for each signal.

```
plot(t*1e6,abs(ymf))
xlabel('Time (microsec)')
ylabel('Magnitude')
title('Matched-Filtered Reflected Signals')
axis([50.65 50.7 0 .0026])
legend('Signal 1','Signal 2')
```



## Input Arguments

### pedestrian — Pedestrian target

backscatterPedestrian object

Pedestrian target model, specified as a backscatterPedestrian object.

### X — Incident radar signals

complex-valued  $M$ -by-16 matrix

Incident radar signals on each body segment, specified as a complex-valued  $M$ -by-16 matrix.  $M$  is the number of samples in the signal. See “Body Segment Indices” on page 4-478 for the column representing the incident signal at each body segment.

Data Types: double

Complex Number Support: Yes

**ANG — Directions of incident signals**

real-valued 2-by-16 matrix

Directions of incident signals on the body segments, specified as a real-valued 2-by-16 matrix. Each column of ANG specifies the incident direction of the signal to the corresponding body part. Each column takes the form of an azimuth-elevation pair, [AzimuthAngle;ElevationAngle]. Units are in degrees. See “Body Segment Indices” on page 4-478 for the column representing the incident direction at each body segment.

Data Types: double

**Output Arguments****Y — Combined reflected radar signals**complex-valued  $M$ -by-1 column vector

Combined reflected radar signals, returned as a complex-valued  $M$ -by-1 column vector.  $M$  equals the same number of samples as in the input signal, X.

Data Types: double

Complex Number Support: Yes

**More About****Body Segment Indices**

Body segment indices define which columns in X and ANG contain the data for a specific body segment. For example, column 3 of X contains sample data for the left lower leg. Column 3 of ANG contains the arrival angle of the signal at the left lower leg.

Body Segment	Index	
Left foot	1	
Right foot	2	
Left lower leg	3	
Right lower leg	4	
Left upper leg	5	
Right upper leg	6	
Left hip	7	
Right hip	8	
Left lower arm	9	
Right lower arm	10	
Left upper arm	11	
Right upper arm	12	
Left shoulder	13	
Right shoulder	14	
Head	15	
Torso	16	

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

`backscatterPedestrian` | `move` | `plot`

**Introduced in R2019a**

## constantGammaClutter

Simulate constant gamma clutter

### Description

The `constantGammaClutter` object simulates clutter.

To compute the clutter return:

- 1 Define and set up your clutter simulator. See “Construction” on page 4-480.
- 2 Call `step` to simulate the clutter return for your system according to the properties of `constantGammaClutter`. The behavior of `step` is specific to each object in the toolbox.

The clutter simulation that `constantGammaClutter` provides is based on these assumptions:

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

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

### Construction

`H = constantGammaClutter` creates a constant gamma clutter simulation System object, `H`. This object simulates the clutter return of a monostatic radar system using the constant gamma model.

`H = constantGammaClutter(Name, Value)` creates a constant gamma clutter simulation object, `H`, with additional options specified by one or more `Name, Value` pair arguments. `Name` is a property name on page 4-480, and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Properties

#### Sensor

Handle of sensor



Specify the sensor as an antenna element object or as an array object whose `Element` property value is an antenna element object. If the sensor is an array, it can contain subarrays.

**Default:** `phased.ULA` with default property values

### **Gamma**

Terrain gamma value

Specify the  $\gamma$  value used in the constant  $\gamma$  clutter model, as a scalar in decibels. The  $\gamma$  value depends on both terrain type and the operating frequency.

**Default:** 0

### **EarthModel**

Earth model

Specify the earth model used in clutter simulation as one of | `'Flat'` | `'Curved'` |. When you set this property to `'Flat'`, the earth is assumed to be a flat plane. When you set this property to `'Curved'`, the earth is assumed to be a sphere.

**Default:** `'Flat'`

### **ClutterMinRange**

Minimum range of clutter region (m)

Minimum range at which to computer clutter returns, specified as a positive scalar. The minimum range must be nonnegative. This value is ignored if it less than the value of `PlatformHeight`. Units are in meters.

**Default:** 0

### **ClutterMaxRange**

Maximum range of clutter region (m)

Specify the maximum range at which to compute clutter returns. for the clutter simulation as a positive scalar. The maximum range must be greater than the value specified in the `PlatformHeight` property. Units are in meters.

**Default:** 5000

### **ClutterAzimuthCenter**

Azimuth center of clutter region (deg)

The azimuth angle in the ground plane about which clutter patches are generated. Patches are generated symmetrically about this angle.

**Default:** 0

### **ClutterAzimuthSpan**

Azimuth span of clutter region (deg)

Specify the coverage in azimuth (in degrees) of the clutter region as a positive scalar. The clutter simulation covers a region having the specified azimuth span, symmetric around `ClutterAzimuthCenter`. Typically, all clutter patches have their azimuth centers within the region, but the `PatchAzimuthSpan` value can cause some patches to extend beyond the region.

**Default:** 60

### **PatchAzimuthSpan**

Azimuth span of clutter patches (deg)

Specify the azimuth span (in degrees) of each clutter patch as a positive scalar.

**Default:** 1

### **CoherenceTime**

Clutter coherence time

Specify the coherence time in seconds for the clutter simulation as a positive scalar. After the coherence time elapses, the `step` method updates the random numbers it uses for the clutter simulation at the next pulse. A value of `inf` means the random numbers are never updated.

**Default:** `inf`

### **PropagationSpeed**

Signal propagation speed

Specify the propagation speed of the signal, in meters per second, as a positive scalar.

**Default:** Speed of light

### **SampleRate**

Sample rate

Specify the sample rate, in hertz, as a positive scalar. The default value corresponds to 1 MHz.

**Default:** `1e6`

### **PRF**

Pulse repetition frequency

Pulse repetition frequency, *PRF*, specified as a scalar or a row vector. Units are in Hz. The pulse repetition interval, *PRI*, is the inverse of the pulse repetition frequency, *PRF*. The *PRF* must satisfy these restrictions:

- The product of *PRF* and *PulseWidth* must be less than or equal to one. This condition expresses the requirement that the pulse width is less than one pulse repetition interval. For the phase-coded waveform, the pulse width is the product of the chip width and number of chips.
- The ratio of sample rate to any element of *PRF* must be an integer. This condition expresses the requirement that the number of samples in one pulse repetition interval is an integer.

You can select the value of *PRF* using property settings alone or using property settings in conjunction with the `prfidx` input argument of the `step` method.

- When `PRFSelectionInputPort` is `false`, you set the *PRF* using properties only. You can
  - implement a constant *PRF* by specifying `PRF` as a positive real-valued scalar.
  - implement a staggered *PRF* by specifying `PRF` as a row vector with positive real-valued entries. Then, each call to the `step` method uses successive elements of this vector for the *PRF*. If the last element of the vector is reached, the process continues cyclically with the first element of the vector.
- When `PRFSelectionInputPort` is `true`, you can implement a selectable *PRF* by specifying `PRF` as a row vector with positive real-valued entries. But this time, when you execute the `step` method, select a *PRF* by passing an argument specifying an index into the *PRF* vector.

In all cases, the number of output samples is fixed when you set the `OutputFormat` property to `'Samples'`. When you use a varying *PRF* and set the `OutputFormat` property to `'Pulses'`, the number of samples can vary.

**Default:** `10e3`

### **PRFSelectionInputPort**

Enable PRF selection input

Enable the PRF selection input, specified as `true` or `false`. When you set this property to `false`, the `step` method uses the values set in the `PRF` property. When you set this property to `true`, you pass an index argument into the `step` method to select a value from the `PRF` vector.

**Default:** `false`

### **OutputFormat**

Output signal format

Specify the format of the output signal as one of `'Pulses'` | `'Samples'` |. When you set the `OutputFormat` property to `'Pulses'`, the output of the `step` method is in the form of multiple pulses. In this case, the number of pulses is the value of the `NumPulses` property.

When you set the `OutputFormat` property to `'Samples'`, the output of the `step` method is in the form of multiple samples. In this case, the number of samples is the value of the `NumSamples` property. In staggered PRF applications, you might find the `'Samples'` option more convenient because the `step` output always has the same matrix size.

**Default:** `'Pulses'`

### **NumPulses**

Number of pulses in output

Specify the number of pulses in the output of the `step` method as a positive integer. This property applies only when you set the `OutputFormat` property to `'Pulses'`.

**Default:** `1`

**NumSamples**

Number of samples in output

Specify the number of samples in the output of the `step` method as a positive integer. Typically, you use the number of samples in one pulse. This property applies only when you set the `OutputFormat` property to `'Samples'`.

**Default:** 100

**OperatingFrequency**

System operating frequency

Specify the operating frequency of the system in hertz as a positive scalar. The default value corresponds to 300 MHz.

**Default:** 3e8

**TransmitSignalInputPort**

Add input to specify transmit signal

Set this property to `true` to add input to specify the transmit signal in the `step` syntax. Set this property to `false` omit the transmit signal in the `step` syntax. The `false` option is less computationally expensive; to use this option, you must also specify the `TransmitERP` property.

**Default:** false

**WeightsInputPort**

Enable weights input

Set this property to `true` to input weights.

**Default:** false

**TransmitERP**

Effective transmitted power

Specify the transmitted effective radiated power (ERP) of the radar system in watts as a positive scalar. This property applies only when you set the `TransmitSignalInputPort` property to `false`.

**Default:** 5000

**PlatformHeight**

Radar platform height from surface

Specify the radar platform height (in meters) measured upward from the surface as a nonnegative scalar.

**Default:** 300

**PlatformSpeed**

Radar platform speed

Specify the radar platform's speed as a nonnegative scalar in meters per second.

**Default:** 300

**PlatformDirection**

Direction of radar platform motion

Specify the direction of radar platform motion as a 2-by-1 vector in the form [AzimuthAngle; ElevationAngle] in degrees. The default value of this property indicates that the platform moves perpendicular to the radar antenna array's broadside.

Both azimuth and elevation angle are measured in the local coordinate system of the radar antenna or antenna array. Azimuth angle must be between -180 and 180 degrees. Elevation angle must be between -90 and 90 degrees.

**Default:** [90;0]

**MountingAngles**

Sensor mounting angles (deg)

Specify a 3-element vector that gives the intrinsic yaw, pitch, and roll of the sensor frame from the inertial frame. The 3 elements define the rotations around the z, y, and x axes respectively, in that order. The first rotation, rotates the body axes around the z-axis. Because these angles define intrinsic rotations, the second rotation is performed around the y-axis in its new position resulting from the previous rotation. The final rotation around the x-axis is performed around the x-axis as rotated by the first two rotations in the intrinsic system.

**Default:** [0 0 0]

**SeedSource**

Source of seed for random number generator

Specify how the object generates random numbers. Values of this property are:

'Auto'	The default MATLAB random number generator produces the random numbers. Use 'Auto' if you are using this object with Parallel Computing Toolbox software.
'Property'	The object uses its own private random number generator to produce random numbers. The Seed property of this object specifies the seed of the random number generator. Use 'Property' if you want repeatable results and are not using this object with Parallel Computing Toolbox software.

**Default:** 'Auto'

**Seed**

Seed for random number generator

Specify the seed for the random number generator as a scalar integer between 0 and  $2^{32}-1$ . This property applies when you set the SeedSource property to 'Property'.

**Default:** 0

## Methods

reset    Reset random numbers and time count for clutter simulation  
 step    Simulate clutter using constant gamma model

Common to All System Objects	
release	Allow System object property value changes

## Examples

### Simulate Clutter for System with Known Power

Simulate the clutter return from terrain with a gamma value of 0 dB. The effective transmitted power of the radar system is 5 kW.

Set up the characteristics of the radar system. This system uses a four-element uniform linear array (ULA). The sample rate is 1 MHz, and the PRF is 10 kHz. The propagation speed is the speed of light, and the operating frequency is 300 MHz. The radar platform is flying 1 km above the ground with a path parallel to the ground along the array axis. The platform speed is 2 km/s. The mainlobe has a depression angle of 30°.

```
Nele = 4;
c = physconst('Lightspeed');
fc = 300.0e6;
lambda = c/fc;
array = phased.ULA('NumElements',Nele,'ElementSpacing',lambda/2);
fs = 1.0e6;
prf = 10.0e3;
height = 1000.0;
direction = [90;0];
speed = 2.0e3;
depang = 30.0;
mountingAng = [depang,0,0];
```

Create the clutter simulation object. The configuration assumes the earth is flat. The maximum clutter range of interest is 5 km, and the maximum azimuth coverage is  $\pm 60^\circ$ .

```
Rmax = 5000.0;
Azcov = 120.0;
tergamma = 0.0;
tpower = 5000.0;
clutter = constantGammaClutter('Sensor',array,...
    'PropagationSpeed',c,'OperatingFrequency',fc,'PRF',prf,...
    'SampleRate',fs,'Gamma',tergamma,'EarthModel','Flat',...
    'TransmitERP',tpower,'PlatformHeight',height,...
    'PlatformSpeed',speed,'PlatformDirection',direction,...
    'MountingAngles',mountingAng,'ClutterMaxRange',Rmax,...
```

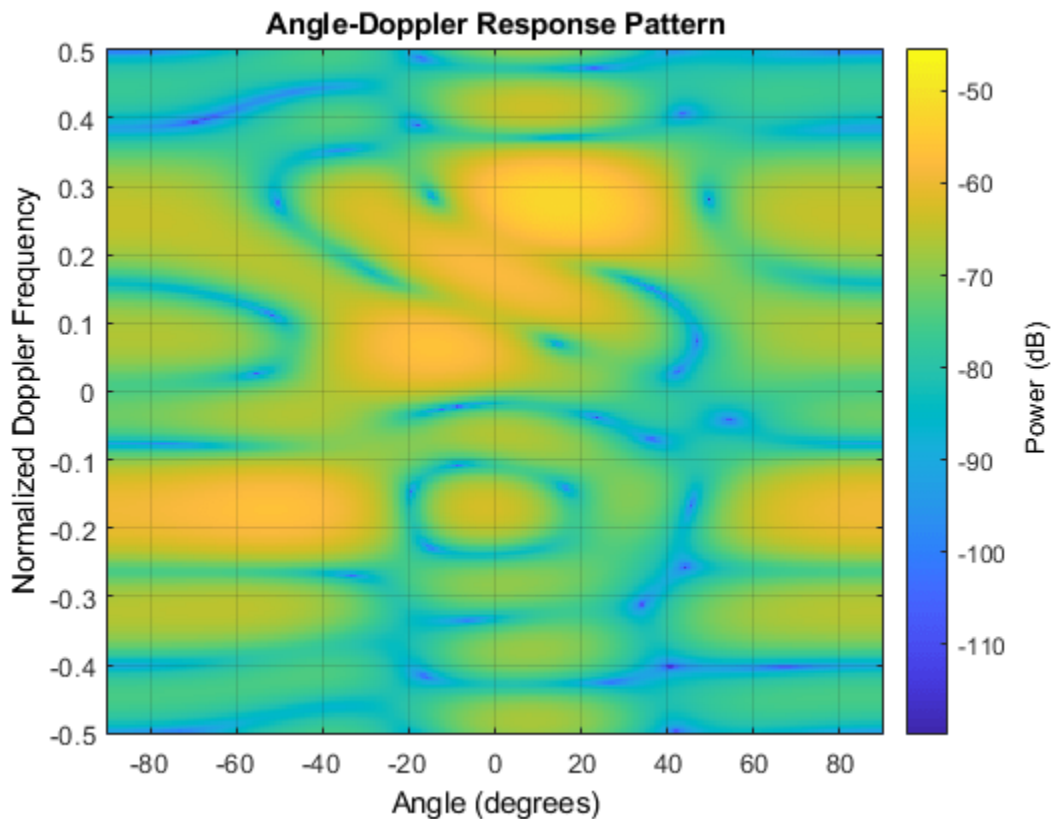
```
'ClutterAzimuthSpan',Azcov,'SeedSource','Property',...
'Seed',40547);
```

Simulate the clutter return for 10 pulses.

```
Nsamp = fs/prf;
Npulse = 10;
sig = zeros(Nsamp,Nele,Npulse);
for m = 1:Npulse
    sig(:,:,m) = clutter();
end
```

Plot the angle-Doppler response of the clutter at the 20th range bin.

```
response = phased.AngleDopplerResponse('SensorArray',array,...
'OperatingFrequency',fc,'PropagationSpeed',c,'PRF',prf);
plotResponse(response,shiftdim(sig(20,:,:)), 'NormalizeDoppler',true)
```



### Simulate Clutter Using Known Transmit Signal

Simulate the clutter return from terrain with a gamma value of 0 dB. You input the transmit signal of the radar system when creating clutter. In this case, you do not use the `TransmitERP` property.

Set up the characteristics of the radar system. This system has a 4-element uniform linear array (ULA). The sample rate is 1 MHz, and the PRF is 10 kHz. The propagation speed is the speed of light,

and the operating frequency is 300 MHz. The radar platform is flying 1 km above the ground with a path parallel to the ground along the array axis. The platform speed is 2 km/s. The mainlobe has a depression angle of 30°.

```
Nele = 4;
c = physconst('Lightspeed');
fc = 300.0e6;
lambda = c/fc;
ula = phased.ULA('NumElements',Nele,'ElementSpacing',lambda/2);
fs = 1.0e6;
prf = 10.0e3;
height = 1.0e3;
direction = [90;0];
speed = 2.0e3;
depan = 30;
mountingAng = [depan,0,0];
```

Create the clutter simulation object and configure it to accept an transmit signal as an input argument. The configuration assumes the earth is flat. The maximum clutter range of interest is 5 km, and the maximum azimuth coverage is  $\pm 60^\circ$ .

```
Rmax = 5000.0;
Azcov = 120.0;
tergamma = 0.0;
clutter = constantGammaClutter('Sensor',ula,...
    'PropagationSpeed',c,'OperatingFrequency',fc,'PRF',prf,...
    'SampleRate',fs,'Gamma',tergamma,'EarthModel','Flat',...
    'TransmitSignalInputPort',true,'PlatformHeight',height,...
    'PlatformSpeed',speed,'PlatformDirection',direction,...
    'MountingAngles',mountingAng,'ClutterMaxRange',Rmax,...
    'ClutterAzimuthSpan',Azcov,'SeedSource','Property',...
    'Seed',40547);
```

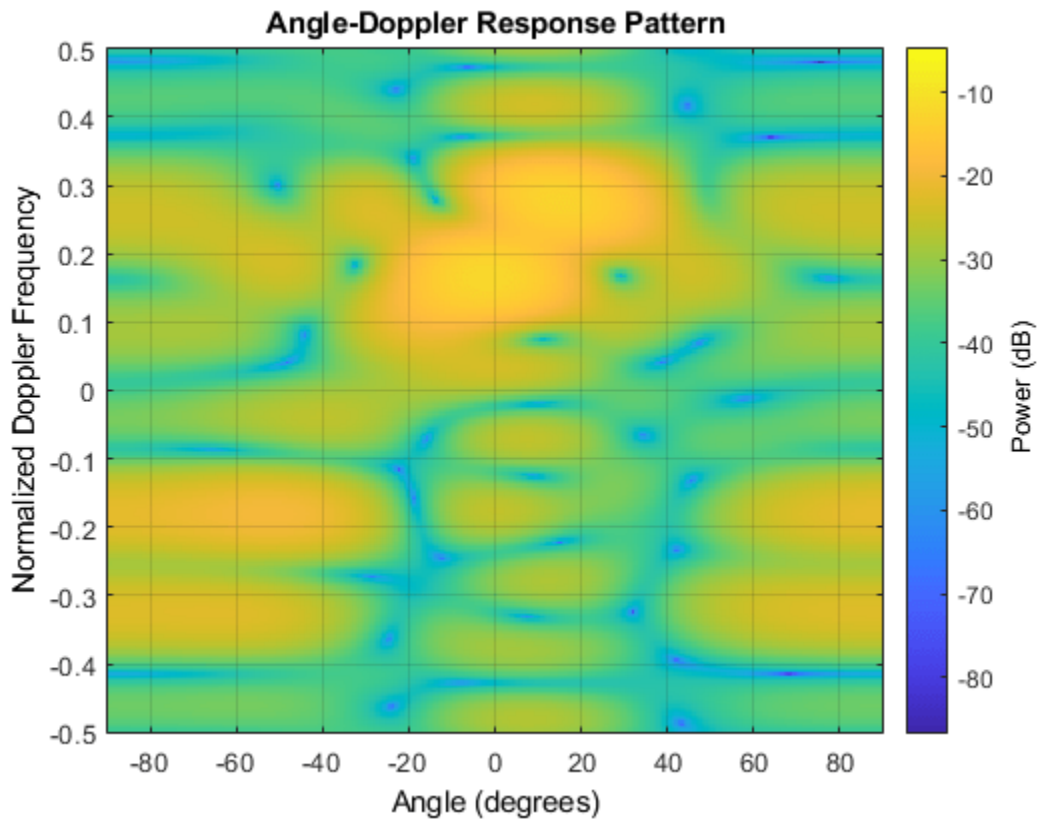
Simulate the clutter return for 10 pulses. At each step, pass the transmit signal as an input argument. The software computes the effective transmitted power of the signal. The transmit signal is a rectangular waveform with a pulse width of 2  $\mu$ s.

```
tpower = 5.0e3;
pw = 2.0e-6;
X = tpower*ones(floor(pw*fs),1);
Nsamp = fs/prf;
Npulse = 10;
sig = zeros(Nsamp,Nele,Npulse);
for m = 1:Npulse
    sig(:,:,m) = step(clutter,X);
end
```

Plot the angle-Doppler response of the clutter at the 20th range bin.

```
response = phased.AngleDopplerResponse('SensorArray',ula,...
    'OperatingFrequency',fc,'PropagationSpeed',c,'PRF',prf);
plotResponse(response,shiftdim(sig(20,:,:)), 'NormalizeDoppler',true)
```





## References

- [1] Barton, David. "Land Clutter Models for Radar Design and Analysis," *Proceedings of the IEEE*. Vol. 73, Number 2, February, 1985, pp. 198-204.
- [2] Long, Maurice W. *Radar Reflectivity of Land and Sea*, 3rd Ed. Boston: Artech House, 2001.
- [3] Nathanson, Fred E., J. Patrick Reilly, and Marvin N. Cohen. *Radar Design Principles*, 2nd Ed. Mendham, NJ: SciTech Publishing, 1999.
- [4] Ward, J. "Space-Time Adaptive Processing for Airborne Radar Data Systems," *Technical Report 1015*, MIT Lincoln Laboratory, December, 1994.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

**See Also**

barrageJammer | gpuConstantGammaClutter | surfacegamma | uv2azel | phitheta2azel

**Topics**

Ground Clutter Mitigation with Moving Target Indication (MTI) Radar  
“DPCA Pulse Canceller to Reject Clutter”  
“Clutter Modeling”

**Introduced in R2021a**

# reset

**System object:** constantGammaClutter

Reset random numbers and time count for clutter simulation

## Syntax

reset(H)

## Description

reset(H) resets the states of the constantGammaClutter object, H. This method resets the random number generator state if the SeedSource property is set to 'Property'. This method resets the elapsed coherence time. Also, if the PRF property is a vector, the next call to step uses the first PRF value in the vector.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## step

**System object:** constantGammaClutter

Simulate clutter using constant gamma model

### Syntax

```
Y = step(H)
Y = step(H,X)
Y = step(H,STEERANGLE)
Y = step(H,X,WS)
Y = step(H,PRFIDX)
Y = step(H,X,STEERANGLE)
```

### Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

`Y = step(H)` computes the collected clutter return at each sensor. This syntax is available when you set the `TransmitSignalInputPort` property to `false`.

`Y = step(H,X)` specifies the transmit signal in `X`. Transmit signal refers to the output of the transmitter while it is on during a given pulse. This syntax is available when you set the `TransmitSignalInputPort` property to `true`.

`Y = step(H,STEERANGLE)` uses `STEERANGLE` as the subarray steering angle. This syntax is available when you configure `H` so that `H.Sensor` is an array that contains subarrays and `H.Sensor.SubarraySteering` is either `'Phase'` or `'Time'`.

`Y = step(H,X,WS)` uses `WS` as weights applied to each element within each subarray. To use this syntax, set the `Sensor` property to an array that supports subarrays and set the `SubarraySteering` property of the array to `'Custom'`.

`Y = step(H,PRFIDX)` uses the index, `PRFIDX`, to select the PRF from a predetermined list of PRFs specified by the `PRF` property. To enable this syntax, set the `PRFSelectionInputPort` to `true`.

`Y = step(H,X,STEERANGLE)` combines all input arguments. This syntax is available when you configure `H` so that `H.TransmitSignalInputPort` is `true`, `H.Sensor` is an array that contains subarrays, and `H.Sensor.SubarraySteering` is either `'Phase'` or `'Time'`.

### Input Arguments

**H**

Constant gamma clutter object.

**X**

Transmit signal, specified as a column vector.

**STEERANGLE**

Subarray steering angle in degrees. **STEERANGLE** can be a length-2 column vector or a scalar.

If **STEERANGLE** is a length-2 vector, it has the form [azimuth; elevation]. The azimuth angle must be between -180 degrees and 180 degrees, and the elevation angle must be between -90 degrees and 90 degrees.

If **STEERANGLE** is a scalar, it represents the azimuth angle. In this case, the elevation angle is assumed to be 0.

**WS**

Subarray element weights

Subarray element weights, specified as complex-valued  $N_{SE}$ -by- $N$  matrix or 1-by- $N$  cell array where  $N$  is the number of subarrays. These weights are applied to the individual elements within a subarray.

**Subarray Element Weights**

Sensor Array	Subarray Weights
phased.ReplicatedSubarray	All subarrays have the same dimensions and sizes. Then, the subarray weights form an $N_{SE}$ -by- $N$ matrix. $N_{SE}$ is the number of elements in each subarray and $N$ is the number of subarrays. Each column of <b>WS</b> specifies the weights for the corresponding subarray.
phased.PartitionedArray	When subarrays do not have the same dimensions and sizes, you can specify subarray weights as <ul style="list-style-type: none"> <li>an <math>N_{SE}</math>-by-<math>N</math> matrix, where <math>N_{SE}</math> is now the number of elements in the largest subarray. The first <math>Q</math> entries in each column are the element weights for the subarray where <math>Q</math> is the number of elements in the subarray.</li> <li>a 1-by-<math>N</math> cell array. Each cell contains a column vector of weights for the corresponding subarray. The column vectors have lengths equal to the number of elements in the corresponding subarray.</li> </ul>

**Dependencies**

To enable this argument, set the **Sensor** property to an array that contains subarrays and set the **SubarraySteering** property of the array to 'Custom'.

**PRFIDX**

Index of pulse repetition frequency, specified as a positive integer. The index selects one of the entries specified in the **PRF** property as the PRF for the next transmission.

Example: 3

### Dependencies

To enable this argument, set the `PRFSelectionInputPort` to `true`.

## Output Arguments

### Y

Collected clutter return at each sensor. `Y` has dimensions  $N$ -by- $M$  matrix. If `H.Sensor` contains subarrays,  $M$  is the number of subarrays in the radar system. Otherwise it is the number of sensors. When you set the `OutputFormat` property to `'Samples'`,  $N$  is defined by the `NumSamples` property. When you set the `OutputFormat` property to `'Pulses'`,  $N$  is the total number of samples in the next  $L$  pulses. In this case,  $L$  is defined by the `NumPulses` property.

## Examples

### Simulate Clutter for System with Known Power

Simulate the clutter return from terrain with a gamma value of 0 dB. The effective transmitted power of the radar system is 5 kW.

Set up the characteristics of the radar system. This system uses a four-element uniform linear array (ULA). The sample rate is 1 MHz, and the PRF is 10 kHz. The propagation speed is the speed of light, and the operating frequency is 300 MHz. The radar platform is flying 1 km above the ground with a path parallel to the ground along the array axis. The platform speed is 2 km/s. The mainlobe has a depression angle of 30°.

```
Nele = 4;
c = physconst('Lightspeed');
fc = 300.0e6;
lambda = c/fc;
array = phased.ULA('NumElements',Nele,'ElementSpacing',lambda/2);
fs = 1.0e6;
prf = 10.0e3;
height = 1000.0;
direction = [90;0];
speed = 2.0e3;
depan = 30.0;
mountingAng = [depan,0,0];
```

Create the clutter simulation object. The configuration assumes the earth is flat. The maximum clutter range of interest is 5 km, and the maximum azimuth coverage is  $\pm 60^\circ$ .

```
Rmax = 5000.0;
Azcov = 120.0;
tergamma = 0.0;
tpower = 5000.0;
clutter = constantGammaClutter('Sensor',array,...
    'PropagationSpeed',c,'OperatingFrequency',fc,'PRF',prf,...
    'SampleRate',fs,'Gamma',tergamma,'EarthModel','Flat',...
    'TransmitERP',tpower,'PlatformHeight',height,...
    'PlatformSpeed',speed,'PlatformDirection',direction,...
```

```

'MountingAngles',mountingAng,'ClutterMaxRange',Rmax,...
'ClutterAzimuthSpan',Azcov,'SeedSource','Property',...
'Seed',40547);

```

Simulate the clutter return for 10 pulses.

```

Nsamp = fs/prf;
Npulse = 10;
sig = zeros(Nsamp,Nele,Npulse);
for m = 1:Npulse
    sig(:,:,m) = clutter();
end

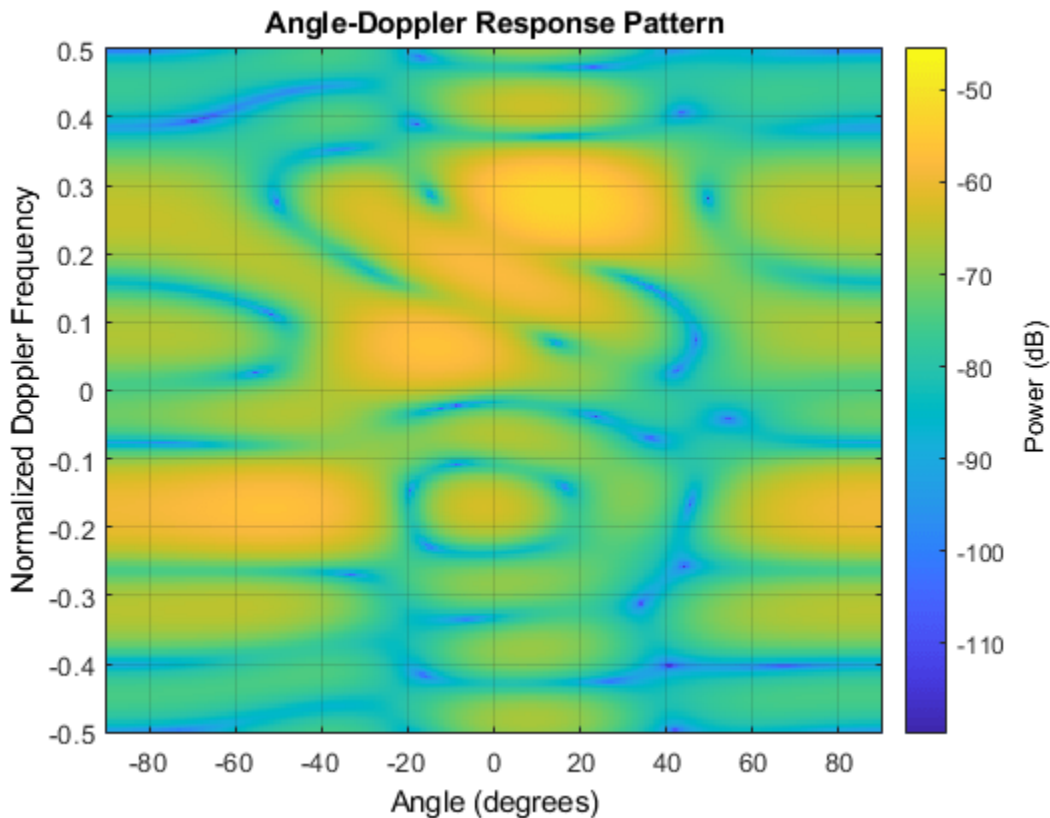
```

Plot the angle-Doppler response of the clutter at the 20th range bin.

```

response = phased.AngleDopplerResponse('SensorArray',array,...
'OperatingFrequency',fc,'PropagationSpeed',c,'PRF',prf);
plotResponse(response,shiftdim(sig(20,:,:)),'NormalizeDoppler',true)

```



### Simulate Clutter Using Known Transmit Signal

Simulate the clutter return from terrain with a gamma value of 0 dB. You input the transmit signal of the radar system when creating clutter. In this case, you do not use the `TransmitERP` property.

Set up the characteristics of the radar system. This system has a 4-element uniform linear array (ULA). The sample rate is 1 MHz, and the PRF is 10 kHz. The propagation speed is the speed of light,

and the operating frequency is 300 MHz. The radar platform is flying 1 km above the ground with a path parallel to the ground along the array axis. The platform speed is 2 km/s. The mainlobe has a depression angle of 30°.

```
Nele = 4;
c = physconst('Lightspeed');
fc = 300.0e6;
lambda = c/fc;
ula = phased.ULA('NumElements',Nele,'ElementSpacing',lambda/2);
fs = 1.0e6;
prf = 10.0e3;
height = 1.0e3;
direction = [90;0];
speed = 2.0e3;
depan = 30;
mountingAng = [depan,0,0];
```

Create the clutter simulation object and configure it to accept an transmit signal as an input argument. The configuration assumes the earth is flat. The maximum clutter range of interest is 5 km, and the maximum azimuth coverage is  $\pm 60^\circ$ .

```
Rmax = 5000.0;
Azcov = 120.0;
tergamma = 0.0;
clutter = constantGammaClutter('Sensor',ula,...
    'PropagationSpeed',c,'OperatingFrequency',fc,'PRF',prf,...
    'SampleRate',fs,'Gamma',tergamma,'EarthModel','Flat',...
    'TransmitSignalInputPort',true,'PlatformHeight',height,...
    'PlatformSpeed',speed,'PlatformDirection',direction,...
    'MountingAngles',mountingAng,'ClutterMaxRange',Rmax,...
    'ClutterAzimuthSpan',Azcov,'SeedSource','Property',...
    'Seed',40547);
```

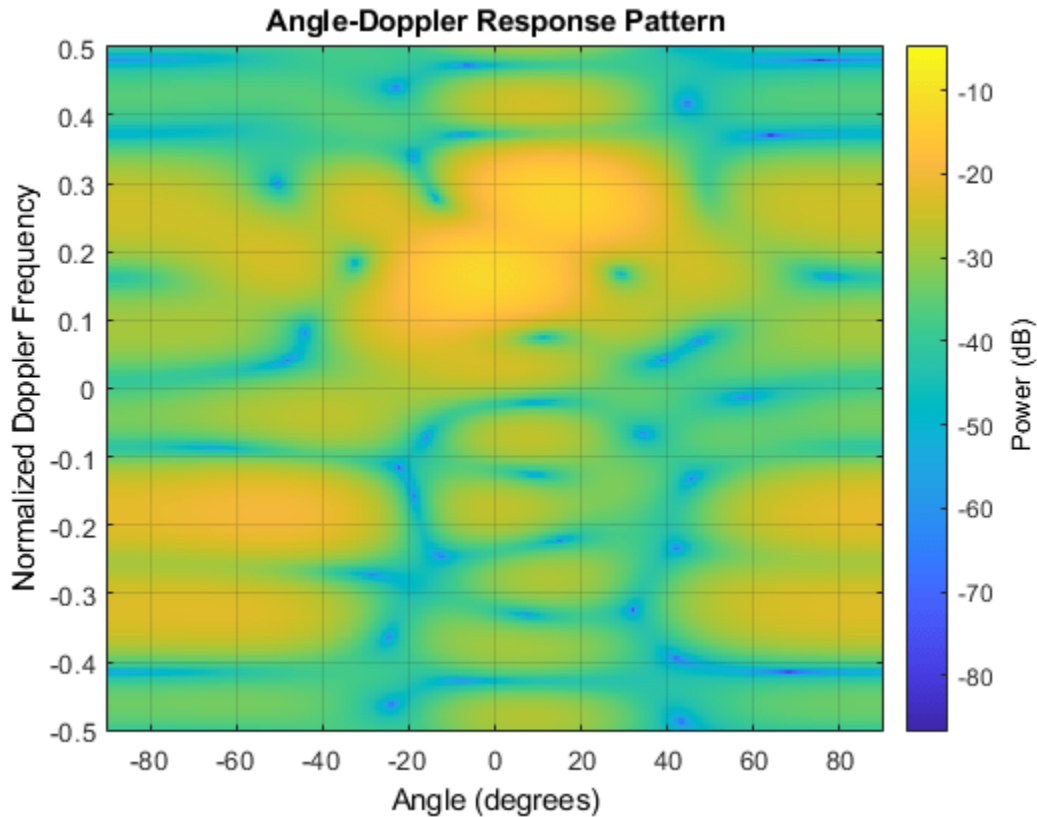
Simulate the clutter return for 10 pulses. At each step, pass the transmit signal as an input argument. The software computes the effective transmitted power of the signal. The transmit signal is a rectangular waveform with a pulse width of 2  $\mu$ s.

```
tpower = 5.0e3;
pw = 2.0e-6;
X = tpower*ones(floor(pw*fs),1);
Nsamp = fs/prf;
Npulse = 10;
sig = zeros(Nsamp,Nele,Npulse);
for m = 1:Npulse
    sig(:, :, m) = step(clutter, X);
end
```

Plot the angle-Doppler response of the clutter at the 20th range bin.

```
response = phased.AngleDopplerResponse('SensorArray',ula,...
    'OperatingFrequency',fc,'PropagationSpeed',c,'PRF',prf);
plotResponse(response,shiftdim(sig(20,:,:)), 'NormalizeDoppler',true)
```





## Tips

The clutter simulation that `constantGammaClutter` provides is based on these assumptions:

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

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

### **Topics**

Ground Clutter Mitigation with Moving Target Indication (MTI) Radar  
“DPCA Pulse Canceller to Reject Clutter”  
“Clutter Modeling”

# gpuConstantGammaClutter

Simulate constant-gamma clutter using GPU

## Description

The `gpuConstantGammaClutter` object simulates clutter, performing the computations on a GPU.

---

**Note** To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see “GPU Computing” (Parallel Computing Toolbox).

---

To compute the clutter return:

- 1 Define and set up your clutter simulator. See “Construction” on page 4-499.
- 2 Call `step` to simulate the clutter return for your system according to the properties of `gpuConstantGammaClutter`. The behavior of `step` is specific to each object in the toolbox.

The clutter simulation that `constantGammaClutter` provides is based on these assumptions:

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

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

---

## Construction

`H = gpuConstantGammaClutter` creates a constant-gamma clutter simulation System object, `H`. This object simulates the clutter return of a monostatic radar system using the constant gamma model.

`H = gpuConstantGammaClutter(Name, Value)` creates a constant gamma clutter simulation object, `H`, with additional options specified by one or more `Name, Value` pair arguments. `Name` is a property name on page 4-500, and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Properties

### Sensor

Handle of sensor

Specify the sensor as an antenna element object or as an array object whose `Element` property value is an antenna element object. If the sensor is an array, it can contain subarrays.

**Default:** `phased.ULA` with default property values

### Gamma

Terrain gamma value

Specify the  $\gamma$  value used in the constant  $\gamma$  clutter model, as a scalar in decibels. The  $\gamma$  value depends on both terrain type and the operating frequency.

**Default:** 0

### EarthModel

Earth model

Specify the earth model used in clutter simulation as one of | 'Flat' | 'Curved' |. When you set this property to 'Flat', the earth is assumed to be a flat plane. When you set this property to 'Curved', the earth is assumed to be a sphere.

**Default:** 'Flat'

### ClutterMinRange

Minimum range of clutter region (m)

Minimum range at which to computer clutter returns, specified as a positive scalar. The minimum range must be nonnegative. This value is ignored if it less than the value of `PlatformHeight`. Units are in meters.

**Default:** 0

### ClutterMaxRange

Maximum range of clutter region (m)

Specify the maximum range at which to compute clutter returns. for the clutter simulation as a positive scalar. The maximum range must be greater than the value specified in the `PlatformHeight` property. Units are in meters.

**Default:** 5000

### ClutterAzimuthCenter

Azimuth center of clutter region (deg)

The azimuth angle in the ground plane about which clutter patches are generated. Patches are generated symmetrically about this angle.

**Default:** 0

### **ClutterAzimuthSpan**

Azimuth span of clutter patches (deg)

Specify the coverage in azimuth (in degrees) of the clutter region as a positive scalar. The clutter simulation covers a region having the specified azimuth span, symmetric around `ClutterAzimuthCenter`. Typically, all clutter patches have their azimuth centers within the region, but the `PatchAzimuthSpan` value can cause some patches to extend beyond the region.

**Default:** 60

### **PatchAzimuthSpan**

Azimuth span of clutter patches (deg)

Specify the azimuth span (in degrees) of each clutter patch as a positive scalar.

**Default:** 1

### **CoherenceTime**

Clutter coherence time

Specify the coherence time in seconds for the clutter simulation as a positive scalar. After the coherence time elapses, the `step` method updates the random numbers it uses for the clutter simulation at the next pulse. A value of `inf` means the random numbers are never updated.

**Default:** `inf`

### **PropagationSpeed**

Signal propagation speed

Specify the propagation speed of the signal, in meters per second, as a positive scalar.

**Default:** Speed of light

### **SampleRate**

Sample rate

Specify the sample rate, in hertz, as a positive scalar. The default value corresponds to 1 MHz.

**Default:** `1e6`

### **PRF**

Pulse repetition frequency

Pulse repetition frequency, *PRF*, specified as a scalar or a row vector. Units are in Hz. The pulse repetition interval, *PRI*, is the inverse of the pulse repetition frequency, *PRF*. The *PRF* must satisfy these restrictions:

- The product of *PRF* and *PulseWidth* must be less than or equal to one. This condition expresses the requirement that the pulse width is less than one pulse repetition interval. For the phase-coded waveform, the pulse width is the product of the chip width and number of chips.
- The ratio of sample rate to any element of *PRF* must be an integer. This condition expresses the requirement that the number of samples in one pulse repetition interval is an integer.

You can select the value of *PRF* using property settings alone or using property settings in conjunction with the *prfidx* input argument of the *step* method.

- When *PRFSelectionInputPort* is *false*, you set the *PRF* using properties only. You can
  - implement a constant *PRF* by specifying *PRF* as a positive real-valued scalar.
  - implement a staggered *PRF* by specifying *PRF* as a row vector with positive real-valued entries. Then, each call to the *step* method uses successive elements of this vector for the *PRF*. If the last element of the vector is reached, the process continues cyclically with the first element of the vector.
- When *PRFSelectionInputPort* is *true*, you can implement a selectable *PRF* by specifying *PRF* as a row vector with positive real-valued entries. But this time, when you execute the *step* method, select a *PRF* by passing an argument specifying an index into the *PRF* vector.

In all cases, the number of output samples is fixed when you set the *OutputFormat* property to 'Samples'. When you use a varying *PRF* and set the *OutputFormat* property to 'Pulses', the number of samples can vary.

**Default:** 10e3

### **PRFSelectionInputPort**

Enable PRF selection input

Enable the PRF selection input, specified as *true* or *false*. When you set this property to *false*, the *step* method uses the values set in the *PRF* property. When you set this property to *true*, you pass an index argument into the *step* method to select a value from the *PRF* vector.

**Default:** *false*

### **OutputFormat**

Output signal format

Specify the format of the output signal as one of | 'Pulses' | 'Samples' |. When you set the *OutputFormat* property to 'Pulses', the output of the *step* method is in the form of multiple pulses. In this case, the number of pulses is the value of the *NumPulses* property.

When you set the *OutputFormat* property to 'Samples', the output of the *step* method is in the form of multiple samples. In this case, the number of samples is the value of the *NumSamples* property. In staggered PRF applications, you might find the 'Samples' option more convenient because the *step* output always has the same matrix size.

**Default:** 'Pulses'

### **NumPulses**

Number of pulses in output

Specify the number of pulses in the output of the `step` method as a positive integer. This property applies only when you set the `OutputFormat` property to 'Pulses'.

**Default:** 1

### **NumSamples**

Number of samples in output

Specify the number of samples in the output of the `step` method as a positive integer. Typically, you use the number of samples in one pulse. This property applies only when you set the `OutputFormat` property to 'Samples'.

**Default:** 100

### **OperatingFrequency**

System operating frequency

Specify the operating frequency of the system in hertz as a positive scalar. The default value corresponds to 300 MHz.

**Default:** 3e8

### **TransmitSignalInputPort**

Add input to specify transmit signal

Set this property to `true` to add input to specify the transmit signal in the `step` syntax. Set this property to `false` omit the transmit signal in the `step` syntax. The `false` option is less computationally expensive; to use this option, you must also specify the `TransmitERP` property.

**Default:** false

### **WeightsInputPort**

Enable weights input

Set this property to `true` to input weights.

**Default:** false

### **TransmitERP**

Effective transmitted power

Specify the transmitted effective radiated power (ERP) of the radar system in watts as a positive scalar. This property applies only when you set the `TransmitSignalInputPort` property to `false`.

**Default:** 5000

### **PlatformHeight**

Radar platform height from surface

Specify the radar platform height (in meters) measured upward from the surface as a nonnegative scalar.

**Default:** 300

### **PlatformSpeed**

Radar platform speed

Specify the radar platform's speed as a nonnegative scalar in meters per second.

**Default:** 300

### **PlatformDirection**

Direction of radar platform motion

Specify the direction of radar platform motion as a 2-by-1 vector in the form [AzimuthAngle; ElevationAngle] in degrees. The default value of this property indicates that the platform moves perpendicular to the radar antenna array's broadside.

Both azimuth and elevation angle are measured in the local coordinate system of the radar antenna or antenna array. Azimuth angle must be between -180 and 180 degrees. Elevation angle must be between -90 and 90 degrees.

**Default:** [90;0]

### **MountingAngles**

Sensor mounting angles (deg)

Specify a 3-element vector that gives the intrinsic yaw, pitch, and roll of the sensor frame from the inertial frame. The 3 elements define the rotations around the z, y, and x axes respectively, in that order. The first rotation, rotates the body axes around the z-axis. Because these angles define intrinsic rotations, the second rotation is performed around the y-axis in its new position resulting from the previous rotation. The final rotation around the x-axis is performed around the x-axis as rotated by the first two rotations in the intrinsic system.

**Default:** [0 0 0]

### **SeedSource**

Source of seed for random number generator

Specify how the object generates random numbers. Values of this property are:

'Auto'	The default MATLAB random number generator produces the random numbers. Use 'Auto' if you are using this object with Parallel Computing Toolbox software.
'Property'	The object uses its own private random number generator to produce random numbers. The Seed property of this object specifies the seed of the random number generator. Use 'Property' if you want repeatable results and are not using this object with Parallel Computing Toolbox software.

**Default:** 'Auto'



## Seed

Seed for random number generator

Specify the seed for the random number generator as a scalar integer between 0 and  $2^{32}-1$ . This property applies when you set the `SeedSource` property to 'Property'.

**Default:** 0

## Methods

`reset`    Reset random numbers and time count for clutter simulation

`step`     Simulate clutter using constant gamma model

### Common to All System Objects

<code>release</code>	Allow System object property value changes
----------------------	--

## Examples

### GPU Clutter Simulation of Radar System with Known Power

Simulate the clutter return from terrain with a gamma value of 0 dB. The effective transmitted power of the radar system is 5 kW.

Set up the characteristics of the radar system. This system uses a 4-element uniform linear array (ULA). The sample rate is 1 MHz, and the PRF is 10 kHz. The propagation speed is the speed of light, and the operating frequency is 300 MHz. The radar platform is flying 1 km above the ground with a path parallel to the ground along the array axis. The platform speed is 2000 m/s. The mainlobe has a depression angle of 30°.

```

Nele = 4;
c = physconst('Lightspeed');
fc = 300e6;
lambda = c/fc;
array = phased.ULA('NumElements',Nele,'ElementSpacing',lambda/2);
fs = 1e6;
prf = 10e3;
height = 1000.0;
direction = [90;0];
speed = 2.0e3;
depang = 30.0;
mountingAng = [0,30,0];

```

Create the GPU clutter simulation object. The configuration assumes the earth is flat. The maximum clutter range of interest is 5 km, and the maximum azimuth coverage is  $\pm 60^\circ$ .

```

Rmax = 5000;
Azcov = 120;
tergamma = 0;
tpower = 5000;
clutter = gpuConstantGammaClutter('Sensor',array, ...
    'PropagationSpeed',c,'OperatingFrequency',fc,'PRF',prf, ...

```

```

'SampleRate',fs,'Gamma',tergamma,'EarthModel','Flat',...
'TransmitERP',tpower,'PlatformHeight',height,...
'PlatformSpeed',speed,'PlatformDirection',direction,...
'MountingAngles',mountingAng,'ClutterMaxRange',Rmax,...
'ClutterAzimuthSpan',Azcov,'SeedSource','Property',...
'Seed',40547);

```

Simulate the clutter return for 10 pulses.

```

Nsamp = fs/prf;
Npulse = 10;
clsig = zeros(Nsamp,Nele,Npulse);
for m = 1:Npulse
    clsig(:, :, m) = clutter();
end

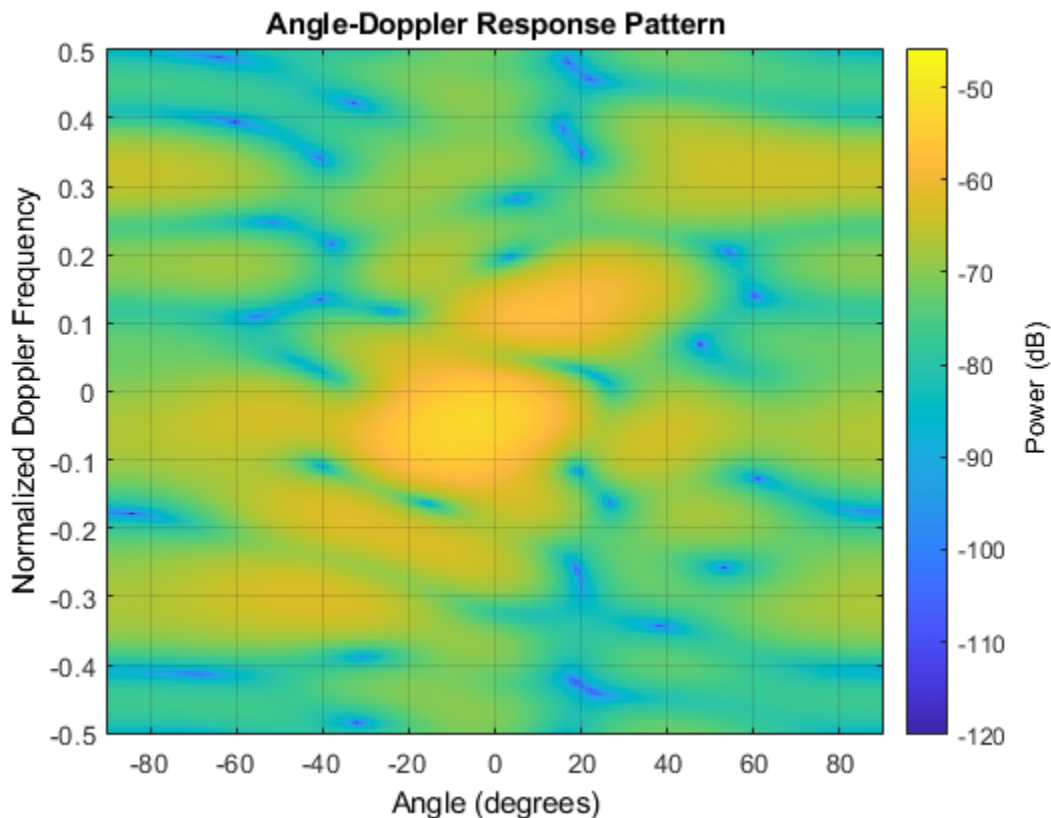
```

Plot the angle-Doppler response of the clutter at the 20th range bin.

```

response = phased.AngleDopplerResponse('SensorArray',array,...
'OperatingFrequency',fc,'PropagationSpeed',c,'PRF',prf);
plotResponse(response,shiftdim(clsig(20,:,:)), 'NormalizeDoppler',true);

```



The results are not identical to the results obtained by using `constantGammaClutter` because of differences between CPU and GPU computations.

## GPU Clutter Simulation With Known Transmit Signal

Simulate the clutter return from terrain with a gamma value of 0 dB. You input the transmit signal of the radar system when creating clutter. In this case, you do not specify the effective transmitted power of the signal in a property.

Set up the characteristics of the radar system. This system has a 4-element uniform linear array (ULA). The sample rate is 1 MHz, and the PRF is 10 kHz. The propagation speed is the speed of light, and the operating frequency is 300 MHz. The radar platform is flying 1 km above the ground with a path parallel to the ground along the array axis. The platform speed is 2000 m/s. The mainlobe has a depression angle of 30°.

```
Nele = 4;
c = physconst('LightSpeed');
fc = 300e6;
lambda = c/fc;
ha = phased.ULA('NumElements',Nele,'ElementSpacing',lambda/2);
fs = 1e6;
prf = 10e3;
height = 1000;
direction = [90;0];
speed = 2000;
mountingAng = [0,30,0];
```

Create the GPU clutter simulation object and configure it to take a transmitted signal as an input argument. The configuration assumes the earth is flat. The maximum clutter range of interest is 5 km, and the maximum azimuth coverage is  $\pm 60^\circ$ .

```
Rmax = 5000;
Azcov = 120;
tergamma = 0;
clutter = gpuConstantGammaClutter('Sensor',ha,...
    'PropagationSpeed',c,'OperatingFrequency',fc,'PRF',prf,...
    'SampleRate',fs,'Gamma',tergamma,'EarthModel','Flat',...
    'TransmitSignalInputPort',true,'PlatformHeight',height,...
    'PlatformSpeed',speed,'PlatformDirection',direction,...
    'MountingAngles',mountingAng,'ClutterMaxRange',Rmax,...
    'ClutterAzimuthSpan',Azcov,'SeedSource','Property','Seed',40547);
```

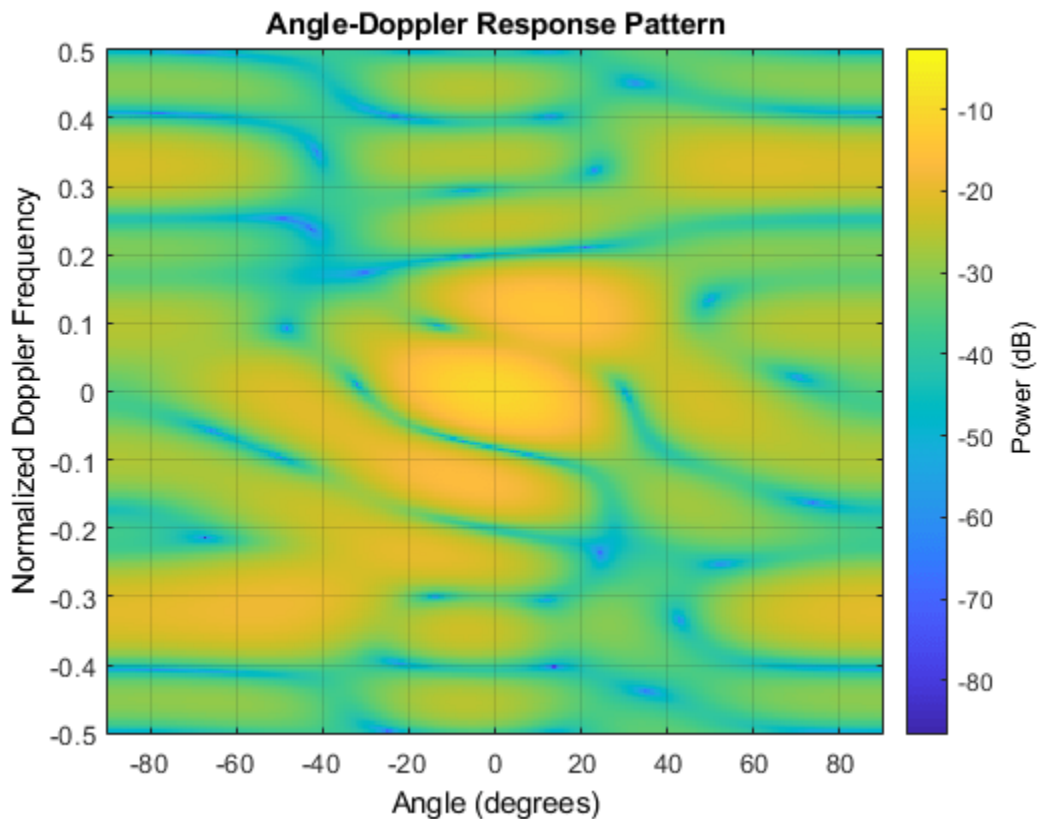
Simulate the clutter return for 10 pulses. At each object call, pass the transmit signal as an input argument. The software automatically computes the effective transmitted power of the signal. The transmit signal is a rectangular waveform with a pulse width of 2  $\mu$ s.

```
tpower = 5000;
pw = 2e-6;
X = tpower*ones(floor(pw*fs),1);
Nsamp = fs/prf;
Npulse = 10;
clsig = zeros(Nsamp,Nele,Npulse);
for m = 1:Npulse
    clsig(:, :, m) = clutter(X);
end
```

Plot the angle-Doppler response of the clutter at the 20th range bin.

```
response = phased.AngleDopplerResponse('SensorArray',ha,...
    'OperatingFrequency',fc,'PropagationSpeed',c,'PRF',prf);
```

```
plotResponse(response,shiftdim(clsig(20,:,:)),...
    'NormalizeDoppler',true);
```



The results are not identical to the results obtained by using `constantGammaClutter` because of differences between CPU and GPU computations.

### Random Number Comparison Between GPU and CPU

In most cases, it does not matter that the GPU and CPU use different random numbers. Sometimes, you may need to reproduce the same stream on both GPU and CPU. In such cases, you can set up the two global streams so they produce identical random numbers. Both GPU and CPU support the combined multiple recursive generator (mrg32k3a) with the `NormalTransform` parameter set to `'Inversion'`.

Define a seed value to use for both the GPU stream and the CPU stream.

```
seed = 7151;
```

Create a CPU random number stream that uses the combined multiple recursive generator and the chosen seed value. Then, use this stream as the global stream for random number generation on the CPU.

```
stream_cpu = RandStream('CombRecursive','Seed',seed, ...
    'NormalTransform','Inversion');
RandStream.setGlobalStream(stream_cpu);
```

Create a GPU random number stream that uses the combined multiple recursive generator and the same seed value. Then, use this stream as the global stream for random number generation on the GPU.

```
stream_gpu = parallel.gpu.RandStream('CombRecursive','Seed',seed, ...
    'NormalTransform','Inversion');
parallel.gpu.RandStream.setGlobalStream(stream_gpu);
```

Generate clutter on both the CPU and the GPU, using the global stream on each platform.

```
clutter_cpu = constantGammaClutter('SeedSource','Auto');
clutter_gpu = gpuConstantGammaClutter('SeedSource','Auto');
cl_cpu = clutter_cpu();
cl_gpu = clutter_gpu();
```

Check that the element-wise differences between the CPU and GPU results are negligible.

```
maxdiff = max(max(abs(cl_cpu - cl_gpu)))
```

```
maxdiff = 3.4027e-18
```

```
eps
```

```
ans = 2.2204e-16
```

## References

- [1] Barton, David. "Land Clutter Models for Radar Design and Analysis," *Proceedings of the IEEE*. Vol. 73, Number 2, February, 1985, pp. 198-204.
- [2] Long, Maurice W. *Radar Reflectivity of Land and Sea*, 3rd Ed. Boston: Artech House, 2001.
- [3] Nathanson, Fred E., J. Patrick Reilly, and Marvin N. Cohen. *Radar Design Principles*, 2nd Ed. Mendham, NJ: SciTech Publishing, 1999.
- [4] Ward, J. "Space-Time Adaptive Processing for Airborne Radar Data Systems," *Technical Report 1015*, MIT Lincoln Laboratory, December, 1994.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

## See Also

barrageJammer | surfacegamma | uv2azel | phitheta2azel

## Topics

Acceleration of Clutter Simulation Using GPU and Code Generation  
 Ground Clutter Mitigation with Moving Target Indication (MTI) Radar  
 "Clutter Modeling"

“GPU Computing” (Parallel Computing Toolbox)

**Introduced in R2021a**

# reset

**System object:** `gpuConstantGammaClutter`

Reset random numbers and time count for clutter simulation

## Syntax

`reset(H)`

## Description

`reset(H)` resets the states of the `gpuConstantGammaClutter` object, `H`. This method resets the random number generator state if the `SeedSource` property is set to `'Property'`. This method resets the elapsed coherence time. Also, if the `PRF` property is a vector, the next call to `step` uses the first `PRF` value in the vector.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## step

**System object:** `gpuConstantGammaClutter`

Simulate clutter using constant gamma model

### Syntax

```
Y = step(H)
Y = step(H,X)
Y = step(H,STEERANGLE)
Y = step(H,WS)
Y = step(H,PRFIDX)
Y = step(H,X,STEERANGLE)
```

### Description

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

`Y = step(H)` computes the collected clutter return at each sensor. This syntax is available when you set the `TransmitSignalInputPort` property to `false`.

`Y = step(H,X)` specifies the transmit signal in `X`. Transmit signal refers to the output of the transmitter while it is on during a given pulse. This syntax is available when you set the `TransmitSignalInputPort` property to `true`.

`Y = step(H,STEERANGLE)` uses `STEERANGLE` as the subarray steering angle. This syntax is available when you configure `H` so that `H.Sensor` is an array that contains subarrays and `H.Sensor.SubarraySteering` is either `'Phase'` or `'Time'`.

`Y = step(H,WS)` uses `WS` as weights applied to each element within each subarray. To use this syntax, set the `Sensor` property to an array that supports subarrays and set the `SubarraySteering` property of the array to `'Custom'`.

`Y = step(H,PRFIDX)` uses the index, `PRFIDX`, to select the PRF from a predetermined list of PRFs specified by the `PRF` property. To enable this syntax, set the `PRFSelectionInputPort` to `true`.

`Y = step(H,X,STEERANGLE)` combines all input arguments. This syntax is available when you configure `H` so that `H.TransmitSignalInputPort` is `true`, `H.Sensor` is an array that contains subarrays, and `H.Sensor.SubarraySteering` is either `'Phase'` or `'Time'`.

### Input Arguments

**H**

Constant gamma clutter object.



**X**

Transmit signal, specified as a column vector of data type `double`. The System object handles data transfer between the CPU and GPU.

**STEERANGLE**

Subarray steering angle in degrees. `STEERANGLE` can be a length-2 column vector or a scalar.

If `STEERANGLE` is a length-2 vector, it has the form [azimuth; elevation]. The azimuth angle must be between -180 degrees and 180 degrees, and the elevation angle must be between -90 degrees and 90 degrees.

If `STEERANGLE` is a scalar, it represents the azimuth angle. In this case, the elevation angle is assumed to be 0.

**WS**

Subarray element weights

Subarray element weights, specified as complex-valued  $N_{SE}$ -by- $N$  matrix or 1-by- $N$  cell array where  $N$  is the number of subarrays. These weights are applied to the individual elements within a subarray.

**Subarray Element Weights**

Sensor Array	Subarray weights
<code>phased.ReplicatedSubarray</code>	All subarrays have the same dimensions and sizes. Then, the subarray weights form an $N_{SE}$ -by- $N$ matrix. $N_{SE}$ is the number of elements in each subarray and $N$ is the number of subarrays. Each column of <code>WS</code> specifies the weights for the corresponding subarray.
<code>phased.PartitionedArray</code>	When subarrays do not have the same dimensions and sizes, you can specify subarray weights as <ul style="list-style-type: none"> <li>an <math>N_{SE}</math>-by-<math>N</math> matrix, where <math>N_{SE}</math> is now the number of elements in the largest subarray. The first <math>Q</math> entries in each column are the element weights for the subarray where <math>Q</math> is the number of elements in the subarray.</li> <li>a 1-by-<math>N</math> cell array. Each cell contains a column vector of weights for the corresponding subarray. The column vectors have lengths equal to the number of elements in the corresponding subarray.</li> </ul>

**Dependencies**

To enable this argument, set the `Sensor` property to an array that contains subarrays and set the `SubarraySteering` property of the array to `'Custom'`.

## PRFIDX

Index of pulse repetition frequency, specified as a positive integer. The index selects one of the entries specified in the PRF property as the PRF for the next transmission.

Example: 4

## Dependencies

To enable this argument, set the PRFSelectionInputPort to true.

## Output Arguments

### Y

Collected clutter return at each sensor. Y has dimensions  $N$ -by- $M$  matrix. If H.Sensor contains subarrays,  $M$  is the number of subarrays in the radar system. Otherwise it is the number of sensors. When you set the OutputFormat property to 'Samples',  $N$  is defined by the NumSamples property. When you set the OutputFormat property to 'Pulses',  $N$  is the total number of samples in the next  $L$  pulses. In this case,  $L$  is defined by the NumPulses property.

## Examples

### GPU Clutter Simulation of Radar System with Known Power

Simulate the clutter return from terrain with a gamma value of 0 dB. The effective transmitted power of the radar system is 5 kW.

Set up the characteristics of the radar system. This system uses a 4-element uniform linear array (ULA). The sample rate is 1 MHz, and the PRF is 10 kHz. The propagation speed is the speed of light, and the operating frequency is 300 MHz. The radar platform is flying 1 km above the ground with a path parallel to the ground along the array axis. The platform speed is 2000 m/s. The mainlobe has a depression angle of 30°.

```
Nele = 4;
c = physconst('Lightspeed');
fc = 300e6;
lambda = c/fc;
array = phased.ULA('NumElements',Nele,'ElementSpacing',lambda/2);
fs = 1e6;
prf = 10e3;
height = 1000.0;
direction = [90;0];
speed = 2.0e3;
depang = 30.0;
mountingAng = [0,30,0];
```

Create the GPU clutter simulation object. The configuration assumes the earth is flat. The maximum clutter range of interest is 5 km, and the maximum azimuth coverage is  $\pm 60^\circ$ .

```
Rmax = 5000;
Azcov = 120;
tergamma = 0;
tpower = 5000;
```

```

clutter = gpuConstantGammaClutter('Sensor',array, ...
    'PropagationSpeed',c,'OperatingFrequency',fc,'PRF',prf, ...
    'SampleRate',fs,'Gamma',tergamma,'EarthModel','Flat', ...
    'TransmitERP',tpower,'PlatformHeight',height, ...
    'PlatformSpeed',speed,'PlatformDirection',direction, ...
    'MountingAngles',mountingAng,'ClutterMaxRange',Rmax, ...
    'ClutterAzimuthSpan',Azcov,'SeedSource','Property', ...
    'Seed',40547);

```

Simulate the clutter return for 10 pulses.

```

Nsamp = fs/prf;
Npulse = 10;
clsig = zeros(Nsamp,Nele,Npulse);
for m = 1:Npulse
    clsig(:,:,m) = clutter();
end

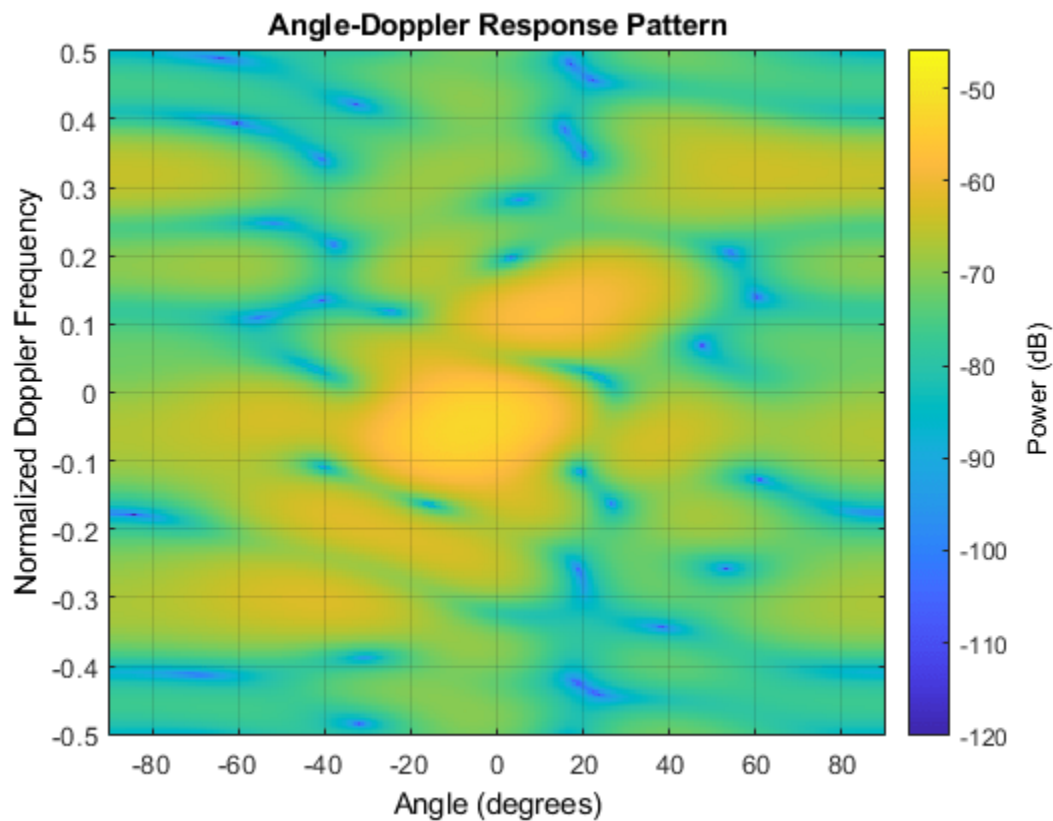
```

Plot the angle-Doppler response of the clutter at the 20th range bin.

```

response = phased.AngleDopplerResponse('SensorArray',array, ...
    'OperatingFrequency',fc,'PropagationSpeed',c,'PRF',prf);
plotResponse(response,shiftdim(clsig(20,:,:)),'NormalizeDoppler',true);

```



The results are not identical to the results obtained by using `constantGammaClutter` because of differences between CPU and GPU computations.

### GPU Clutter Simulation With Known Transmit Signal

Simulate the clutter return from terrain with a gamma value of 0 dB. You input the transmit signal of the radar system when creating clutter. In this case, you do not specify the effective transmitted power of the signal in a property.

Set up the characteristics of the radar system. This system has a 4-element uniform linear array (ULA). The sample rate is 1 MHz, and the PRF is 10 kHz. The propagation speed is the speed of light, and the operating frequency is 300 MHz. The radar platform is flying 1 km above the ground with a path parallel to the ground along the array axis. The platform speed is 2000 m/s. The mainlobe has a depression angle of 30°.

```
Nele = 4;
c = physconst('LightSpeed');
fc = 300e6;
lambda = c/fc;
ha = phased.ULA('NumElements',Nele,'ElementSpacing',lambda/2);
fs = 1e6;
prf = 10e3;
height = 1000;
direction = [90;0];
speed = 2000;
mountingAng = [0,30,0];
```

Create the GPU clutter simulation object and configure it to take a transmitted signal as an input argument. The configuration assumes the earth is flat. The maximum clutter range of interest is 5 km, and the maximum azimuth coverage is  $\pm 60^\circ$ .

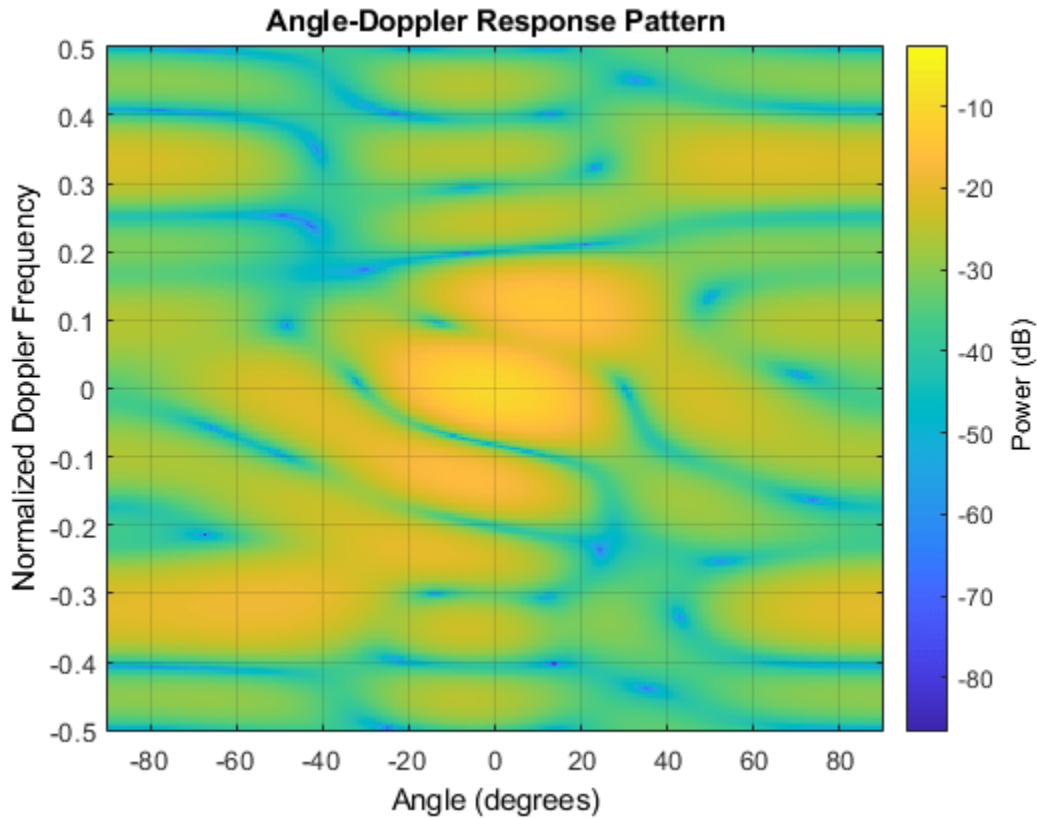
```
Rmax = 5000;
Azcov = 120;
tergamma = 0;
clutter = gpuConstantGammaClutter('Sensor',ha,...
    'PropagationSpeed',c,'OperatingFrequency',fc,'PRF',prf,...
    'SampleRate',fs,'Gamma',tergamma,'EarthModel','Flat',...
    'TransmitSignalInputPort',true,'PlatformHeight',height,...
    'PlatformSpeed',speed,'PlatformDirection',direction,...
    'MountingAngles',mountingAng,'ClutterMaxRange',Rmax,...
    'ClutterAzimuthSpan',Azcov,'SeedSource','Property','Seed',40547);
```

Simulate the clutter return for 10 pulses. At each object call, pass the transmit signal as an input argument. The software automatically computes the effective transmitted power of the signal. The transmit signal is a rectangular waveform with a pulse width of 2  $\mu$ s.

```
tpower = 5000;
pw = 2e-6;
X = tpower*ones(floor(pw*fs),1);
Nsamp = fs/prf;
Npulse = 10;
clsig = zeros(Nsamp,Nele,Npulse);
for m = 1:Npulse
    clsig(:, :, m) = clutter(X);
end
```

Plot the angle-Doppler response of the clutter at the 20th range bin.

```
response = phased.AngleDopplerResponse('SensorArray',ha,...
    'OperatingFrequency',fc,'PropagationSpeed',c,'PRF',prf);
plotResponse(response,shiftdim(clsig(20,:,:)),...
    'NormalizeDoppler',true);
```



The results are not identical to the results obtained by using `constantGammaClutter` because of differences between CPU and GPU computations.

## Tips

The clutter simulation that `constantGammaClutter` provides is based on these assumptions:

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

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **GPU Code Generation**

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

## **See Also**

### **Topics**

Acceleration of Clutter Simulation Using GPU and Code Generation  
Ground Clutter Mitigation with Moving Target Indication (MTI) Radar  
“Clutter Modeling”  
“GPU Computing” (Parallel Computing Toolbox)

# geoTrajectory

Waypoint trajectory in geodetic coordinates

## Description

The `geoTrajectory` System object generates trajectories based on waypoints in geodetic coordinates. When you create the System object, you can specify the time of arrival, velocity, and orientation at each waypoint. The `geoTrajectory` System object involves three coordinate systems. For more details, see “Coordinate Frames in Geo Trajectory” on page 4-526.

To generate an Earth-centered waypoint trajectory in geodetic coordinates:

- 1 Create the `geoTrajectory` object and set its properties.
- 2 Call the object as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#).

## Creation

### Syntax

```
trajectory = geoTrajectory(Waypoints,TimeOfArrival)
trajectory = geoTrajectory(Waypoints,TimeOfArrival,Name,Value)
```

### Description

`trajectory = geoTrajectory(Waypoints,TimeOfArrival)` returns a `geoTrajectory` System object, `trajectory`, based on the specified geodetic waypoints, `Waypoints`, and the corresponding time, `TimeOfArrival`.

`trajectory = geoTrajectory(Waypoints,TimeOfArrival,Name,Value)` sets each creation argument or property `Name` to the specified `Value`. Unspecified properties and creation arguments have default or inferred values.

Example: `trajectory = geoTrajectory([10,10,1000;10,11,1100],[0,3600])` creates a geodetic waypoint trajectory System object, `geojectory`, that moves one degree in longitude and 100 meters in altitude in one hour.

### Creation Arguments

Creation arguments are properties which are set during creation of the System object and cannot be modified later. If you do not explicitly set a creation argument value, the property value is inferred.

You can specify `Waypoints` and `TimeOfArrival` as value-only arguments or name-value pairs.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### **SampleRate** — Sample rate of trajectory (Hz)

1 (default) | positive scalar

Sample rate of the trajectory in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: `double`

### **SamplesPerFrame** — Number of samples per output frame

1 (default) | positive integer

Number of samples per output frame, specified as a positive integer.

**Tunable:** Yes

Data Types: `double`

### **Waypoints** — Positions in geodetic coordinates [deg deg m]

[0 0 0] (default) |  $N$ -by-3 matrix

Positions in geodetic coordinates, specified as an  $N$ -by-3 matrix.  $N$  is the number of waypoints. In each row, the three elements represent the latitude in degrees, longitude in degrees, and altitude above the WGS84 reference ellipsoid in meters of the geodetic waypoint. When  $N = 1$ , the trajectory is at a stationary position.

#### **Dependencies**

To set this property, you must also set valid values for the `TimeOfArrival` property.

Data Types: `double`

### **TimeOfArrival** — Time at each waypoint (s)

`Inf` (default) |  $N$ -element column vector of nonnegative increasing numbers

Time at each waypoint in seconds, specified as an  $N$ -element column vector. The number of samples,  $N$ , must be the same as the number of samples (rows) defined by `Waypoints`. If the trajectory is stationary (only one waypoint specified in the `Waypoints` property), then the specified property value for `TimeOfArrival` is ignored and the default value, `Inf`, is used.

#### **Dependencies**

To set this property, you must also set valid values for the `Waypoints` property.

Data Types: `double`

### **Velocities** — Velocity in local reference frame at each waypoint (m/s)

[0 0 0] (default) |  $N$ -by-3 matrix



Velocity in the local reference frame at each waypoint in meters per second, specified as an  $N$ -by-3 matrix. The number of samples,  $N$ , must be the same as the number of samples (rows) defined by `Waypoints`.

- If you do not specify the velocity, the object infers velocities from waypoints.
- If you specify the velocity as a non-zero value, the object obtains the course of the trajectory accordingly.

Data Types: `double`

### **Course — Angle between velocity direction and North (degree)**

$N$ -element vector of scalars

Angle between the velocity direction and the North direction, specified as an  $N$ -element vector of scalars in degrees. The number of samples,  $N$ , must be the same as the number of samples (rows) defined by `Waypoints`. If neither `Velocities` nor `Course` is specified, course is inferred from the waypoints.

#### **Dependencies**

To set this property, do not specify the `Velocities` property during object creation.

Data Types: `double`

### **GroundSpeed — Groundspeed at each waypoint (m/s)**

$N$ -element real vector

Groundspeed at each waypoint, specified as an  $N$ -element real vector in m/s. If you do not specify the property, it is inferred from the waypoints. The number of samples,  $N$ , must be the same as the number of samples (rows) defined by `Waypoints`.

#### **Dependencies**

To set this property, do not specify the `Velocities` property during object creation.

Data Types: `double`

### **Climbrate — Climb rate at each waypoint (m/s)**

$N$ -element real vector

Climb rate at each waypoint, specified as an  $N$ -element real vector in degrees. The number of samples,  $N$ , must be the same as the number of samples (rows) defined by `Waypoints`. If neither `Velocities` nor `Course` is specified, climb rate is inferred from the waypoints.

#### **Dependencies**

To set this property, do not specify the `Velocities` property during object creation.

Data Types: `double`

### **Orientation — Orientation at each waypoint**

$N$ -element quaternion column vector | 3-by-3-by- $N$  array of real numbers

Orientation at each waypoint, specified as an  $N$ -element quaternion column vector or as a 3-by-3-by- $N$  array of real numbers in which each 3-by-3 array is a rotation matrix. The number of quaternions or rotation matrices,  $N$ , must be the same as the number of samples (rows) defined by `Waypoints`.

Each quaternion or rotation matrix is a frame rotation from the local reference frame (NED or ENU) at the waypoint to the body frame of the platform on the trajectory.

Data Types: `quaternion` | `double`

#### **AutoPitch — Align pitch angle with direction of motion**

`false` (default) | `true`

Align pitch angle with the direction of motion, specified as `true` or `false`. When specified as `true`, the pitch angle aligns with the direction of motion. If specified as `false`, the pitch angle is set to zero.

#### **Dependencies**

To set this property, the `Orientation` property must not be specified during object creation.

#### **AutoBank — Align roll angle to counteract centripetal force**

`false` (default) | `true`

Align the roll angle to counteract the centripetal force, specified as `true` or `false`. When specified as `true`, the roll angle automatically counteracts the centripetal force. If specified as `false`, the roll angle is set to zero (flat orientation).

#### **Dependencies**

To set this property, do not specify the `Orientation` property during object creation.

#### **ReferenceFrame — Local reference frame of trajectory**

`'NED'` (default) | `'ENU'`

Local reference frame of the trajectory, specified as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up). The local reference frame corresponds to the current waypoint of the trajectory. The velocity, acceleration, and orientation of the platform are reported in the local reference frame. For more details, see “Coordinate Frames in Geo Trajectory” on page 4-526.

## **Usage**

### **Syntax**

```
[positionLLA,orientation,velocity,acceleration,angularVelocity,ecef2ref] =  
trajectory()
```

### **Description**

`[positionLLA,orientation,velocity,acceleration,angularVelocity,ecef2ref] = trajectory()` outputs a frame of trajectory data based on specified creation arguments and properties, where `trajectory` is a `geoTrajectory` object.

### **Output Arguments**

**positionLLA — Geodetic positions in latitude, longitude, and altitude (deg deg m)**

*M*-by-3 matrix

Geodetic positions in latitude, longitude, and altitude, returned as an  $M$ -by-3 matrix. In each row, the three elements represent the latitude in degrees, longitude in degrees, and altitude above the WGS84 reference ellipsoid in meters of the geodetic waypoint.

$M$  is specified by the `SamplesPerFrame` property.

Data Types: `double`

#### **orientation — Orientation in local reference coordinate system**

$M$ -element quaternion column vector | 3-by-3-by- $M$  real array

Orientation in the local reference coordinate system, returned as an  $M$ -by-1 quaternion column vector or as a 3-by-3-by- $M$  real array in which each 3-by-3 array is a rotation matrix.

Each quaternion or rotation matrix is a frame rotation from the local reference frame (NED or ENU) to the body frame.

$M$  is specified by the `SamplesPerFrame` property.

Data Types: `double`

#### **velocity — Velocity in local reference coordinate system (m/s)**

$M$ -by-3 matrix

Velocity in the local reference coordinate system in meters per second, returned as an  $M$ -by-3 matrix.

$M$  is specified by the `SamplesPerFrame` property.

Data Types: `double`

#### **acceleration — Acceleration in local reference coordinate system (m/s<sup>2</sup>)**

$M$ -by-3 matrix

Acceleration in the local reference coordinate system in meters per second squared, returned as an  $M$ -by-3 matrix.

$M$  is specified by the `SamplesPerFrame` property.

Data Types: `double`

#### **angularVelocity — Angular velocity in local reference coordinate system (rad/s)**

$M$ -by-3 matrix

Angular velocity in the local reference coordinate system in radians per second, returned as an  $M$ -by-3 matrix.

$M$  is specified by the `SamplesPerFrame` property.

Data Types: `double`

#### **ecef2ref — Orientation of local reference frame with respect to ECEF frame**

$M$ -element quaternion column vector | 3-by-3-by- $M$  real array

Orientation of the local reference frame with respect to the ECEF (Earth-Centered-Earth-Fixed) frame, returned as an  $M$ -by-1 quaternion column vector or as a 3-by-3-by- $M$  real array in which each 3-by-3 array is a rotation matrix.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the ECEF frame to the local reference frame (NED or ENU) corresponding to the current waypoint.

$M$  is specified by the `SamplesPerFrame` property.

Data Types: `double`

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `geoTrajectory`

`lookupPose` Obtain pose of geodetic trajectory for a certain time  
`perturbations` Perturbation defined on object  
`perturb` Apply perturbations to object

### Common to All System Objects

`clone` Create duplicate System object  
`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object  
`isDone` End-of-data status

## Examples

### Create `geoTrajectory` and Look Up Pose

Create a `geoTrajectory` with starting LLA at [15 15 0] and ending LLA at [75 75 100]. Set the flight time to ten hours. Sample the trajectory every 1000 seconds.

```
startLLA = [15 15 0];  
endLLA = [75 75 100];  
timeOfTravel = [0 3600*10];  
sampleRate = 0.001;
```

```
trajectory = geoTrajectory([startLLA;endLLA],timeOfTravel,'SampleRate',sampleRate);
```

Output the LLA waypoints of the trajectory.

```
positionsLLA = startLLA;  
while ~isDone(trajectory)  
    positionsLLA = [positionsLLA;trajectory()];  
end  
positionsLLA
```

```
positionsLLA = 37×3
```

```
15.0000 15.0000 0  
16.6667 16.6667 2.7778
```

```

18.3333  18.3333  5.5556
20.0000  20.0000  8.3333
21.6667  21.6667  11.1111
23.3333  23.3333  13.8889
25.0000  25.0000  16.6667
26.6667  26.6667  19.4444
28.3333  28.3333  22.2222
30.0000  30.0000  25.0000
:

```

Look up the Cartesian waypoints of the trajectory in the ECEF frame by using the `lookupPose` function.

```

sampleTimes = 0:1000:3600*10;
n = length(sampleTimes);
positionsCart = lookupPose(trajjectory, sampleTimes, 'ECEF');

```

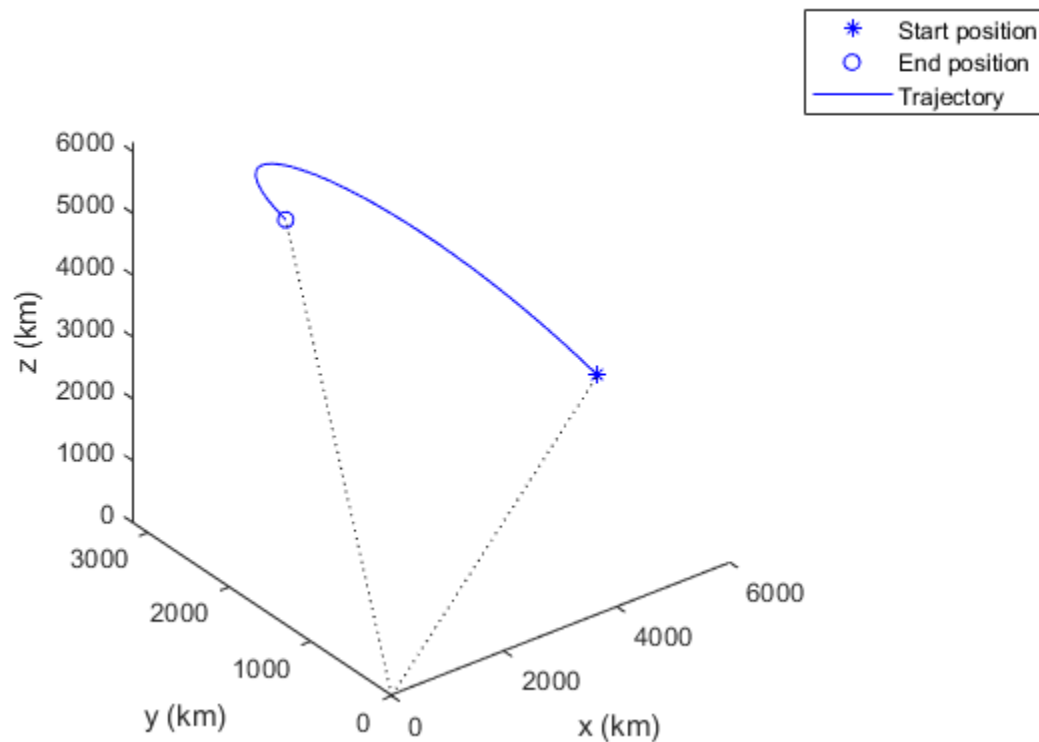
Visualize the results in the ECEF frame.

```

figure()
km = 1000;
plot3(positionsCart(1,1)/km, positionsCart(1,2)/km, positionsCart(1,3)/km, 'b*');
hold on;
plot3(positionsCart(end,1)/km, positionsCart(end,2)/km, positionsCart(end,3)/km, 'bo');
plot3(positionsCart(:,1)/km, positionsCart(:,2)/km, positionsCart(:,3)/km, 'b');

plot3([0 positionsCart(1,1)]/km, [0 positionsCart(1,2)]/km, [0 positionsCart(1,3)]/km, 'k:');
plot3([0 positionsCart(end,1)]/km, [0 positionsCart(end,2)]/km, [0 positionsCart(end,3)]/km, 'k:');
xlabel('x (km)'); ylabel('y (km)'); zlabel('z (km)');
legend('Start position', 'End position', 'Trajectory')

```



## Algorithms

### Coordinate Frames in Geo Trajectory

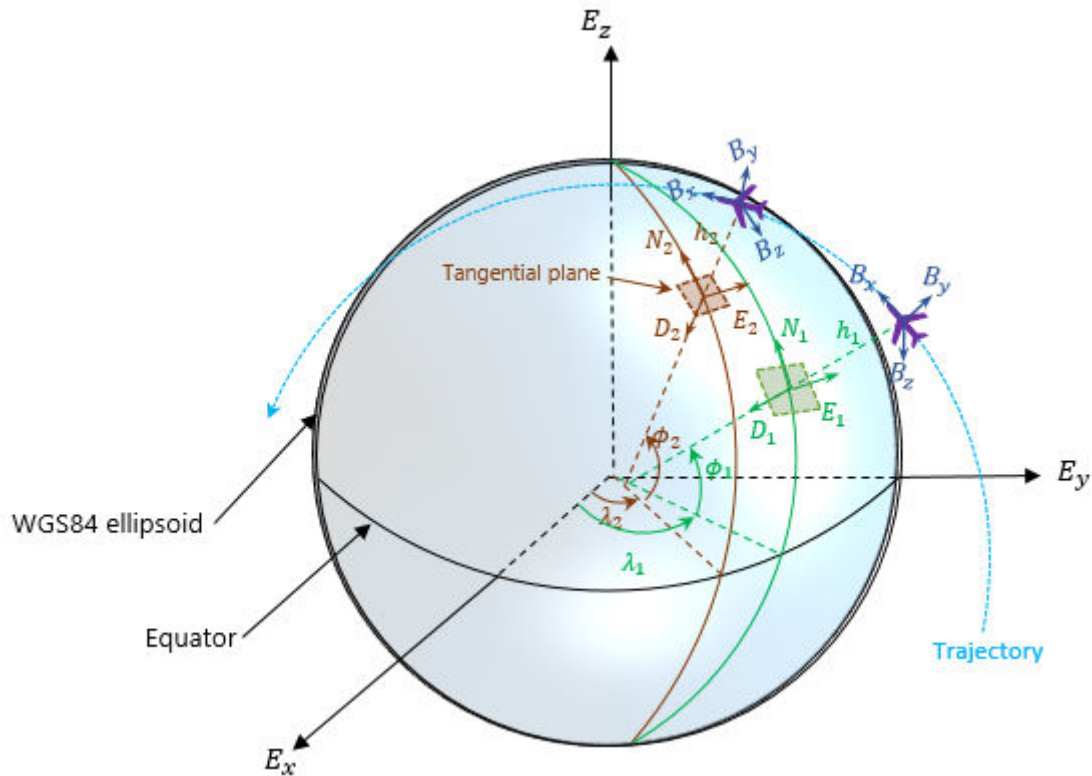
The geoTrajectory System object involves three coordinate frames:

- ECEF (Earth-Centered-Earth-Fixed) frame
- Local reference frame: local NED (North-East-Down) or ENU (East-North-Up) frame
- Target body frame

The figure shows an Earth-centered trajectory with two waypoints highlighted. The figure uses the **NED** local reference frame as an example, but you can certainly use the ENU local reference frame. In the figure,

- $E_x$ ,  $E_y$ , and  $E_z$  are the three axes of the ECEF frame, which is fixed on the Earth.
- $B_x$ ,  $B_y$ , and  $B_z$  are the three axes of the target body frame, which is fixed on the target.
- $N$ ,  $E$ , and  $D$  are the three axes of the local NED frame. The figure highlights two local NED reference frames,  $N_1-E_1-D_1$  and  $N_2-E_2-D_2$ . The origin of each local NED frame is the Earth surface point corresponding to the trajectory waypoint based on the WGS84 ellipsoid model. The horizontal plane of the local NED frame is tangent to the WGS84 ellipsoid model's surface.

$\lambda$  and  $\phi$  are the geodetic longitude and latitude, respectively. The orientation of the target by using the NED local frame convention is defined as the rotation from the local NED frame to the target's body frame, such as the rotation from  $N_1$ - $E_1$ - $D_1$  to  $B_x$ - $B_y$ - $B_z$ .



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The object functions, perturbations and perturb, do not support code generation.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

waypointTrajectory | kinematicTrajectory

**Introduced in R2021a**

## lookupPose

Obtain pose of geodetic trajectory for a certain time

### Syntax

```
[position,orientation,velocity,acceleration,angularVelocity,ecef2ref] =
lookupPose(traj,sampleTimes)
[ ___ ] = lookupPose(traj,sampleTimes,coordinateSystem)
```

### Description

[position,orientation,velocity,acceleration,angularVelocity,ecef2ref] = lookupPose(traj,sampleTimes) returns the pose information of the waypoint trajectory at the specified sample times. If any sample time is beyond the duration of the trajectory, the corresponding pose information is returned as NaN.

[ \_\_\_ ] = lookupPose(traj,sampleTimes,coordinateSystem) additionally enables you to specify the format of the position output.

### Examples

#### Create geoTrajectory and Look Up Pose

Create a geoTrajectory with starting LLA at [15 15 0] and ending LLA at [75 75 100]. Set the flight time to ten hours. Sample the trajectory every 1000 seconds.

```
startLLA = [15 15 0];
endLLA = [75 75 100];
timeOfTravel = [0 3600*10];
sampleRate = 0.001;
```

```
trajectory = geoTrajectory([startLLA;endLLA],timeOfTravel,'SampleRate',sampleRate);
```

Output the LLA waypoints of the trajectory.

```
positionsLLA = startLLA;
while ~isDone(trajectory)
    positionsLLA = [positionsLLA;trajectory()];
end
```

```
positionsLLA
```

```
positionsLLA = 37×3
```

```
15.0000    15.0000         0
16.6667    16.6667    2.7778
18.3333    18.3333    5.5556
20.0000    20.0000    8.3333
21.6667    21.6667   11.1111
23.3333    23.3333   13.8889
25.0000    25.0000   16.6667
26.6667    26.6667   19.4444
```



```

28.3333  28.3333  22.2222
30.0000  30.0000  25.0000
⋮

```

Look up the Cartesian waypoints of the trajectory in the ECEF frame by using the `lookupPose` function.

```

sampleTimes = 0:1000:3600*10;
n = length(sampleTimes);
positionsCart = lookupPose(trajjectory,sampleTimes,'ECEF');

```

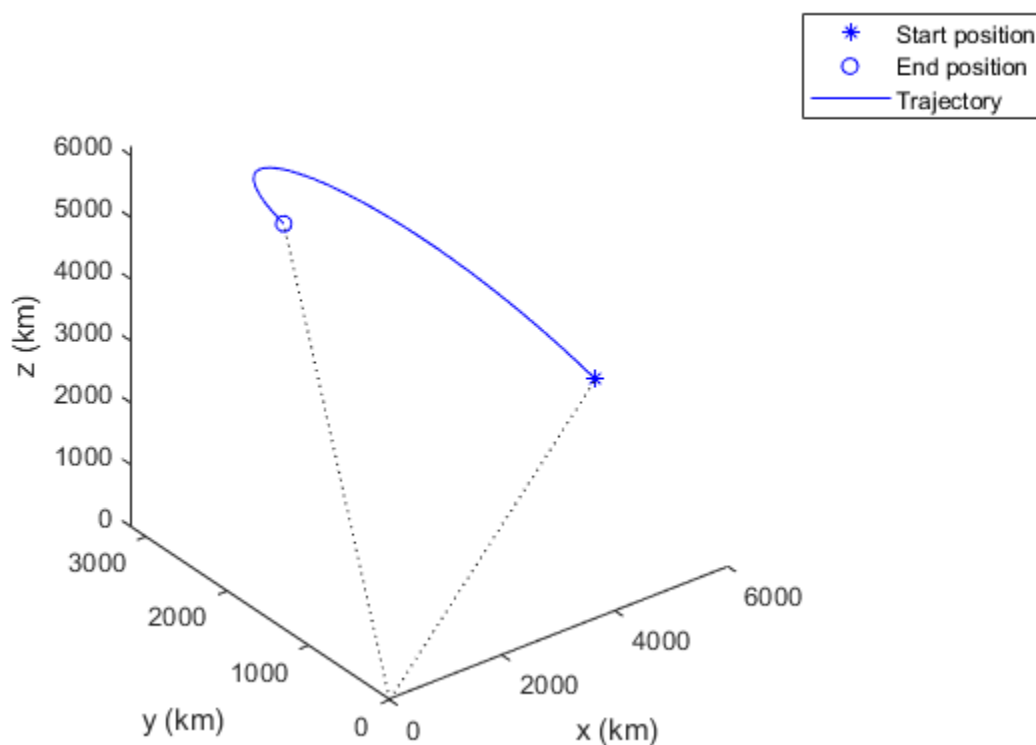
Visualize the results in the ECEF frame.

```

figure()
km = 1000;
plot3(positionsCart(1,1)/km,positionsCart(1,2)/km,positionsCart(1,3)/km, 'b*');
hold on;
plot3(positionsCart(end,1)/km,positionsCart(end,2)/km,positionsCart(end,3)/km, 'bo');
plot3(positionsCart(:,1)/km,positionsCart(:,2)/km,positionsCart(:,3)/km,'b');

plot3([0 positionsCart(1,1)]/km,[0 positionsCart(1,2)]/km,[0 positionsCart(1,3)]/km,'k:');
plot3([0 positionsCart(end,1)]/km,[0 positionsCart(end,2)]/km,[0 positionsCart(end,3)]/km,'k:');
xlabel('x (km)'); ylabel('y (km)'); zlabel('z (km)');
legend('Start position','End position', 'Trajectory')

```



## Input Arguments

### **traj** — Geodetic trajectory

geoTrajectory object

Geodetic trajectory, specified as a geoTrajectory object.

### **sampleTimes** — Sample times

$K$ -element vector of nonnegative scalar

Sample times in seconds, specified as an  $K$ -element vector of nonnegative scalars.

### **coordinateSystem** — Coordinate system to report positions

'LLA' (default) | 'ECEF'

Coordinate system to report positions, specified as:

- 'LLA' — Report positions as latitude in degrees, longitude in degrees, and altitude above the WGS84 reference ellipsoid in meters.
- 'ECEF' — Report positions as Cartesian coordinates in the ECEF (Earth-Centered-Earth-Fixed) coordinate frame in meters.

.

## Output Arguments

### **position** — Positions in local reference coordinate system (deg deg m)

$K$ -by-3 matrix

Geodetic positions in local reference coordinate system, returned as a  $K$ -by-3 matrix.  $K$  is the number of SampleTimes.

- When the coordinateSystem input is specified as 'LLA', the three elements in each row represent the latitude in degrees, longitude in degrees, and altitude above the WGS84 reference ellipsoid in meters of the geodetic waypoint.
- When the coordinateSystem input is specified as 'ECEF', the three elements in each row represent the Cartesian position coordinates in the ECEF (Earth-Centered-Earth-Fixed) coordinate frame in meters.

Data Types: double

### **orientation** — Orientation in local reference coordinate system

$K$ -element quaternion column vector | 3-by-3-by- $K$  real array

Orientation in the local reference coordinate system, returned as a  $K$ -by-1 quaternion column vector or as a 3-by-3-by- $K$  real array in which each 3-by-3 matrix is a rotation matrix.

Each quaternion or rotation matrix is a frame rotation from the local reference frame (NED or ENU) at the waypoint to the body frame of the target on the trajectory.

$K$  is the number of SampleTimes.

Data Types: double

**velocity — Velocity in local reference coordinate system (m/s)***K*-by-3 matrix

Velocity in the local reference coordinate system in meters per second, returned as an *M*-by-3 matrix.

*K* is specified by the `SamplesPerFrame` property.

Data Types: `double`

**acceleration — Acceleration in local reference coordinate system (m/s<sup>2</sup>)***K*-by-3 matrix

Acceleration in the local reference coordinate system in meters per second squared, returned as an *M*-by-3 matrix.

*K* is the number of `SampleTimes`.

Data Types: `double`

**angularVelocity — Angular velocity in local reference coordinate system (rad/s)***K*-by-3 matrix

Angular velocity in the local reference coordinate system in radians per second, returned as a *K*-by-3 matrix.

*K* is the number of `SampleTimes`.

Data Types: `double`

**ecef2ref — Orientation of reference frame with respect to ECEF frame***K*-element quaternion column vector | 3-by-3-by-*M* real array

Orientation of the reference frame with respect to the ECEF (Earth-Centered-Earth-Fixed) frame, returned as a *K*-by-1 quaternion column vector or as a 3-by-3-by-*K* real array, in which each 3-by-3 matrix is a rotation matrix.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the ECEF frame to the local reference frame (NED or ENU) at the current trajectory position.

*K* is the number of `SampleTimes`.

Data Types: `double`

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`geoTrajectory`

**Introduced in R2021a**

# kinematicTrajectory

Rate-driven trajectory generator

## Description

The `kinematicTrajectory` System object generates trajectories using specified acceleration and angular velocity.

To generate a trajectory from rates:

- 1 Create the `kinematicTrajectory` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
trajectory = kinematicTrajectory  
trajectory = kinematicTrajectory(Name,Value)
```

### Description

`trajectory = kinematicTrajectory` returns a System object, `trajectory`, that generates a trajectory based on acceleration and angular velocity.

`trajectory = kinematicTrajectory(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `trajectory = kinematicTrajectory('SampleRate',200,'Position',[0,1,10])` creates a kinematic trajectory System object, `trajectory`, with a sample rate of 200 Hz and the initial position set to `[0,1,10]`.

## Properties

If a property is *tunable*, you can change its value at any time.

### SampleRate — Sample rate of trajectory (Hz)

100 (default) | positive scalar

Sample rate of trajectory in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: `single` | `double`

### Position — Position state in local navigation coordinate system (m)

[0 0 0] (default) | 3-element row vector

Position state in the local navigation coordinate system in meters, specified as a three-element row vector.

**Tunable:** Yes

Data Types: `single` | `double`

### **Velocity — Velocity state in local navigation coordinate system (m/s)**

`[0 0 0]` (default) | 3-element row vector

Velocity state in the local navigation coordinate system in m/s, specified as a three-element row vector.

**Tunable:** Yes

Data Types: `single` | `double`

### **Orientation — Orientation state in local navigation coordinate system**

`quaternion(1,0,0,0)` (default) | scalar quaternion | 3-by-3 real matrix

Orientation state in the local navigation coordinate system, specified as a scalar quaternion or 3-by-3 real matrix. The orientation is a frame rotation from the local navigation coordinate system to the current body frame.

**Tunable:** Yes

Data Types: `quaternion` | `single` | `double`

### **AccelerationSource — Source of acceleration state**

`'Input'` (default) | `'Property'`

Source of acceleration state, specified as `'Input'` or `'Property'`.

- `'Input'` -- specify acceleration state as an input argument to the kinematic trajectory object
- `'Property'` -- specify acceleration state by setting the `Acceleration` property

**Tunable:** No

Data Types: `char` | `string`

### **Acceleration — Acceleration state (m/s<sup>2</sup>)**

`[0 0 0]` (default) | three-element row vector

Acceleration state in m/s<sup>2</sup>, specified as a three-element row vector.

**Tunable:** Yes

#### **Dependencies**

To enable this property, set `AccelerationSource` to `'Property'`.

Data Types: `single` | `double`

### **AngularVelocitySource — Source of angular velocity state**

`'Input'` (default) | `'Property'`

Source of angular velocity state, specified as `'Input'` or `'Property'`.

- 'Input' -- specify angular velocity state as an input argument to the kinematic trajectory object
- 'Property' -- specify angular velocity state by setting the AngularVelocity property

**Tunable:** No

Data Types: char | string

**AngularVelocity — Angular velocity state (rad/s)**

[0 0 0] (default) | three-element row vector

Angular velocity state in rad/s, specified as a three-element row vector.

**Tunable:** Yes

**Dependencies**

To enable this property, set AngularVelocitySource to 'Property'.

Data Types: single | double

**SamplesPerFrame — Number of samples per output frame**

1 (default) | positive integer

Number of samples per output frame, specified as a positive integer.

**Tunable:** No

**Dependencies**

To enable this property, set AngularVelocitySource to 'Property' and AccelerationSource to 'Property'.

Data Types: single | double

## Usage

## Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = trajectory(  
bodyAcceleration,bodyAngularVelocity)  
[position,orientation,velocity,acceleration,angularVelocity] = trajectory(  
bodyAngularVelocity)  
[position,orientation,velocity,acceleration,angularVelocity] = trajectory(  
bodyAcceleration)  
[position,orientation,velocity,acceleration,angularVelocity] = trajectory()
```

## Description

[position,orientation,velocity,acceleration,angularVelocity] = trajectory(bodyAcceleration,bodyAngularVelocity) outputs the trajectory state and then updates the trajectory state based on bodyAcceleration and bodyAngularVelocity.

This syntax is only valid if AngularVelocitySource is set to 'Input' and AccelerationSource is set to 'Input'.

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory(bodyAngularVelocity)` outputs the trajectory state and then updates the trajectory state based on `bodyAngularAcceleration`.

This syntax is only valid if `AngularVelocitySource` is set to 'Input' and `AccelerationSource` is set to 'Property'.

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory(bodyAcceleration)` outputs the trajectory state and then updates the trajectory state based on `bodyAcceleration`.

This syntax is only valid if `AngularVelocitySource` is set to 'Property' and `AccelerationSource` is set to 'Input'.

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory()` outputs the trajectory state and then updates the trajectory state.

This syntax is only valid if `AngularVelocitySource` is set to 'Property' and `AccelerationSource` is set to 'Property'.

### Input Arguments

#### **bodyAcceleration — Acceleration in body coordinate system (m/s<sup>2</sup>)**

*N*-by-3 matrix

Acceleration in the body coordinate system in meters per second squared, specified as an *N*-by-3 matrix.

*N* is the number of samples in the current frame.

#### **bodyAngularVelocity — Angular velocity in body coordinate system (rad/s)**

*N*-by-3 matrix

Angular velocity in the body coordinate system in radians per second, specified as an *N*-by-3 matrix.

*N* is the number of samples in the current frame.

### Output Arguments

#### **position — Position in local navigation coordinate system (m)**

*N*-by-3 matrix

Position in the local navigation coordinate system in meters, returned as an *N*-by-3 matrix.

*N* is the number of samples in the current frame.

Data Types: `single` | `double`

#### **orientation — Orientation in local navigation coordinate system**

*N*-element quaternion column vector | 3-by-3-by-*N* real array

Orientation in the local navigation coordinate system, returned as an *N*-by-1 quaternion column vector or a 3-by-3-by-*N* real array. Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system.

*N* is the number of samples in the current frame.

Data Types: `single` | `double`

**velocity** — Velocity in local navigation coordinate system (m/s)

*N*-by-3 matrix

Velocity in the local navigation coordinate system in meters per second, returned as an *N*-by-3 matrix.

*N* is the number of samples in the current frame.

Data Types: `single` | `double`

**acceleration** — Acceleration in local navigation coordinate system (m/s<sup>2</sup>)

*N*-by-3 matrix

Acceleration in the local navigation coordinate system in meters per second squared, returned as an *N*-by-3 matrix.

*N* is the number of samples in the current frame.

Data Types: `single` | `double`

**angularVelocity** — Angular velocity in local navigation coordinate system (rad/s)

*N*-by-3 matrix

Angular velocity in the local navigation coordinate system in radians per second, returned as an *N*-by-3 matrix.

*N* is the number of samples in the current frame.

Data Types: `single` | `double`

## Object Functions

### Specific to `kinematicTrajectory`

`perturbations` Perturbation defined on object  
`perturb` Apply perturbations to object

### Common to All System Objects

`step` Run System object algorithm

## Examples

### Create Default `kinematicTrajectory`

Create a default `kinematicTrajectory` System object™ and explore the relationship between input, properties, and the generated trajectories.

```
trajectory = kinematicTrajectory  
  
trajectory =  
    kinematicTrajectory with properties:  
        SampleRate: 100
```



```

        Position: [0 0 0]
        Orientation: [1x1 quaternion]
        Velocity: [0 0 0]
        AccelerationSource: 'Input'
        AngularVelocitySource: 'Input'

```

By default, the `kinematicTrajectory` object has an initial position of `[0 0 0]` and an initial velocity of `[0 0 0]`. Orientation is described by a quaternion one ( $1 + 0i + 0j + 0k$ ).

The `kinematicTrajectory` object maintains a visible and writable state in the properties `Position`, `Velocity`, and `Orientation`. When you call the object, the state is output and then updated.

For example, call the object by specifying an acceleration and angular velocity relative to the body coordinate system.

```

bodyAcceleration = [5,5,0];
bodyAngularVelocity = [0,0,1];
[position,orientation,velocity,acceleration,angularVelocity] = trajectory(bodyAcceleration,bodyAngularVelocity)

```

```
position = 1x3
```

```
    0    0    0
```

```
orientation = quaternion
```

```
    1 + 0i + 0j + 0k
```

```
velocity = 1x3
```

```
    0    0    0
```

```
acceleration = 1x3
```

```
    5    5    0
```

```
angularVelocity = 1x3
```

```
    0    0    1
```

The position, orientation, and velocity output from the `trajectory` object correspond to the state reported by the properties before calling the object. The `trajectory` state is updated after being called and is observable from the properties:

```
trajectory
```

```
trajectory =
```

```
    kinematicTrajectory with properties:
```

```

        SampleRate: 100
        Position: [2.5000e-04 2.5000e-04 0]
        Orientation: [1x1 quaternion]
        Velocity: [0.0500 0.0500 0]
        AccelerationSource: 'Input'

```

```
AngularVelocitySource: 'Input'
```

The acceleration and angularVelocity output from the trajectory object correspond to the bodyAcceleration and bodyAngularVelocity, except that they are returned in the navigation coordinate system. Use the orientation output to rotate acceleration and angularVelocity to the body coordinate system and verify they are approximately equivalent to bodyAcceleration and bodyAngularVelocity.

```
rotatedAcceleration = rotatepoint(orientation,acceleration)
```

```
rotatedAcceleration = 1×3
```

```
5    5    0
```

```
rotatedAngularVelocity = rotatepoint(orientation,angularVelocity)
```

```
rotatedAngularVelocity = 1×3
```

```
0    0    1
```

The kinematicTrajectory System object™ enables you to modify the trajectory state through the properties. Set the position to [0,0,0] and then call the object with a specified acceleration and angular velocity in the body coordinate system. For illustrative purposes, clone the trajectory object before modifying the Position property. Call both objects and observe that the positions diverge.

```
trajectoryClone = clone(trajectory);
trajectory.Position = [0,0,0];
```

```
position = trajectory(bodyAcceleration,bodyAngularVelocity)
```

```
position = 1×3
```

```
0    0    0
```

```
clonePosition = trajectoryClone(bodyAcceleration,bodyAngularVelocity)
```

```
clonePosition = 1×3
```

```
10-3 ×
```

```
0.2500    0.2500    0
```

### Create Oscillating Trajectory

This example shows how to create a trajectory oscillating along the North axis of a local NED coordinate system using the kinematicTrajectory System object™.

Create a default kinematicTrajectory object. The default initial orientation is aligned with the local NED coordinate system.

```
traj = kinematicTrajectory
```

```

traj =
    kinematicTrajectory with properties:
        SampleRate: 100
        Position: [0 0 0]
        Orientation: [1x1 quaternion]
        Velocity: [0 0 0]
        AccelerationSource: 'Input'
        AngularVelocitySource: 'Input'

```

Define a trajectory for a duration of 10 seconds consisting of rotation around the East axis (pitch) and an oscillation along North axis of the local NED coordinate system. Use the default `kinematicTrajectory` sample rate.

```

fs = traj.SampleRate;
duration = 10;

numSamples = duration*fs;

cyclesPerSecond = 1;
samplesPerCycle = fs/cyclesPerSecond;
numCycles = ceil(numSamples/samplesPerCycle);
maxAccel = 20;

triangle = [linspace(maxAccel,1/fs-maxAccel,samplesPerCycle/2), ...
            linspace(-maxAccel,maxAccel-(1/fs),samplesPerCycle/2)'];
oscillation = repmat(triangle,numCycles,1);
oscillation = oscillation(1:numSamples);

accNED = [zeros(numSamples,2),oscillation];

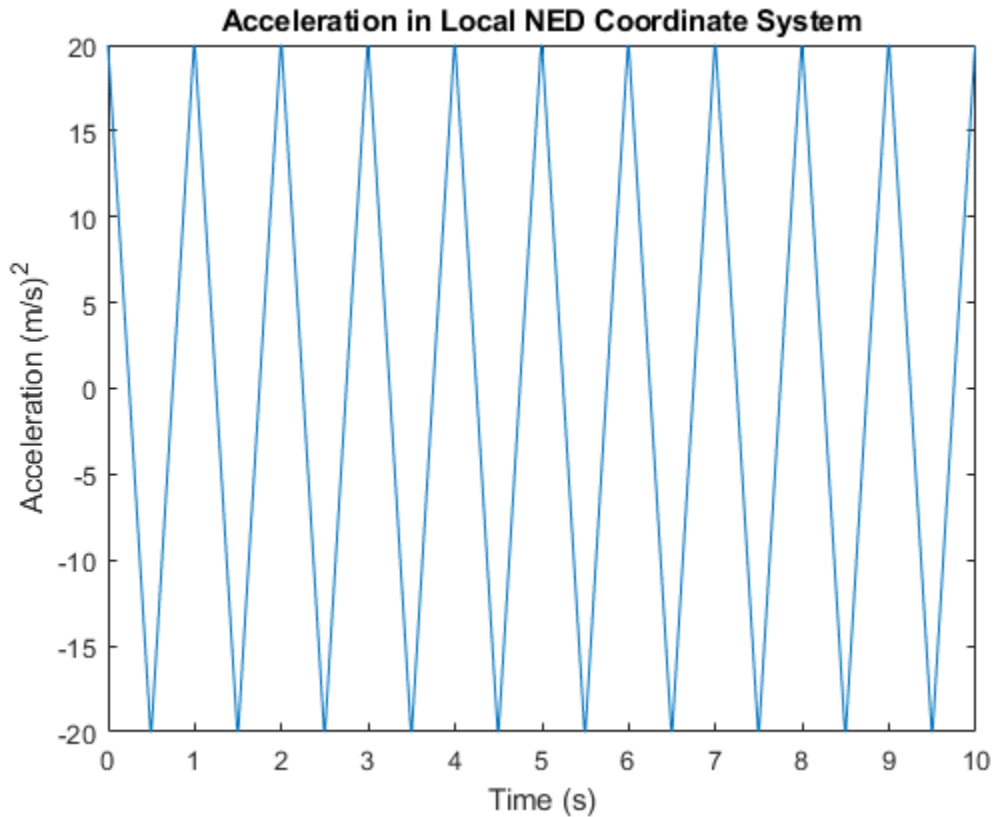
angVelNED = zeros(numSamples,3);
angVelNED(:,2) = 2*pi;

Plot the acceleration control signal.

timeVector = 0:1/fs:(duration-1/fs);

figure(1)
plot(timeVector,oscillation)
xlabel('Time (s)')
ylabel('Acceleration (m/s)^2')
title('Acceleration in Local NED Coordinate System')

```



Generate the trajectory sample-by-sample in a loop. The `kinematicTrajectory` System object assumes the acceleration and angular velocity inputs are in the local sensor body coordinate system. Rotate the acceleration and angular velocity control signals from the NED coordinate system to the sensor body coordinate system using `rotateframe` and the `Orientation` state. Update a 3-D plot of the position at each time. Add `pause` to mimic real-time processing. Once the loop is complete, plot the position over time. Rotating the `accNED` and `angVelNED` control signals to the local body coordinate system assures the motion stays along the Down axis.

```
figure(2)
plotHandle = plot3(traj.Position(1),traj.Position(2),traj.Position(3),'bo');
grid on
xlabel('North')
ylabel('East')
zlabel('Down')
axis([-1 1 -1 1 0 1.5])
hold on

q = ones(numSamples,1,'quaternion');
for ii = 1:numSamples
    accBody = rotateframe(traj.Orientation,accNED(ii,:));
    angVelBody = rotateframe(traj.Orientation,angVelNED(ii,:));

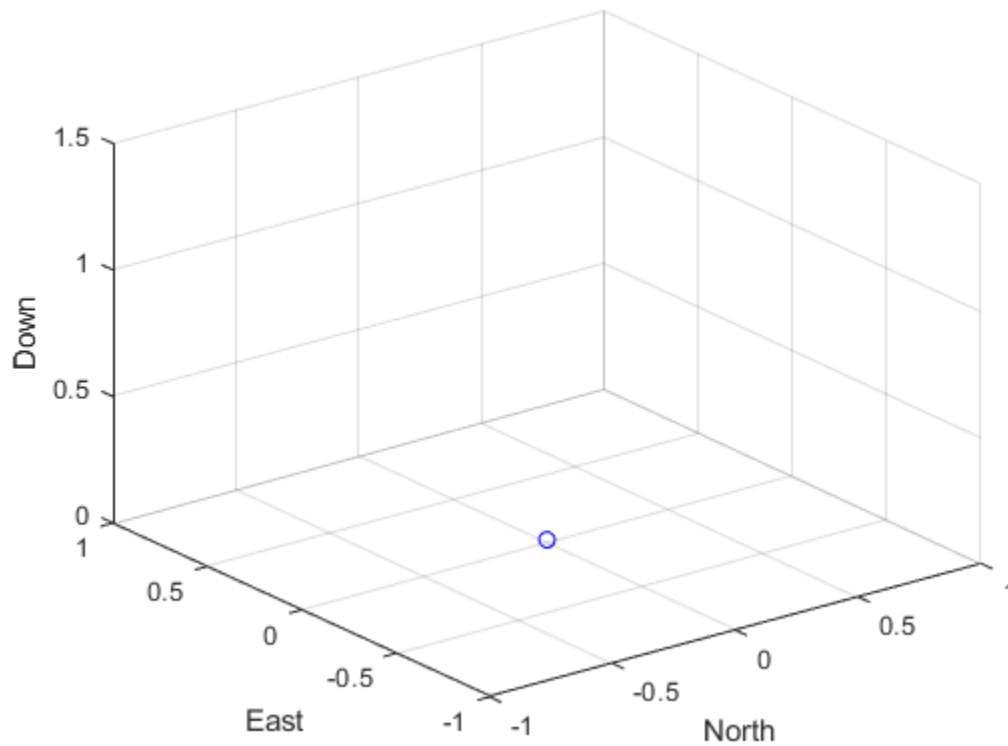
    [pos(ii,:),q(ii),vel,ac] = traj(accBody,angVelBody);

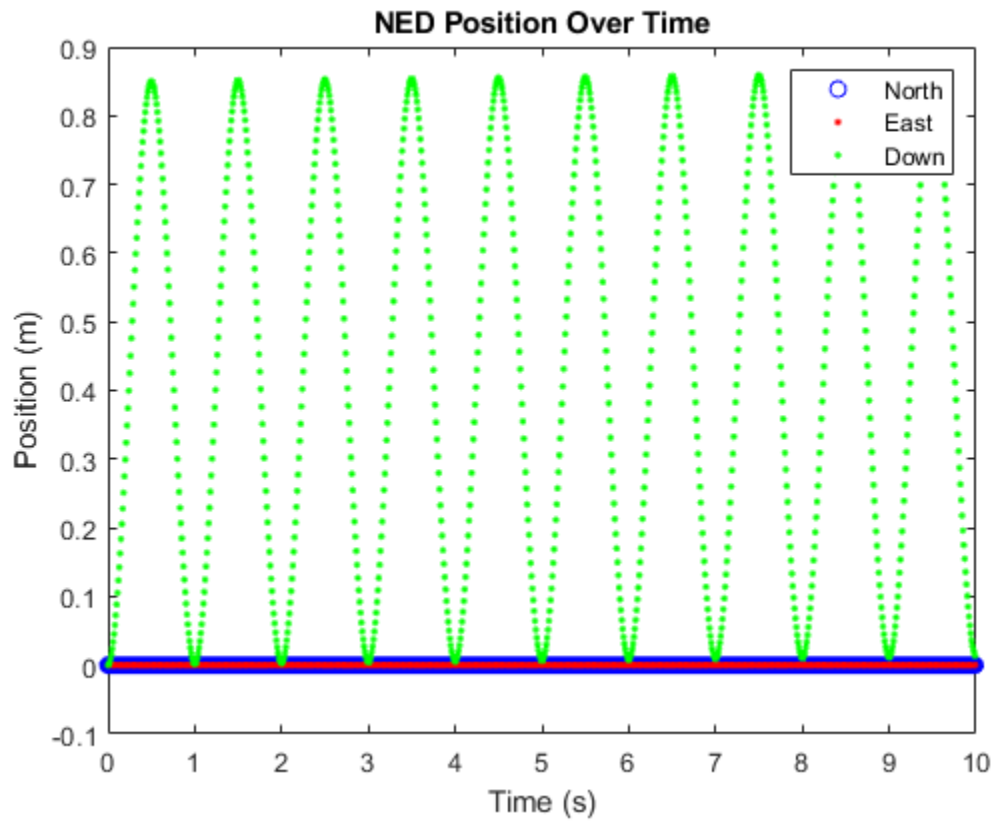
    set(plotHandle,'XData',pos(ii,1),'YData',pos(ii,2),'ZData',pos(ii,3))

    pause(1/fs)
```

```
end  
  
figure(3)  
plot(timeVector,pos(:,1),'bo',...  
      timeVector,pos(:,2),'r.',...  
      timeVector,pos(:,3),'g.')
```

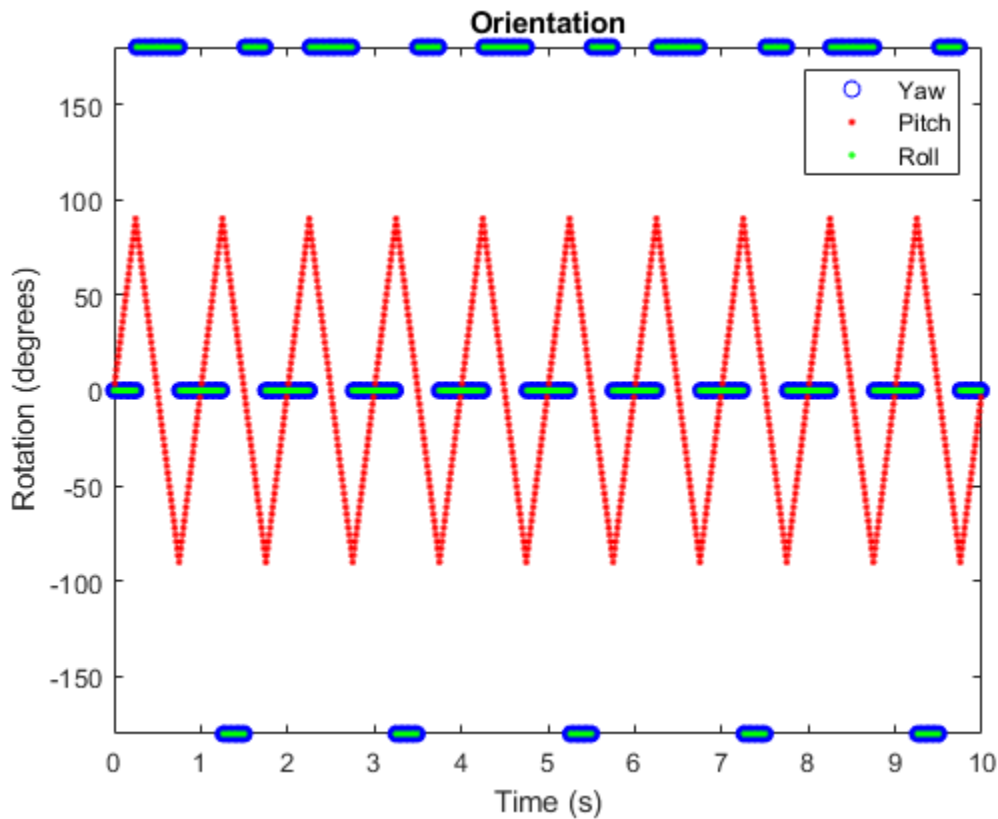
xlabel('Time (s)')  
ylabel('Position (m)')  
title('NED Position Over Time')  
legend('North','East','Down')





Convert the recorded orientation to Euler angles and plot. Although the orientation of the platform changed over time, the acceleration always acted along the North axis.

```
figure(4)
eulerAngles = eulerd(q, 'ZYX', 'frame');
plot(timeVector, eulerAngles(:,1), 'bo', ...
      timeVector, eulerAngles(:,2), 'r.', ...
      timeVector, eulerAngles(:,3), 'g.')
axis([0, duration, -180, 180])
legend('Yaw', 'Pitch', 'Roll')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation')
```



### Generate a Coil Trajectory

This example shows how to generate a coil trajectory using the kinematicTrajectory System object™.

Create a circular trajectory for a 1000 second duration and a sample rate of 10 Hz. Set the radius of the circle to 5000 meters and the speed to 80 meters per second. Set the climb rate to 100 meters per second and the pitch to 15 degrees. Specify the initial orientation as pointed in the direction of motion.

```
duration = 1000; % seconds
fs = 10; % Hz
N = duration*fs; % number of samples

radius = 5000; % meters
speed = 80; % meters per second
climbRate = 50; % meters per second
initialYaw = 90; % degrees
pitch = 15; % degrees

initPos = [radius, 0, 0];
initVel = [0, speed, climbRate];
initOrientation = quaternion([initialYaw,pitch,0], 'eulerd', 'zyx', 'frame');

trajectory = kinematicTrajectory('SampleRate',fs, ...
```

```
'Velocity',initVel, ...  
'Position',initPos, ...  
'Orientation',initOrientation);
```

Specify a constant acceleration and angular velocity in the body coordinate system. Rotate the body frame to account for the pitch.

```
accBody = zeros(N,3);  
accBody(:,2) = speed^2/radius;  
accBody(:,3) = 0.2;
```

```
angVelBody = zeros(N,3);  
angVelBody(:,3) = speed/radius;
```

```
pitchRotation = quaternion([0,pitch,0], 'eulerd', 'zyx', 'frame');  
angVelBody = rotateframe(pitchRotation,angVelBody);  
accBody = rotateframe(pitchRotation,accBody);
```

Call `trajectory` with the specified acceleration and angular velocity in the body coordinate system. Plot the position, orientation, and speed over time.

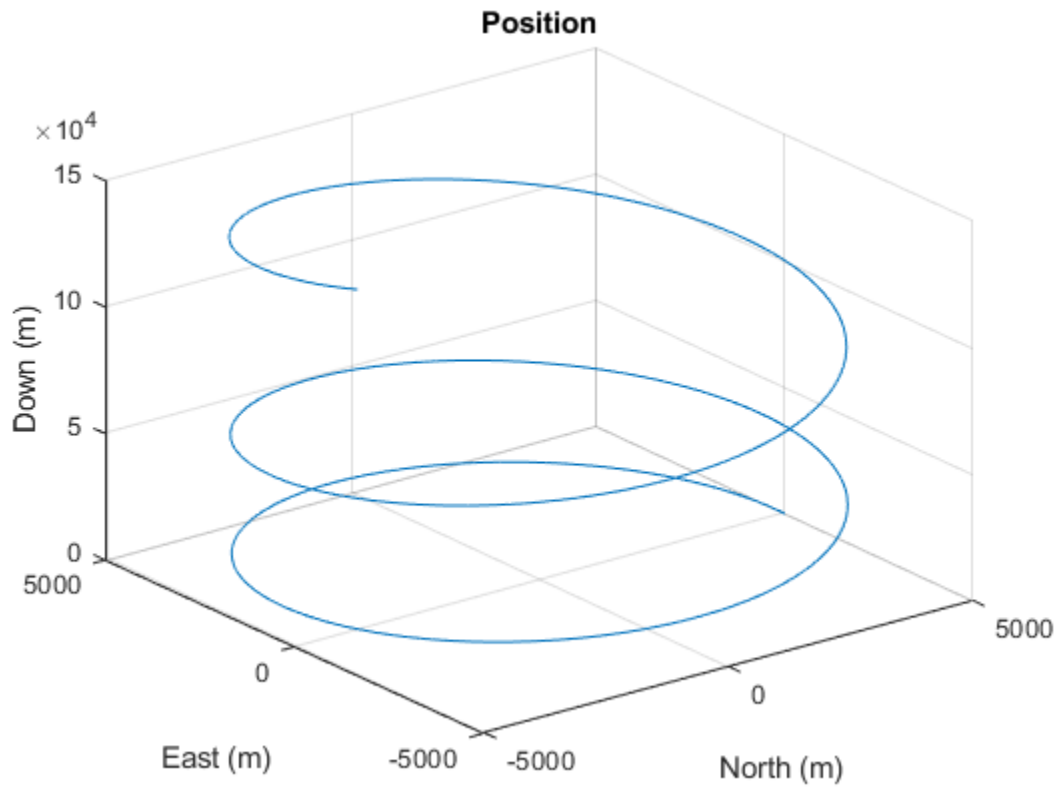
```
[position, orientation, velocity] = trajectory(accBody,angVelBody);
```

```
eulerAngles = eulerd(orientation, 'ZYX', 'frame');  
speed = sqrt(sum(velocity.^2,2));
```

```
timeVector = (0:(N-1))/fs;
```

```
figure(1)  
plot3(position(:,1),position(:,2),position(:,3))  
xlabel('North (m)')  
ylabel('East (m)')  
zlabel('Down (m)')  
title('Position')  
grid on
```

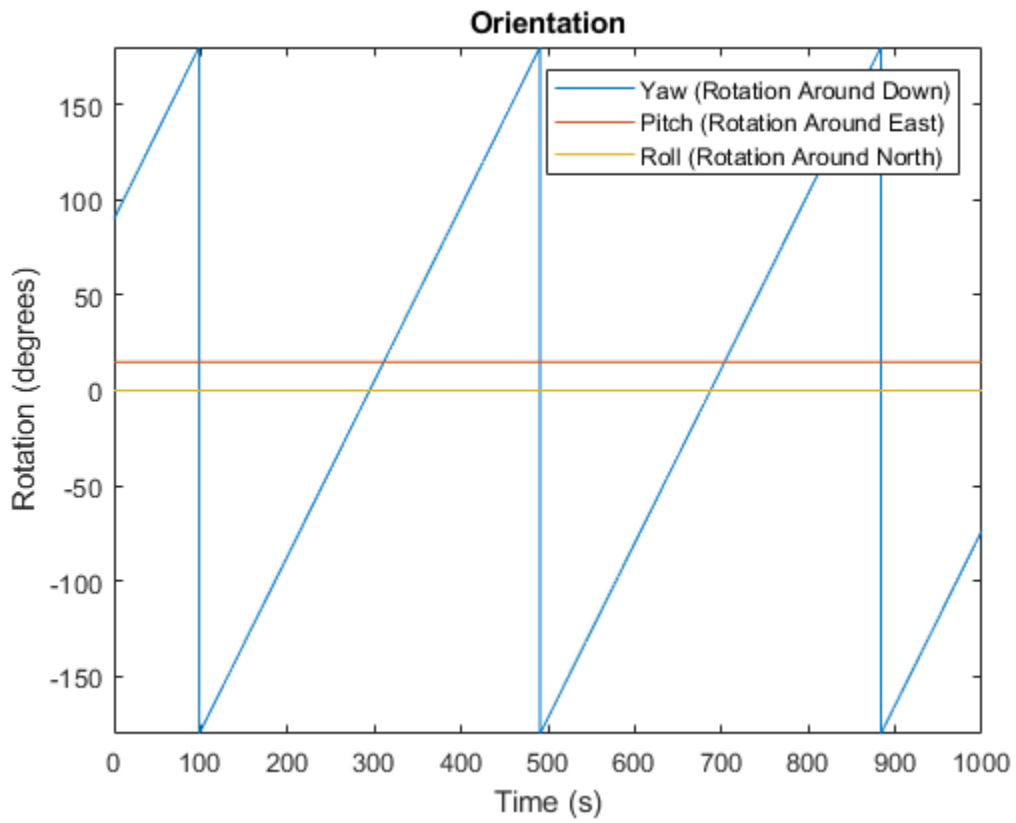




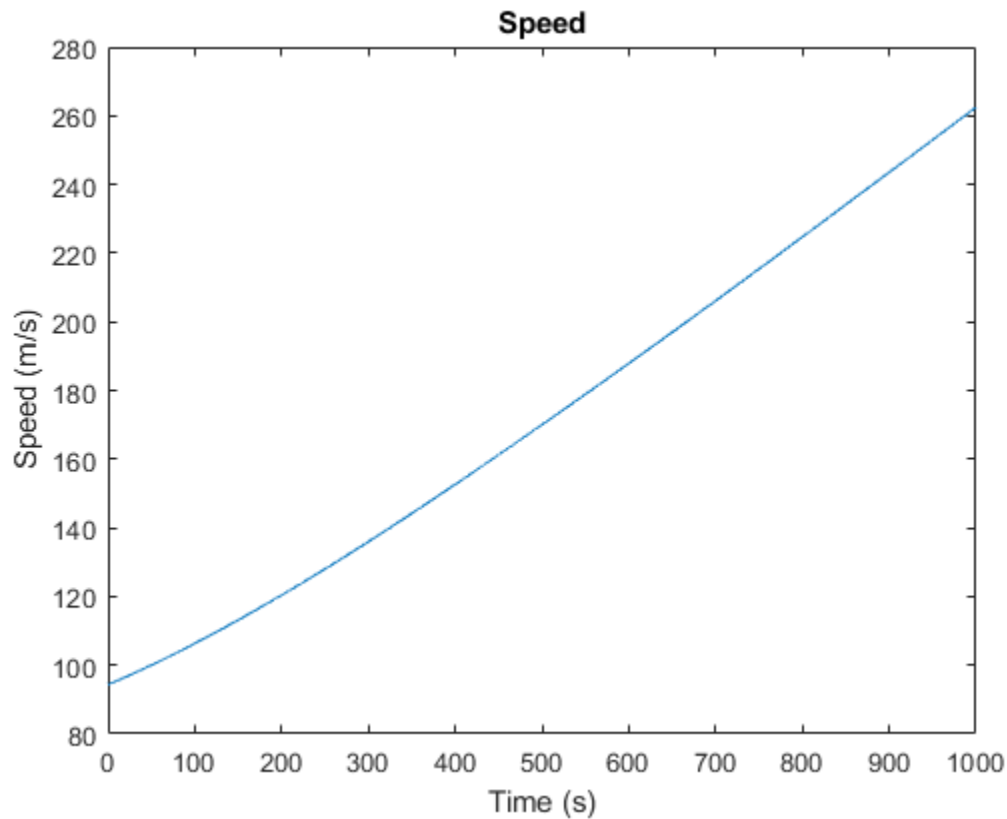
```

figure(2)
plot(timeVector,eulerAngles(:,1),...
      timeVector,eulerAngles(:,2),...
      timeVector,eulerAngles(:,3))
axis([0,duration,-180,180])
legend('Yaw (Rotation Around Down)', 'Pitch (Rotation Around East)', 'Roll (Rotation Around North)')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation')

```



```
figure(3)
plot(timeVector,speed)
xlabel('Time (s)')
ylabel('Speed (m/s)')
title('Speed')
```



### Generate Spiraling Circular Trajectory with No Inputs

Define a constant angular velocity and constant acceleration that describe a spiraling circular trajectory.

```
Fs = 100;
r = 10;
speed = 2.5;
initialYaw = 90;
```

```
initPos = [r 0 0];
initVel = [0 speed 0];
initOrient = quaternion([initialYaw 0 0], 'eulerd', 'ZYX', 'frame');
```

```
accBody = [0 speed^2/r 0.01];
angVelBody = [0 0 speed/r];
```

Create a kinematic trajectory object.

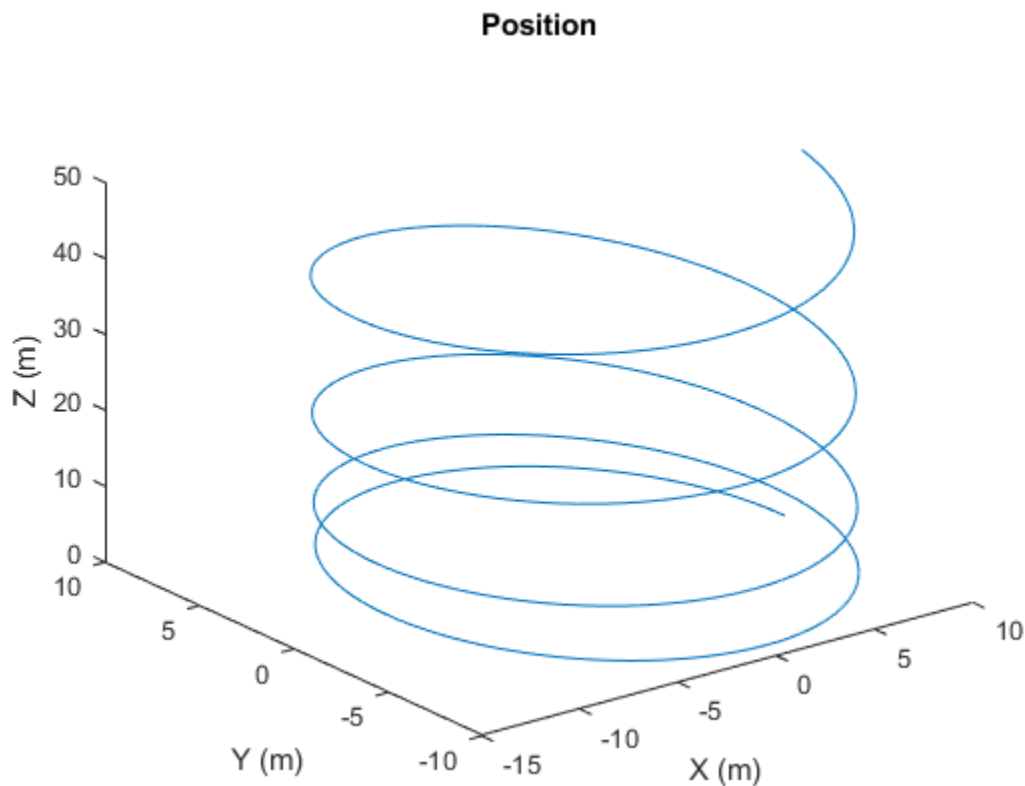
```
traj = kinematicTrajectory('SampleRate',Fs, ...
    'Position',initPos, ...
    'Velocity',initVel, ...
    'Orientation',initOrient, ...
    'AccelerationSource','Property', ...
    'Acceleration',accBody, ...
```

```
'AngularVelocitySource','Property', ...
'AngularVelocity',angVelBody);
```

Call the kinematic trajectory object in a loop and log the position output. Plot the position over time.

```
N = 10000;
pos = zeros(N, 3);
for i = 1:N
    pos(i,:) = traj();
end

plot3(pos(:,1), pos(:,2), pos(:,3))
title('Position')
xlabel('X (m)')
ylabel('Y (m)')
zlabel('Z (m)')
```



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The object functions, `perturbations` and `perturb`, do not support code generation.

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

**See Also**

waypointTrajectory | platform | radarScenario

**Introduced in R2021a**

# waypointTrajectory

Waypoint trajectory generator

## Description

The `waypointTrajectory` System object generates trajectories using specified waypoints. When you create the System object, you can optionally specify the time of arrival, velocity, and orientation at each waypoint. See “Algorithms” on page 4-577 for more details.

To generate a trajectory from waypoints:

- 1 Create the `waypointTrajectory` object and set its properties.
- 2 Call the object as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
trajectory = waypointTrajectory
trajectory = waypointTrajectory(Waypoints,TimeOfArrival)
trajectory = waypointTrajectory(Waypoints,TimeOfArrival,Name,Value)
```

### Description

`trajectory = waypointTrajectory` returns a System object, `trajectory`, that generates a trajectory based on default stationary waypoints.

`trajectory = waypointTrajectory(Waypoints,TimeOfArrival)` specifies the `Waypoints` that the generated trajectory passes through and the `TimeOfArrival` at each waypoint.

`trajectory = waypointTrajectory(Waypoints,TimeOfArrival,Name,Value)` sets each creation argument or property `Name` to the specified `Value`. Unspecified properties and creation arguments have default or inferred values.

Example: `trajectory = waypointTrajectory([10,10,0;20,20,0;20,20,10],[0,0.5,10])` creates a waypoint trajectory System object, `trajectory`, that starts at waypoint `[10,10,0]`, and then passes through `[20,20,0]` after 0.5 seconds and `[20,20,10]` after 10 seconds.

### Creation Arguments

Creation arguments are properties which are set during creation of the System object and cannot be modified later. If you do not explicitly set a creation argument value, the property value is inferred.

If you specify any creation argument, then you must specify both the `Waypoints` and `TimeOfArrival` creation arguments. You can specify `Waypoints` and `TimeOfArrival` as value-only arguments or name-value pairs.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### SampleRate — Sample rate of trajectory (Hz)

100 (default) | positive scalar

Sample rate of trajectory in Hz, specified as a positive scalar.

**Tunable:** Yes

Data Types: double

### SamplesPerFrame — Number of samples per output frame

1 (default) | positive scalar integer

Number of samples per output frame, specified as a positive scalar integer.

**Tunable:** Yes

Data Types: double

### Waypoints — Positions in the navigation coordinate system (m)

$N$ -by-3 matrix

Positions in the navigation coordinate system in meters, specified as an  $N$ -by-3 matrix. The columns of the matrix correspond to the first, second, and third axes, respectively. The rows of the matrix,  $N$ , correspond to individual waypoints.

#### Dependencies

To set this property, you must also set valid values for the `TimeOfArrival` property.

Data Types: double

### TimeOfArrival — Time at each waypoint (s)

$N$ -element column vector of nonnegative increasing numbers

Time corresponding to arrival at each waypoint in seconds, specified as an  $N$ -element column vector. The first element of `TimeOfArrival` must be 0. The number of samples,  $N$ , must be the same as the number of samples (rows) defined by `Waypoints`.

#### Dependencies

To set this property, you must also set valid values for the `Waypoints` property.

Data Types: double

### Velocities — Velocity in navigation coordinate system at each waypoint (m/s)

$N$ -by-3 matrix

Velocity in the navigation coordinate system at each waypoint in meters per second, specified as an  $N$ -by-3 matrix. The columns of the matrix correspond to the first, second, and third axes, respectively.

The number of samples,  $N$ , must be the same as the number of samples (rows) defined by **Waypoints**.

If the velocity is specified as a non-zero value, the object automatically calculates the course of the trajectory. If the velocity is specified as zero, the object infers the course of the trajectory from adjacent waypoints.

**Dependencies**

To set this property, you must also set valid values for the **Waypoints** and **TimeOfArrival** properties.

Data Types: `double`

**Course — Horizontal direction of travel (degree)**

$N$ -element real vector

Horizontal direction of travel, specified as an  $N$ -element real vector in degrees. The number of samples,  $N$ , must be the same as the number of samples (rows) defined by **Waypoints**. If neither **Velocities** nor **Course** is specified, course is inferred from the waypoints.

**Dependencies**

To set this property, the **Velocities** property must not be specified in object creation.

Data Types: `double`

**GroundSpeed — Groundspeed at each waypoint (m/s)**

$N$ -element real vector

Groundspeed at each waypoint, specified as an  $N$ -element real vector in m/s. If the property is not specified, it is inferred from the waypoints. The number of samples,  $N$ , must be the same as the number of samples (rows) defined by **Waypoints**.

**Dependencies**

To set this property, the **Velocities** property must not be specified at object creation.

Data Types: `double`

**CLimbRate — Climb rate at each waypoint (m/s)**

$N$ -element real vector

Climb Rate at each waypoint, specified as an  $N$ -element real vector in degrees. The number of samples,  $N$ , must be the same as the number of samples (rows) defined by **Waypoints**. If neither **Velocities** nor **Course** is specified, climb rate is inferred from the waypoints.

**Dependencies**

To set this property, the **Velocities** property must not be specified at object creation.

Data Types: `double`

**Orientation — Orientation at each waypoint**

$N$ -element quaternion column vector | 3-by-3-by- $N$  array of real numbers

Orientation at each waypoint, specified as an  $N$ -element quaternion column vector or 3-by-3-by- $N$  array of real numbers. Each quaternion must have a norm of 1. Each 3-by-3 rotation matrix must be an orthonormal matrix. The number of quaternions or rotation matrices,  $N$ , must be the same as the number of samples (rows) defined by **Waypoints**.



If `Orientation` is specified by quaternions, the underlying class must be `double`.

#### Dependencies

To set this property, you must also set valid values for the `Waypoints` and `TimeOfArrival` properties.

Data Types: `double`

#### **AutoPitch — Align pitch angle with direction of motion**

`false` (default) | `true`

Align pitch angle with the direction of motion, specified as `true` or `false`. When specified as `true`, the pitch angle automatically aligns with the direction of motion. If specified as `false`, the pitch angle is set to zero (level orientation).

#### Dependencies

To set this property, the `Orientation` property must not be specified at object creation.

#### **AutoBank — Align roll angle to counteract centripetal force**

`false` (default) | `true`

Align roll angle to counteract the centripetal force, specified as `true` or `false`. When specified as `true`, the roll angle automatically counteracts the centripetal force. If specified as `false`, the roll angle is set to zero (flat orientation).

#### Dependencies

To set this property, the `Orientation` property must not be specified at object creation.

#### **ReferenceFrame — Reference frame of trajectory**

`'NED'` (default) | `'ENU'`

Reference frame of the trajectory, specified as `'NED'` (North-East-Down) or `'ENU'` (East-North-Up).

## Usage

### Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = trajectory()
```

#### Description

`[position,orientation,velocity,acceleration,angularVelocity] = trajectory()` outputs a frame of trajectory data based on specified creation arguments and properties.

#### Output Arguments

##### **position — Position in local navigation coordinate system (m)**

*M*-by-3 matrix

Position in the local navigation coordinate system in meters, returned as an *M*-by-3 matrix.

*M* is specified by the `SamplesPerFrame` property.

Data Types: `double`

**orientation — Orientation in local navigation coordinate system**

*M*-element quaternion column vector | 3-by-3-by-*M* real array

Orientation in the local navigation coordinate system, returned as an *M*-by-1 quaternion column vector or a 3-by-3-by-*M* real array.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system.

*M* is specified by the `SamplesPerFrame` property.

Data Types: `double`

**velocity — Velocity in local navigation coordinate system (m/s)**

*M*-by-3 matrix

Velocity in the local navigation coordinate system in meters per second, returned as an *M*-by-3 matrix.

*M* is specified by the `SamplesPerFrame` property.

Data Types: `double`

**acceleration — Acceleration in local navigation coordinate system (m/s<sup>2</sup>)**

*M*-by-3 matrix

Acceleration in the local navigation coordinate system in meters per second squared, returned as an *M*-by-3 matrix.

*M* is specified by the `SamplesPerFrame` property.

Data Types: `double`

**angularVelocity — Angular velocity in local navigation coordinate system (rad/s)**

*M*-by-3 matrix

Angular velocity in the local navigation coordinate system in radians per second, returned as an *M*-by-3 matrix.

*M* is specified by the `SamplesPerFrame` property.

Data Types: `double`

## Object Functions

To use an object function, specify the `System` object as the first input argument. For example, to release system resources of a `System` object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `waypointTrajectory`

<code>waypointInfo</code>	Get waypoint information table
<code>lookupPose</code>	Obtain pose information for certain time
<code>perturbations</code>	Perturbation defined on object
<code>perturb</code>	Apply perturbations to object

## Common to All System Objects

clone Create duplicate System object  
 step Run System object algorithm  
 release Release resources and allow changes to System object property values and input characteristics  
 reset Reset internal states of System object  
 isDone End-of-data status

## Examples

### Create Default waypointTrajectory

```
trajectory = waypointTrajectory

trajectory =
  waypointTrajectory with properties:

    SampleRate: 100
    SamplesPerFrame: 1
    Waypoints: [2x3 double]
    TimeOfArrival: [2x1 double]
    Velocities: [2x3 double]
    Course: [2x1 double]
    GroundSpeed: [2x1 double]
    ClimbRate: [2x1 double]
    Orientation: [2x1 quaternion]
    AutoPitch: 0
    AutoBank: 0
    ReferenceFrame: 'NED'
```

Inspect the default waypoints and times of arrival by calling `waypointInfo`. By default, the waypoints indicate a stationary position for one second.

```
waypointInfo(trajectory)

ans=2x2 table
  TimeOfArrival   Waypoints
  _____   _____
           0         0     0     0
           1         0     0     0
```

### Create Square Trajectory

Create a square trajectory and examine the relationship between waypoint constraints, sample rate, and the generated trajectory.

Create a square trajectory by defining the vertices of the square. Define the orientation at each waypoint as pointing in the direction of motion. Specify a 1 Hz sample rate and use the default `SamplesPerFrame` of 1.

```
waypoints = [0,0,0; ... % Initial position
            0,1,0; ...
            1,1,0; ...
            1,0,0; ...
            0,0,0]; % Final position

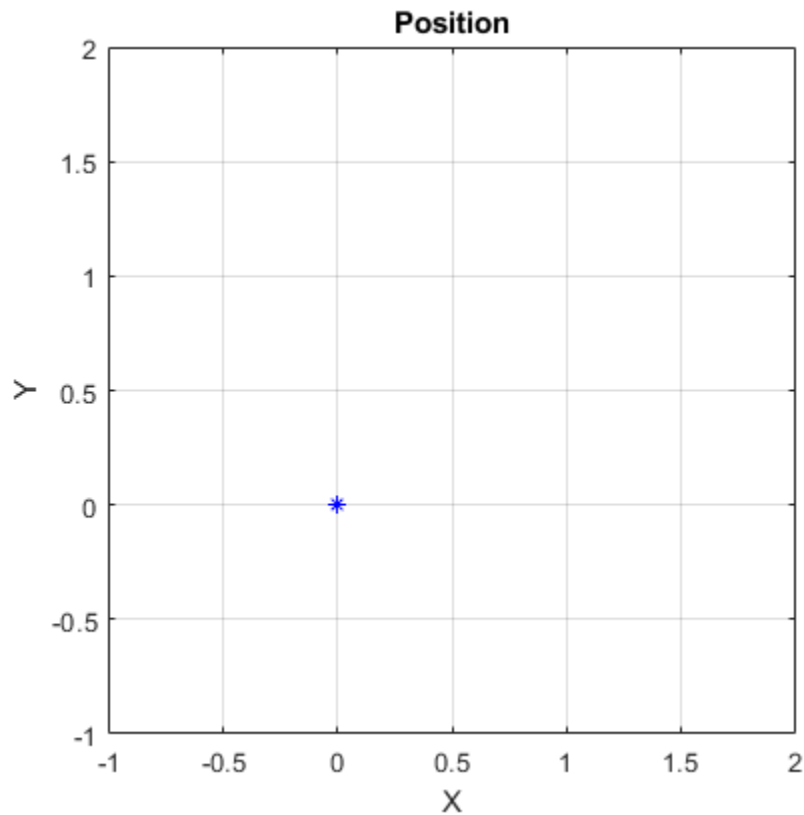
toa = 0:4; % time of arrival

orientation = quaternion([0,0,0; ...
                        45,0,0; ...
                        135,0,0; ...
                        225,0,0; ...
                        0,0,0], ...
                        'eulerd','ZYX','frame');

trajectory = waypointTrajectory(waypoints, ...
                                'TimeOfArrival',toa, ...
                                'Orientation',orientation, ...
                                'SampleRate',1);
```

Create a figure and plot the initial position of the platform.

```
figure(1)
plot(waypoints(1,1),waypoints(1,2),'b*')
title('Position')
axis([-1,2,-1,2])
axis square
xlabel('X')
ylabel('Y')
grid on
hold on
```

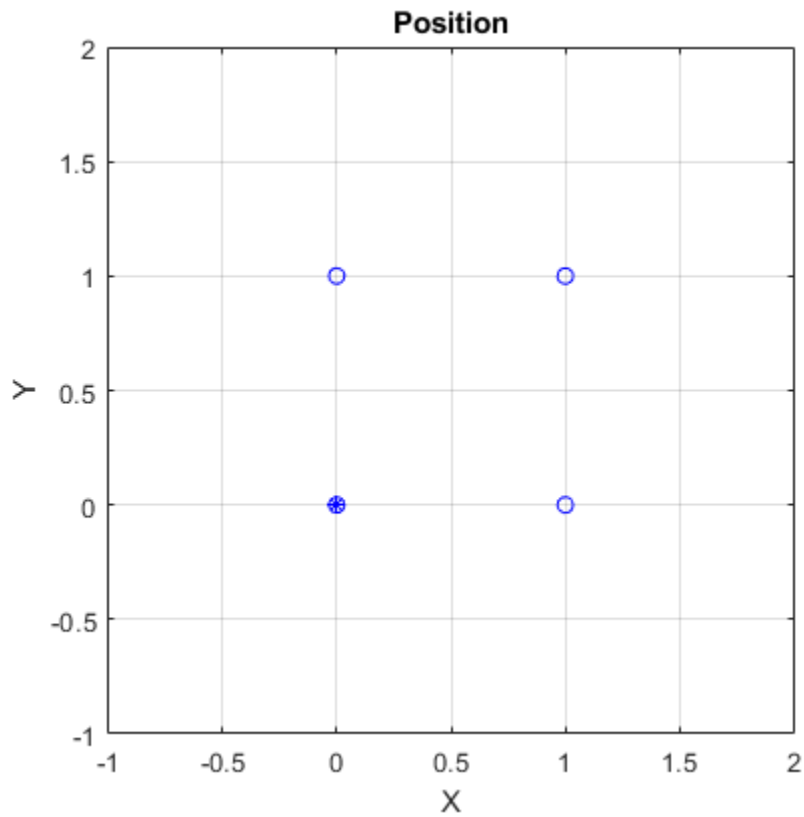


In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

```
orientationLog = zeros(toa(end)*trajectory.SampleRate,1,'quaternion');
count = 1;
while ~isDone(trajectory)
    [currentPosition,orientationLog(count)] = trajectory();

    plot(currentPosition(1),currentPosition(2),'bo')

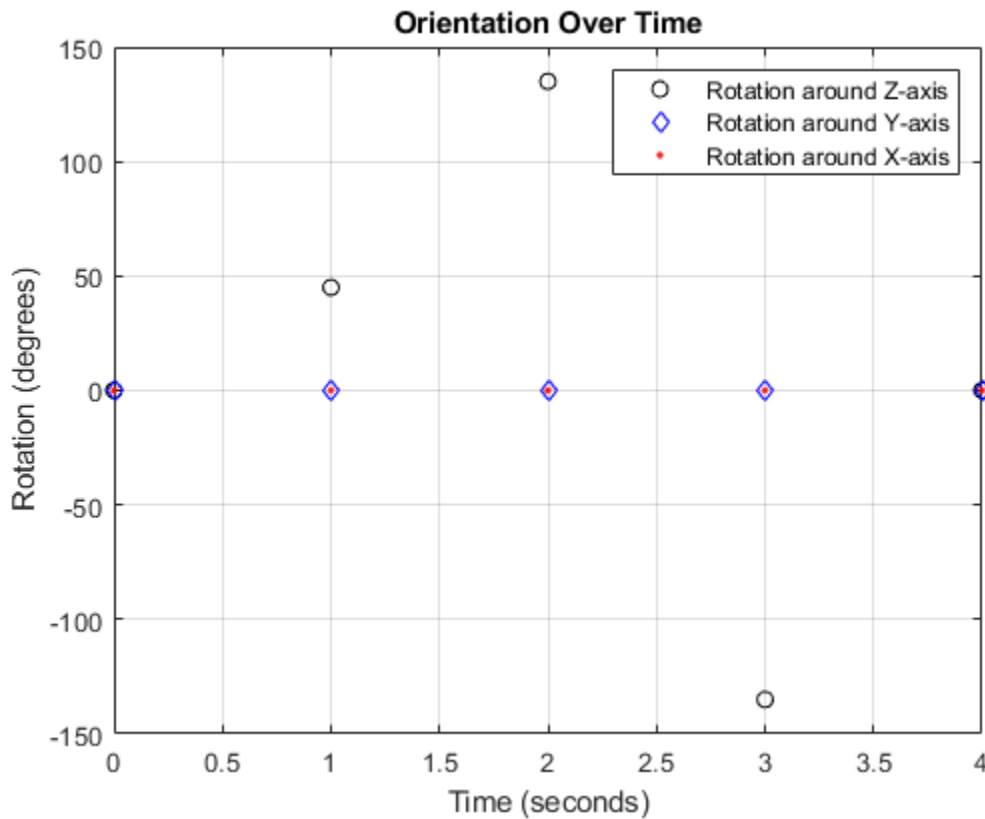
    pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
    count = count + 1;
end
hold off
```



Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog], 'ZYX', 'frame');
plot(toa,eulerAngles(:,1),'ko', ...
      toa,eulerAngles(:,2),'bd', ...
      toa,eulerAngles(:,3),'r.');
```

title('Orientation Over Time')  
 legend('Rotation around Z-axis','Rotation around Y-axis','Rotation around X-axis')  
 xlabel('Time (seconds)')  
 ylabel('Rotation (degrees)')  
 grid on



So far, the trajectory object has only output the waypoints that were specified during construction. To interpolate between waypoints, increase the sample rate to a rate faster than the time of arrivals of the waypoints. Set the trajectory sample rate to 100 Hz and call reset.

```
trajectory.SampleRate = 100;
reset(trajectory)
```

Create a figure and plot the initial position of the platform. In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use pause to mimic real-time processing.

```
figure(1)
plot(waypoints(1,1),waypoints(1,2), 'b*')
title('Position')
axis([-1,2,-1,2])
axis square
xlabel('X')
ylabel('Y')
grid on
hold on

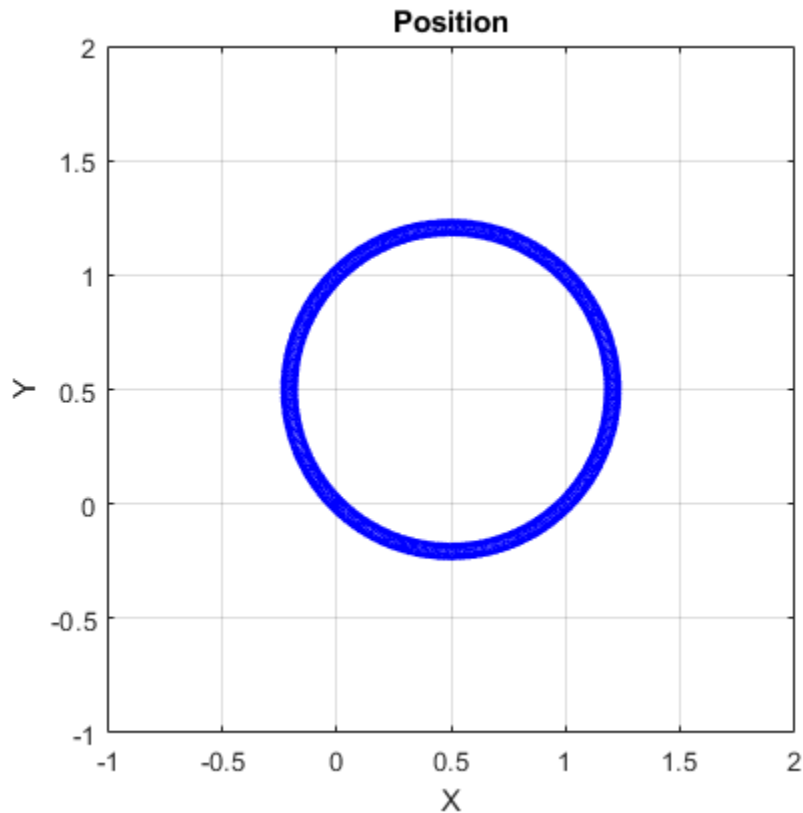
orientationLog = zeros(toa(end)*trajectory.SampleRate,1,'quaternion');
count = 1;
while ~isDone(trajectory)
    [currentPosition,orientationLog(count)] = trajectory();

    plot(currentPosition(1),currentPosition(2), 'bo')
```

```

    pause(trajjectory.SamplesPerFrame/trajjectory.SampleRate)
    count = count + 1;
end
hold off

```



The trajectory output now appears circular. This is because the `waypointTrajectory System object™` minimizes the acceleration and angular velocity when interpolating, which results in smoother, more realistic motions in most scenarios.

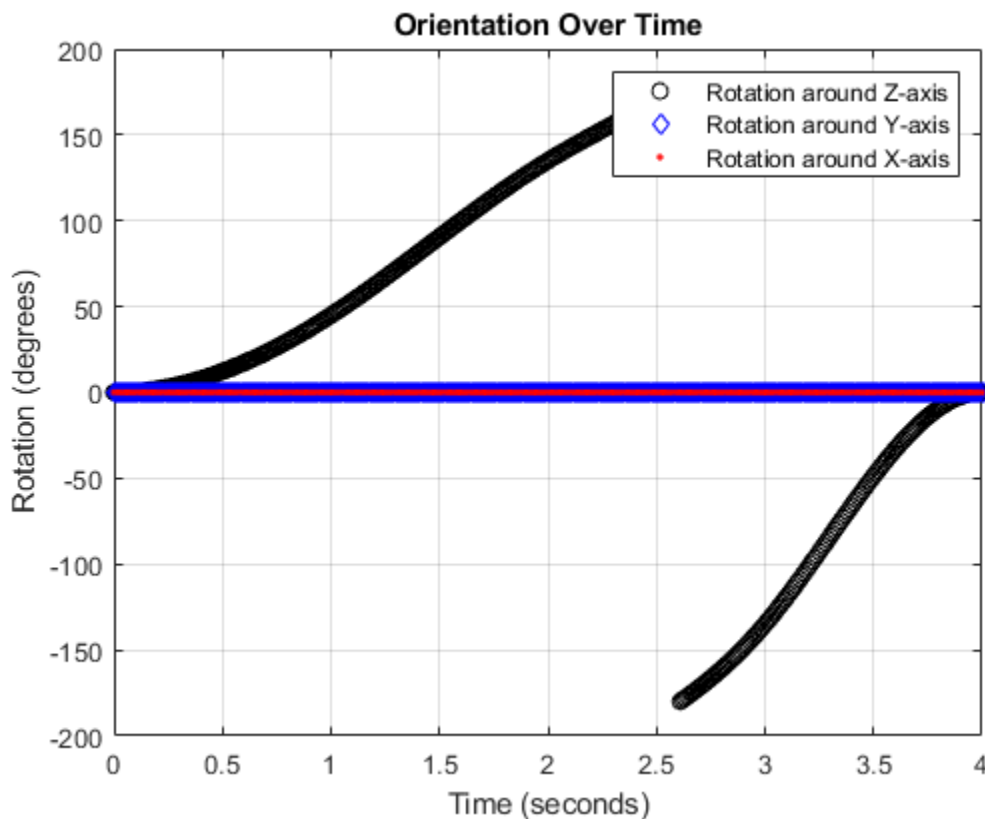
Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time. The orientation is also interpolated.

```

figure(2)
eulerAngles = eulerd([orientation(1);orientationLog], 'ZYX', 'frame');
t = 0:1/trajjectory.SampleRate:4;
plot(t,eulerAngles(:,1),'ko', ...
      t,eulerAngles(:,2),'bd', ...
      t,eulerAngles(:,3),'r. ');
title('Orientation Over Time')
legend('Rotation around Z-axis', 'Rotation around Y-axis', 'Rotation around X-axis')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on

```





The waypointTrajectory algorithm interpolates the waypoints to create a smooth trajectory. To return to the square trajectory, provide more waypoints, especially around sharp changes. To track corresponding times, waypoints, and orientation, specify all the trajectory info in a single matrix.

```

% Time, Waypoint, Orientation
trajectoryInfo = [0, 0,0,0, 0,0,0; ... % Initial position
                 0.1, 0,0,1,0, 0,0,0; ...
                 0.9, 0,0,0,9,0, 0,0,0; ...
                 1, 0,1,0, 45,0,0; ...
                 1.1, 0,1,1,0, 90,0,0; ...
                 1.9, 0,9,1,0, 90,0,0; ...
                 2, 1,1,0, 135,0,0; ...
                 2.1, 1,0,9,0, 180,0,0; ...
                 2.9, 1,0,1,0, 180,0,0; ...
                 3, 1,0,0, 225,0,0; ...
                 3.1, 0,9,0,0, 270,0,0; ...
                 3.9, 0,1,0,0, 270,0,0; ...
                 4, 0,0,0, 270,0,0]; % Final position

trajectory = waypointTrajectory(trajectoryInfo(:,2:4), ...
    'TimeOfArrival',trajectoryInfo(:,1), ...
    'Orientation',quaternion(trajectoryInfo(:,5:end),'eulerd','ZYX','frame'), ...
    'SampleRate',100);

```

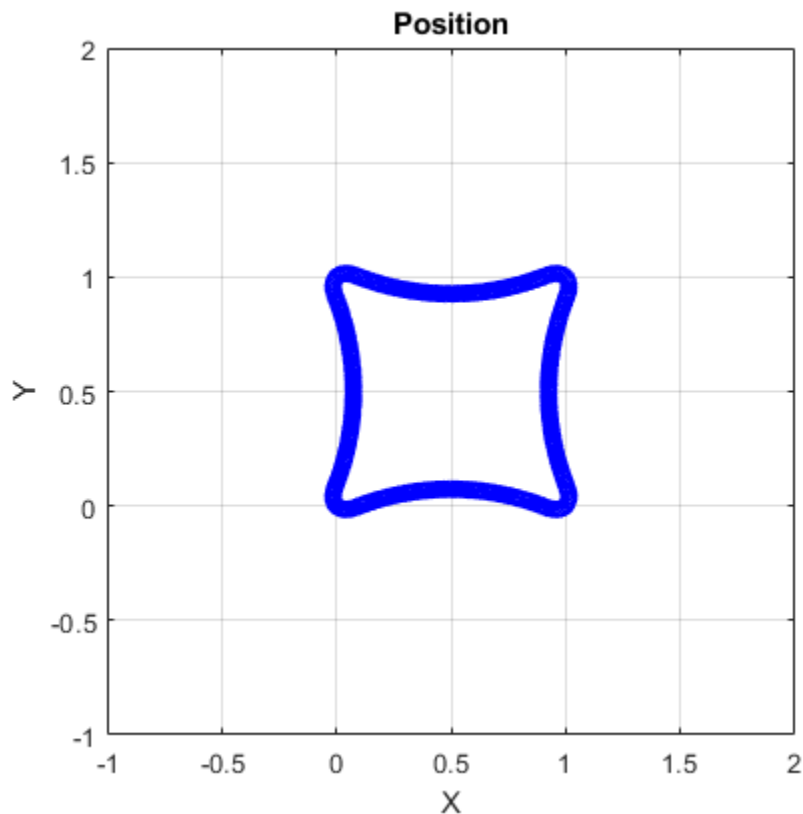
Create a figure and plot the initial position of the platform. In a loop, step through the trajectory to output the current position and current orientation. Plot the current position and log the orientation. Use `pause` to mimic real-time processing.

```
figure(1)
plot(waypoints(1,1),waypoints(1,2), 'b*')
title('Position')
axis([-1,2,-1,2])
axis square
xlabel('X')
ylabel('Y')
grid on
hold on

orientationLog = zeros(toa(end)*trajectory.SampleRate,1, 'quaternion');
count = 1;
while ~isDone(trajectory)
    [currentPosition,orientationLog(count)] = trajectory();

    plot(currentPosition(1),currentPosition(2), 'bo')

    pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
    count = count+1;
end
hold off
```

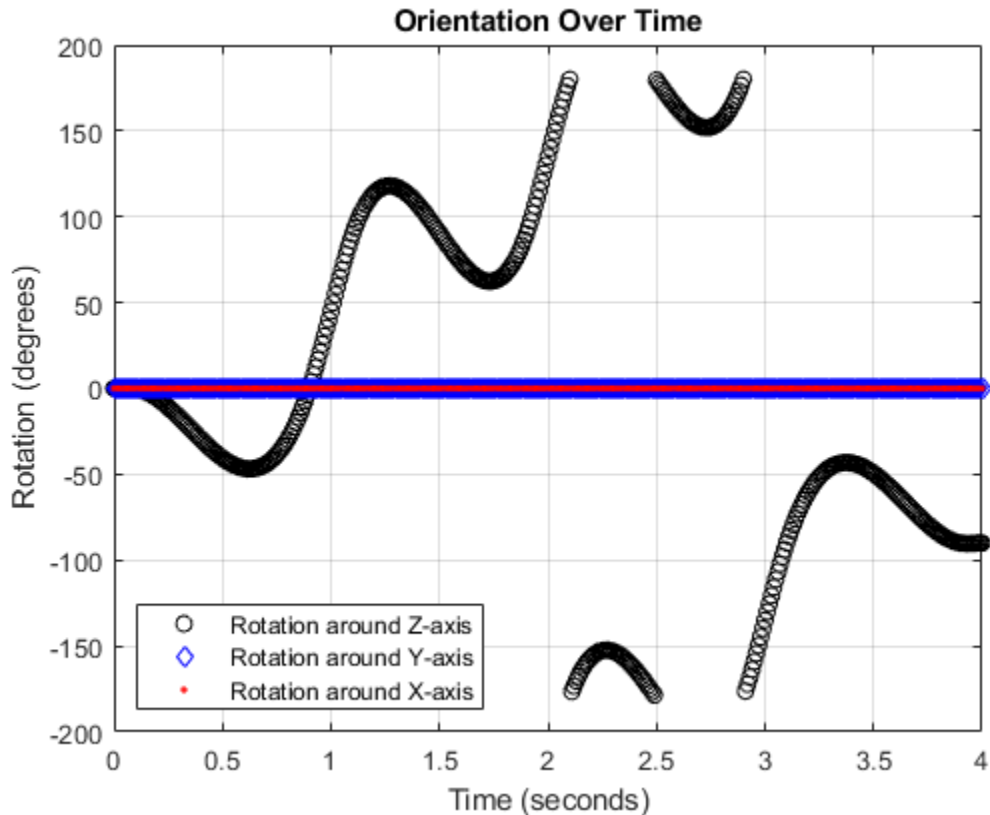


The trajectory output now appears more square-like, especially around the vertices with waypoints.

Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog], 'ZYX', 'frame');
t = 0:1/trajectory.SampleRate:4;
eulerAngles = plot(t,eulerAngles(:,1),'ko', ...
                  t,eulerAngles(:,2),'bd', ...
                  t,eulerAngles(:,3),'r.');
```

```
title('Orientation Over Time')
legend('Rotation around Z-axis', ...
       'Rotation around Y-axis', ...
       'Rotation around X-axis', ...
       'Location', 'SouthWest')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on
```



### Create Arc Trajectory

This example shows how to create an arc trajectory using the `waypointTrajectory` System object™. `waypointTrajectory` creates a path through specified waypoints that minimizes acceleration and angular velocity. After creating an arc trajectory, you restrict the trajectory to be within preset bounds.

### Create an Arc Trajectory

Define a constraints matrix consisting of waypoints, times of arrival, and orientation for an arc trajectory. The generated trajectory passes through the waypoints at the specified times with the specified orientation. The `waypointTrajectory` System object requires orientation to be specified using quaternions or rotation matrices. Convert the Euler angles saved in the constraints matrix to quaternions when specifying the `Orientation` property.

```
% Arrival, Waypoints, Orientation
constraints = [0,    20,20,0,    90,0,0;
              3,    50,20,0,    90,0,0;
              4,    58,15.5,0,  162,0,0;
              5.5,  59.5,0,0    180,0,0];

trajectory = waypointTrajectory(constraints(:,2:4), ...
    'TimeOfArrival',constraints(:,1), ...
    'Orientation',quaternion(constraints(:,5:7),'eulerd','ZYX','frame'));
```

Call `waypointInfo` on `trajectory` to return a table of your specified constraints. The creation properties `Waypoints`, `TimeOfArrival`, and `Orientation` are variables of the table. The table is convenient for indexing while plotting.

```
tInfo = waypointInfo(trajectory)
```

```
tInfo =
```

```
4x3 table
```

TimeOfArrival	Waypoints			Orientation
0	20	20	0	{1x1 quaternion}
3	50	20	0	{1x1 quaternion}
4	58	15.5	0	{1x1 quaternion}
5.5	59.5	0	0	{1x1 quaternion}

The trajectory object outputs the current position, velocity, acceleration, and angular velocity at each call. Call `trajectory` in a loop and plot the position over time. Cache the other outputs.

```
figure(1)
plot(tInfo.Waypoints(1,1),tInfo.Waypoints(1,2),'b*')
title('Position')
axis([20,65,0,25])
xlabel('North')
ylabel('East')
grid on
daspect([1 1 1])
hold on

orient = zeros(tInfo.TimeOfArrival(end)*trajectory.SampleRate,1,'quaternion');
vel = zeros(tInfo.TimeOfArrival(end)*trajectory.SampleRate,3);
acc = vel;
angVel = vel;

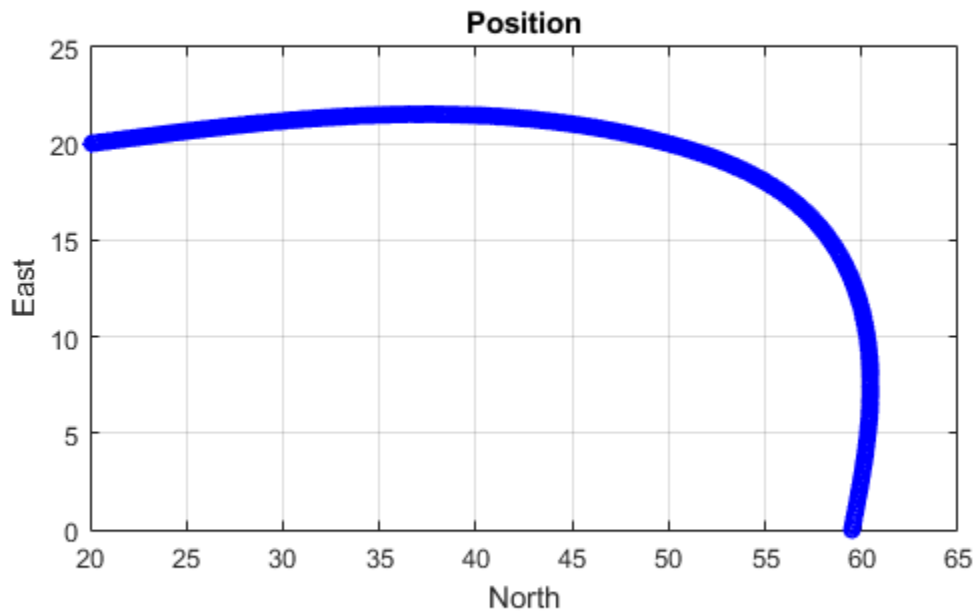
count = 1;
while ~isDone(trajectory)
```

```

[pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajectory();
plot(pos(1),pos(2),'bo')

pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
count = count + 1;
end

```



Inspect the orientation, velocity, acceleration, and angular velocity over time. The `waypointTrajectory` System object™ creates a path through the specified constraints that minimized acceleration and angular velocity.

```

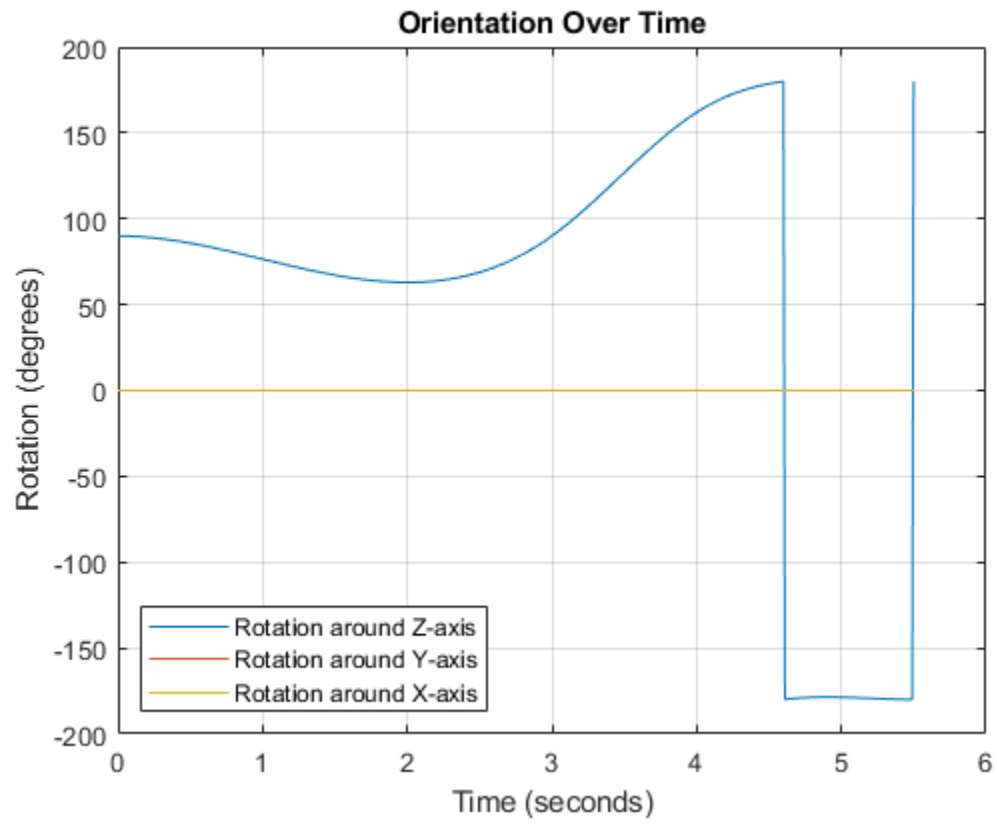
figure(2)
timeVector = 0:(1/trajectory.SampleRate):tInfo.TimeOfArrival(end);
eulerAngles = eulerd([tInfo.Orientation{1};orient],'ZYX','frame');
plot(timeVector,eulerAngles(:,1), ...
      timeVector,eulerAngles(:,2), ...
      timeVector,eulerAngles(:,3));
title('Orientation Over Time')
legend('Rotation around Z-axis', ...
       'Rotation around Y-axis', ...
       'Rotation around X-axis', ...
       'Location','southwest')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on

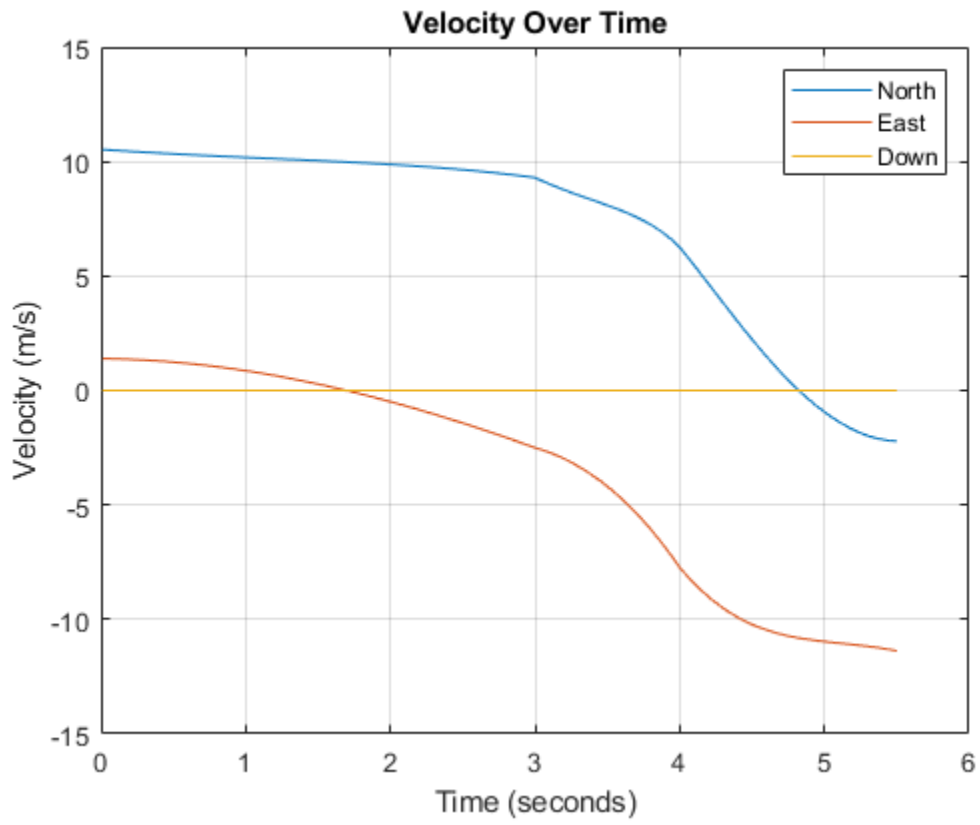
```

```
figure(3)
plot(timeVector(2:end),vel(:,1), ...
      timeVector(2:end),vel(:,2), ...
      timeVector(2:end),vel(:,3));
title('Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Velocity (m/s)')
grid on

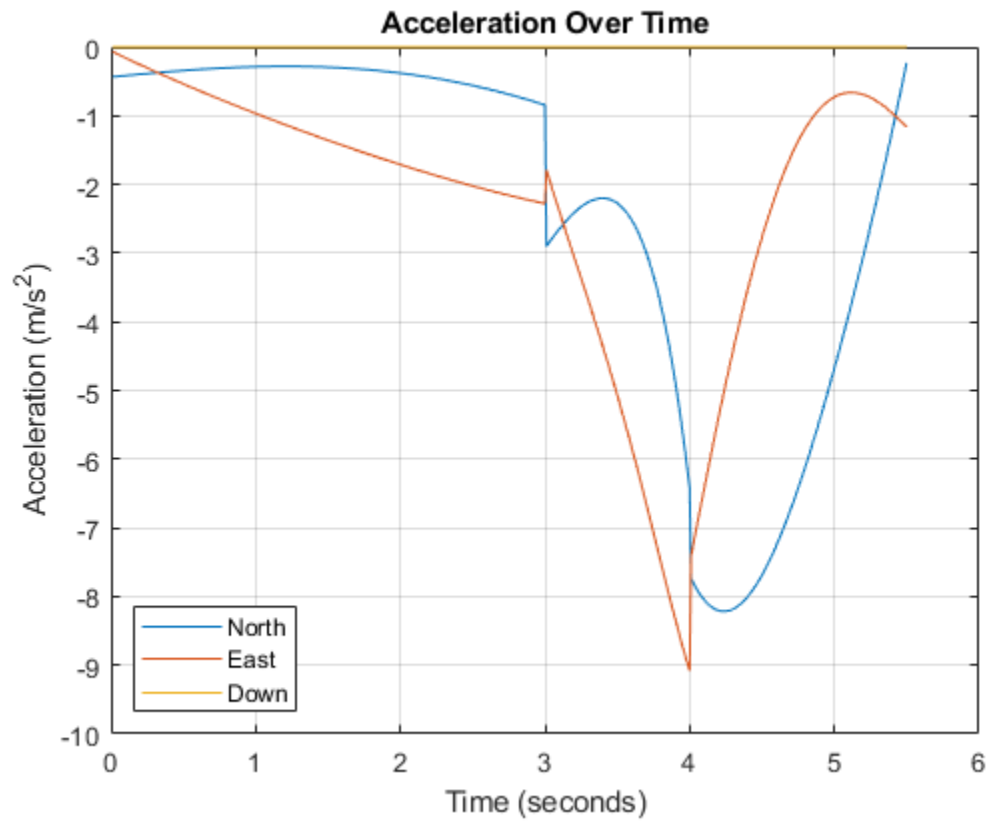
figure(4)
plot(timeVector(2:end),acc(:,1), ...
      timeVector(2:end),acc(:,2), ...
      timeVector(2:end),acc(:,3));
title('Acceleration Over Time')
legend('North','East','Down','Location','southwest')
xlabel('Time (seconds)')
ylabel('Acceleration (m/s^2)')
grid on

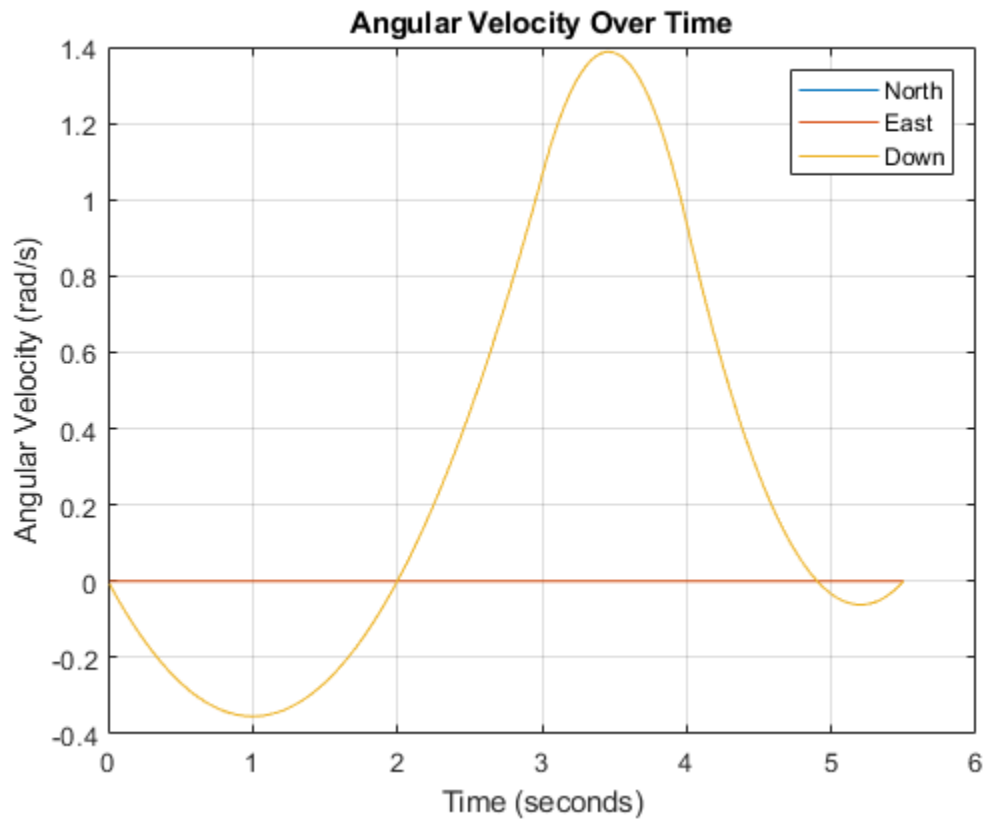
figure(5)
plot(timeVector(2:end),angVel(:,1), ...
      timeVector(2:end),angVel(:,2), ...
      timeVector(2:end),angVel(:,3));
title('Angular Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Angular Velocity (rad/s)')
grid on
```











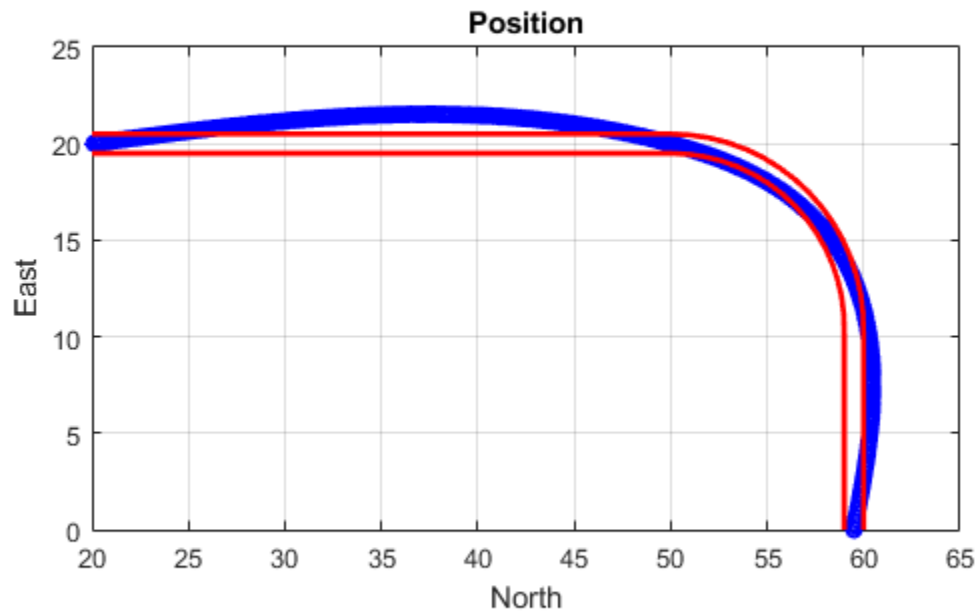
### Restrict Arc Trajectory Within Preset Bounds

You can specify additional waypoints to create trajectories within given bounds. Create upper and lower bounds for the arc trajectory.

```
figure(1)
xUpperBound = [(20:50)';50+10*sin(0:0.1:pi/2)';60*ones(11,1)];
yUpperBound = [20.5.*ones(31,1);10.5+10*cos(0:0.1:pi/2)';(10:-1:0)'];

xLowerBound = [(20:49)';50+9*sin(0:0.1:pi/2)';59*ones(11,1)];
yLowerBound = [19.5.*ones(30,1);10.5+9*cos(0:0.1:pi/2)';(10:-1:0)'];

plot(xUpperBound,yUpperBound,'r','LineWidth',2);
plot(xLowerBound,yLowerBound,'r','LineWidth',2)
```



To create a trajectory within the bounds, add additional waypoints. Create a new `waypointTrajectory` System object™, and then call it in a loop to plot the generated trajectory. Cache the orientation, velocity, acceleration, and angular velocity output from the trajectory object.

```

% Time, Waypoint, Orientation
constraints = [0, 20,20,0, 90,0,0;
              1.5, 35,20,0, 90,0,0;
              2.5, 45,20,0, 90,0,0;
              3, 50,20,0, 90,0,0;
              3.3, 53,19.5,0, 108,0,0;
              3.6, 55.5,18.25,0, 126,0,0;
              3.9, 57.5,16,0, 144,0,0;
              4.2, 59,14,0, 162,0,0;
              4.5, 59.5,10,0, 180,0,0;
              5, 59.5,5,0, 180,0,0;
              5.5, 59.5,0,0, 180,0,0];

trajectory = waypointTrajectory(constraints(:,2:4), ...
    'TimeOfArrival',constraints(:,1), ...
    'Orientation',quaternion(constraints(:,5:7),'eulerd','ZYX','frame'));
tInfo = waypointInfo(trajectory);

figure(1)
plot(tInfo.Waypoints(1,1),tInfo.Waypoints(1,2),'b*')

count = 1;

```

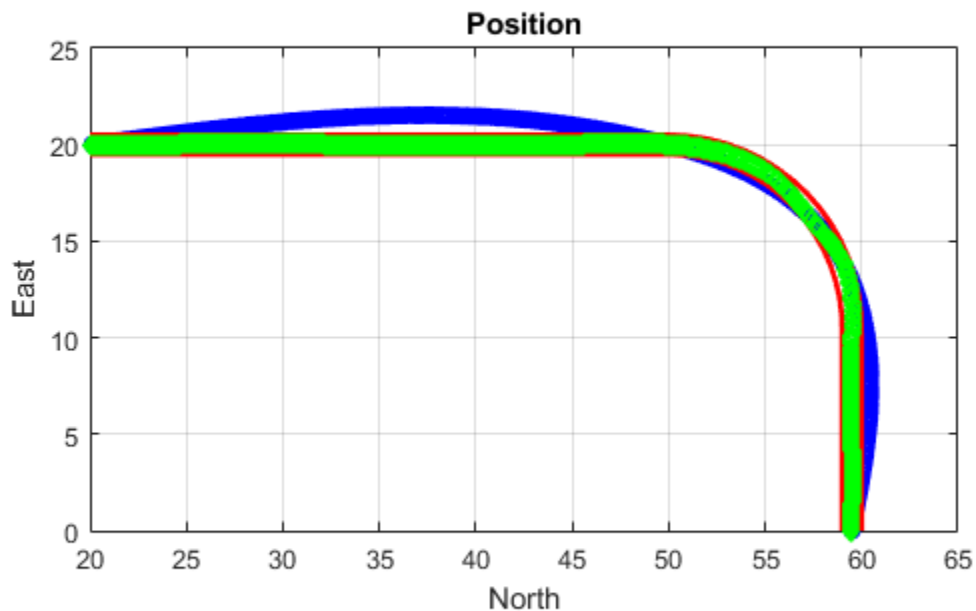
```

while ~isDone(trajjectory)
    [pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajjectory();

    plot(pos(1),pos(2),'gd')

    pause(trajjectory.SamplesPerFrame/trajjectory.SampleRate)
    count = count + 1;
end

```



The generated trajectory now fits within the specified boundaries. Visualize the orientation, velocity, acceleration, and angular velocity of the generated trajectory.

```

figure(2)
timeVector = 0:(1/trajjectory.SampleRate):tInfo.TimeOfArrival(end);
eulerAngles = eulerd(orient,'ZYX','frame');
plot(timeVector(2:end),eulerAngles(:,1), ...
      timeVector(2:end),eulerAngles(:,2), ...
      timeVector(2:end),eulerAngles(:,3));
title('Orientation Over Time')
legend('Rotation around Z-axis', ...
       'Rotation around Y-axis', ...
       'Rotation around X-axis', ...
       'Location','southwest')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on

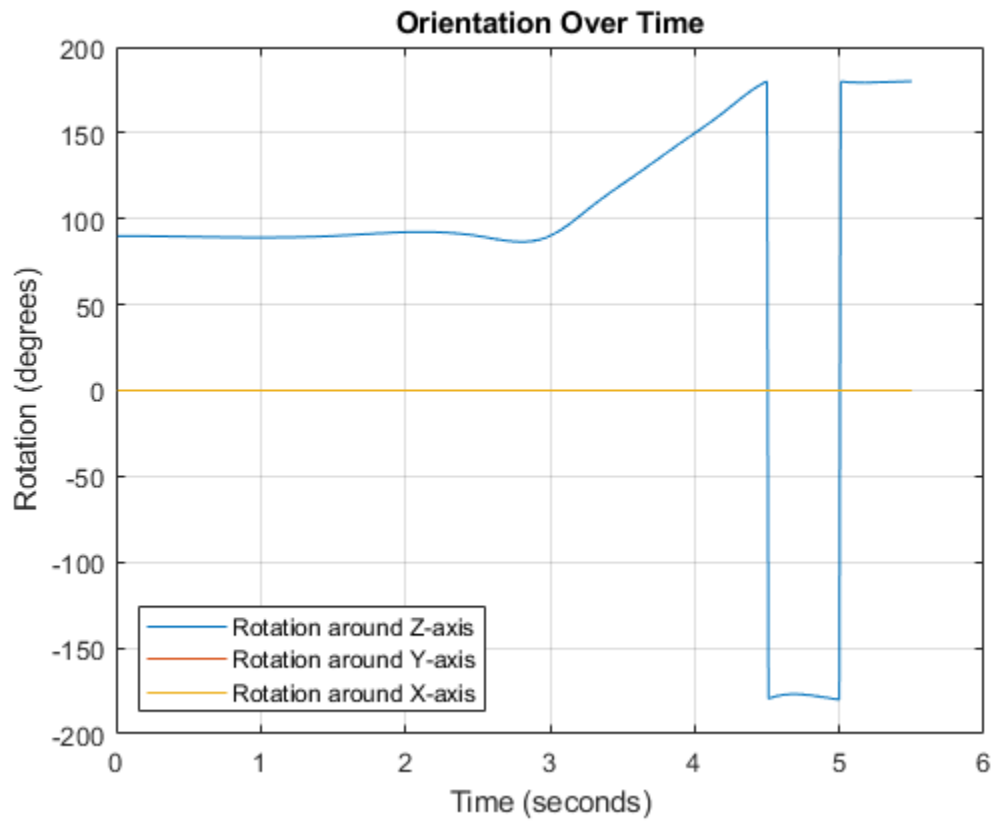
figure(3)

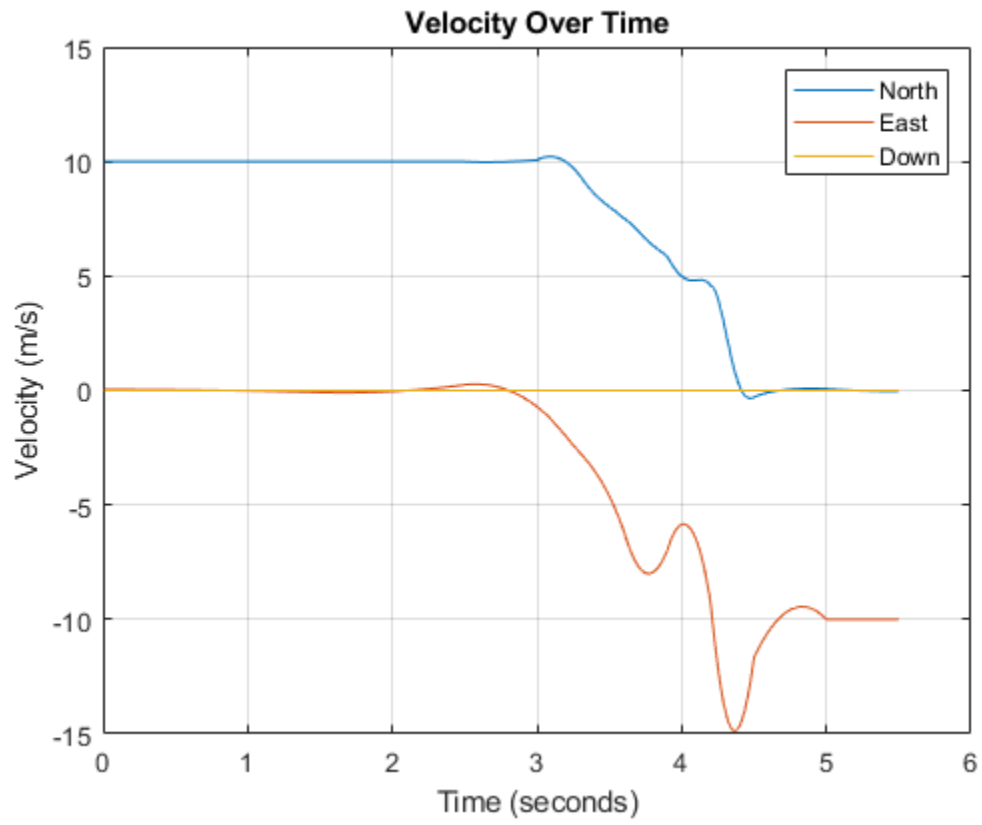
```

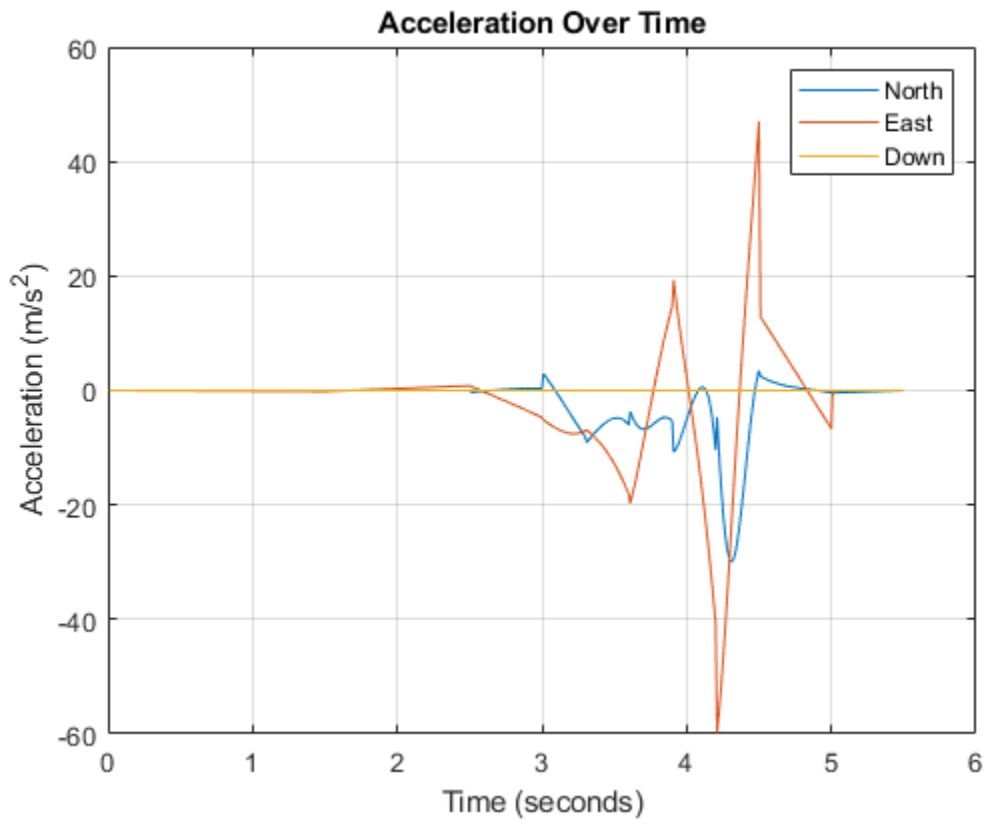
```
plot(timeVector(2:end),vel(:,1), ...
      timeVector(2:end),vel(:,2), ...
      timeVector(2:end),vel(:,3));
title('Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Velocity (m/s)')
grid on

figure(4)
plot(timeVector(2:end),acc(:,1), ...
      timeVector(2:end),acc(:,2), ...
      timeVector(2:end),acc(:,3));
title('Acceleration Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Acceleration (m/s^2)')
grid on

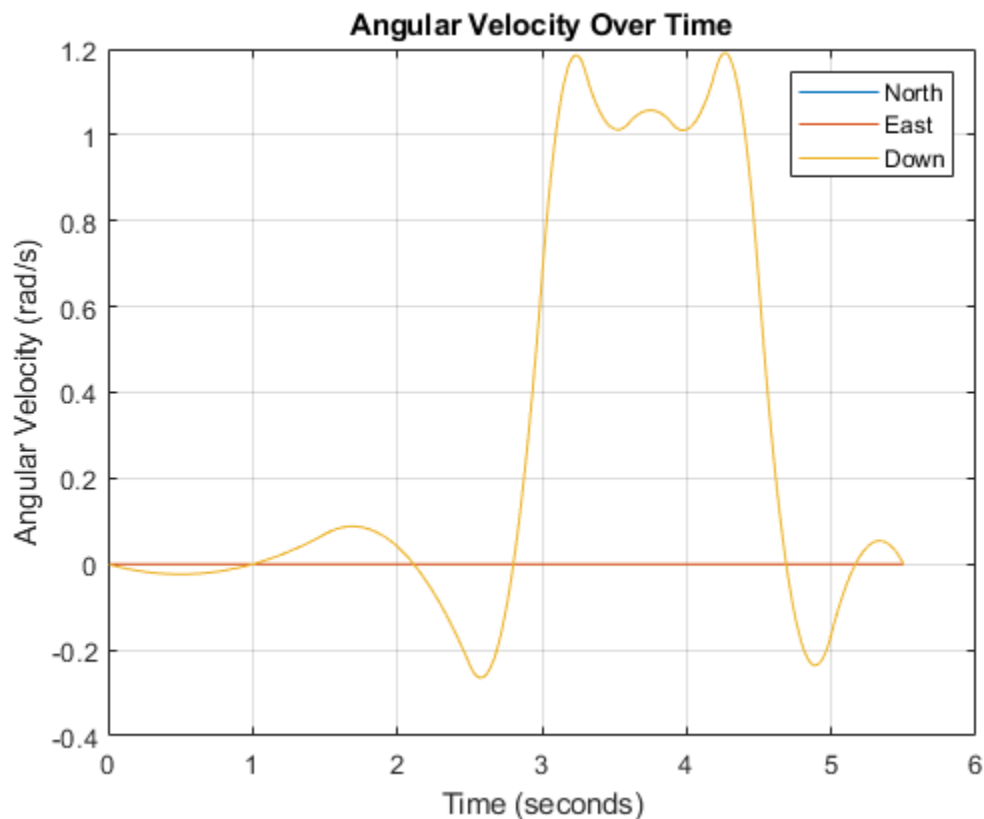
figure(5)
plot(timeVector(2:end),angVel(:,1), ...
      timeVector(2:end),angVel(:,2), ...
      timeVector(2:end),angVel(:,3));
title('Angular Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Angular Velocity (rad/s)')
grid on
```











Note that while the generated trajectory now fits within the spatial boundaries, the acceleration and angular velocity of the trajectory are somewhat erratic. This is due to over-specifying waypoints.

## Algorithms

The `waypointTrajectory` System object defines a trajectory that smoothly passes through waypoints. The trajectory connects the waypoints through an interpolation that assumes the gravity direction expressed in the trajectory reference frame is constant. Generally, you can use `waypointTrajectory` to model platform or vehicle trajectories within a hundreds of kilometers distance span.

The planar path of the trajectory (the  $x$ - $y$  plane projection) consists of piecewise, clothoid curves. The curvature of the curve between two consecutive waypoints varies linearly with the curve length between them. The tangent direction of the path at each waypoint is chosen to minimize discontinuities in the curvature, unless the course is specified explicitly via the `Course` property or implicitly via the `Velocities` property. Once the path is established, the object uses cubic Hermite interpolation to compute the location of the vehicle throughout the path as a function of time and the planar distance traveled.

The normal component ( $z$ -component) of the trajectory is subsequently chosen to satisfy a shape-preserving piecewise spline (PCHIP) unless the climb rate is specified explicitly via the `ClimbRate` property or the third column of the `Velocities` property. Choose the sign of the climb rate based on the selected `ReferenceFrame`:

- When an 'ENU' reference frame is selected, specifying a positive climb rate results in an increasing value of  $z$ .
- When an 'NED' reference frame is selected, specifying a positive climb rate results in a decreasing value of  $z$ .

You can define the orientation of the vehicle through the path in two primary ways:

- If the `Orientation` property is specified, then the object uses a piecewise-cubic, quaternion spline to compute the orientation along the path as a function of time.
- If the `Orientation` property is not specified, then the yaw of the vehicle is always aligned with the path. The roll and pitch are then governed by the `AutoBank` and `AutoPitch` property values, respectively.

<b>AutoBank</b>	<b>AutoPitch</b>	<b>Description</b>
false	false	The vehicle is always level (zero pitch and roll). This is typically used for large marine vessels.
false	true	The vehicle pitch is aligned with the path, and its roll is always zero. This is typically used for ground vehicles.
true	false	The vehicle pitch and roll are chosen so that its local $z$ -axis is aligned with the net acceleration (including gravity). This is typically used for rotary-wing craft.
true	true	The vehicle roll is chosen so that its local transverse plane aligns with the net acceleration (including gravity). The vehicle pitch is aligned with the path. This is typically used for two-wheeled vehicles and fixed-wing aircraft.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The object function, `waypointInfo`, does not support code generation.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Objects

platform | radarScenario

**Introduced in R2021a**

## perturb

Apply perturbations to object

### Syntax

```
offsets = perturb(obj)
```

### Description

`offsets = perturb(obj)` applies the perturbations defined on the object, `obj` and returns the offset values. You can define perturbations on the object by using the `perturbations` function.

### Examples

#### Perturb Waypoint Trajectory

Define a waypoint trajectory. By default, this trajectory contains two waypoints.

```
traj = waypointTrajectory
traj =
  waypointTrajectory with properties:
    SampleRate: 100
    SamplesPerFrame: 1
    Waypoints: [2x3 double]
    TimeOfArrival: [2x1 double]
    Velocities: [2x3 double]
    Course: [2x1 double]
    GroundSpeed: [2x1 double]
    ClimbRate: [2x1 double]
    Orientation: [2x1 quaternion]
    AutoPitch: 0
    AutoBank: 0
    ReferenceFrame: 'NED'
```

Define perturbations on the `Waypoints` property and the `TimeOfArrival` property.

```
rng(2020);
perturbs1 = perturbations(traj, 'Waypoints', 'Normal', 1, 1)
```

```
perturbs1=2x3 table
  Property      Type      Value
  _____  _____  _____
  "Waypoints"   "Normal"  {[ 1]}   {[ 1]}
  "TimeOfArrival" "None"    {[NaN]}  {[NaN]}
```

```
perturbs2 = perturbations(traj, 'TimeOfArrival', 'Selection', {[0;1],[0;2]})
```

```

perturbs2=2x3 table
Property          Type          Value
-----
"Waypoints"      "Normal"      {[ 1]}
"TimeOfArrival" "Selection"   {[0.5000 0.5000]}

```

Perturb the trajectory.

```
offsets = perturb(traj)
```

```
offsets=2x1 struct array with fields:
Property
Offset
PerturbedValue

```

The Waypoints property and the TimeOfArrival property have changed.

```
traj.Waypoints
```

```
ans = 2x3
```

```

1.8674    1.0203    0.7032
2.3154   -0.3207    0.0999

```

```
traj.TimeOfArrival
```

```
ans = 2x1
```

```

0
2

```

### Perturb Accuracy of insSensor

Create an insSensor object.

```
sensor = insSensor
```

```
sensor =
insSensor with properties:
```

```

MountingLocation: [0 0 0]          m
RollAccuracy: 0.2                  deg
PitchAccuracy: 0.2                 deg
YawAccuracy: 1                     deg
PositionAccuracy: [1 1 1]         m
VelocityAccuracy: 0.05             m/s
AccelerationAccuracy: 0             m/s2
AngularVelocityAccuracy: 0         deg/s
TimeInput: 0
RandomStream: 'Global stream'

```

Define the perturbation on the RollAccuracy property as three values with an equal possibility each.

```
values = {0.1 0.2 0.3}
```

```
values=1x3 cell array
    {[0.1000]}    {[0.2000]}    {[0.3000]}
```

```
probabilities = [1/3 1/3 1/3]
```

```
probabilities = 1x3
    0.3333    0.3333    0.3333
```

```
perturbations(sensor, 'RollAccuracy', 'Selection', values, probabilities)
```

```
ans=7x3 table
```

Property	Type		Value
"RollAccuracy"	"Selection"	{1x3 cell}	{[0.3333 0.3333 0.3333]}
"PitchAccuracy"	"None"	{[ NaN]}	{[ NaN]}
"YawAccuracy"	"None"	{[ NaN]}	{[ NaN]}
"PositionAccuracy"	"None"	{[ NaN]}	{[ NaN]}
"VelocityAccuracy"	"None"	{[ NaN]}	{[ NaN]}
"AccelerationAccuracy"	"None"	{[ NaN]}	{[ NaN]}
"AngularVelocityAccuracy"	"None"	{[ NaN]}	{[ NaN]}

Perturb the sensor object using the perturb function.

```
rng(2020)
perturb(sensor);
sensor
```

```
sensor =
  insSensor with properties:
    MountingLocation: [0 0 0]          m
    RollAccuracy: 0.5                 deg
    PitchAccuracy: 0.2                 deg
    YawAccuracy: 1                     deg
    PositionAccuracy: [1 1 1]         m
    VelocityAccuracy: 0.05             m/s
    AccelerationAccuracy: 0            m/s2
    AngularVelocityAccuracy: 0         deg/s
    TimeInput: 0
    RandomStream: 'Global stream'
```

The RollAccuracy is perturbed to 0.5 deg.

## Input Arguments

### **obj** — Object for perturbation

objects

Object for perturbation, specified as an object. The objects that you can perturb include:

- waypointTrajectory
- kinematicTrajectory
- geoTrajectory
- insSensor
- radarEmitter
- radarDataGenerator

## Output Arguments

### **offsets** — Property offsets

array of structure

Property offsets, returned as an array of structures. Each structure contains these fields:

Field Name	Description
Property	Name of perturbed property
Offset	Offset values applied in the perturbation
PerturbedValue	Property values after the perturbation

## See Also

perturbations

**Introduced in R2021a**

## perturbations

Perturbation defined on object

### Syntax

```
perturbs = perturbations(obj)
perturbs = perturbations(obj,property)
perturbs = perturbations(obj,property,'None')
perturbs = perturbations(obj,property,'Selection',values,probabilities)
perturbs = perturbations(obj,property,'Normal',mean,deviation)
perturbs = perturbations(obj,property,'TruncatedNormal',mean,deviation,
lowerLimit,upperLimit)
perturbs = perturbations(obj,property,'Uniform',minVal,maxVal)
perturbs = perturbations(obj,property,'Custom',perturbFcn)
```

### Description

`perturbs = perturbations(obj)` returns the list of property perturbations, `perturbs`, defined on the object, `obj`. The returned `perturbs` lists all the perturbable properties. If any property is not perturbed, then its corresponding `Type` is returned as "Null" and its corresponding `Value` is returned as `{Null,Null}`.

`perturbs = perturbations(obj,property)` returns the current perturbation applied to the specified property.

`perturbs = perturbations(obj,property,'None')` defines a property that must not be perturbed.

`perturbs = perturbations(obj,property,'Selection',values,probabilities)` defines the property perturbation offset drawn from a set of values that have corresponding probabilities.

`perturbs = perturbations(obj,property,'Normal',mean,deviation)` defines the property perturbation offset drawn from a normal distribution with specified mean and standard deviation.

`perturbs = perturbations(obj,property,'TruncatedNormal',mean,deviation,lowerLimit,upperLimit)` defines the property perturbation offset drawn from a normal distribution with specified mean, standard deviation, lower limit, and upper limit.

`perturbs = perturbations(obj,property,'Uniform',minVal,maxVal)` defines the property perturbation offset drawn from a uniform distribution on an interval `[minVal, maxVal]`.

`perturbs = perturbations(obj,property,'Custom',perturbFcn)` enables you to define a custom function, `perturbFcn`, that draws the perturbation offset value.

### Examples



### Default Perturbation Properties of waypointTrajectory

Create a waypointTrajectory object.

```
traj = waypointTrajectory;
```

Show the default perturbation properties using the perturbations method.

```
perturbs = perturbations(traj)
```

```
perturbs=2×3 table
  Property      Type      Value
  _____  _____  _____
  "Waypoints"   "None"    {[NaN]}  {[NaN]}
  "TimeOfArrival" "None"    {[NaN]}  {[NaN]}
```

### Perturb Accuracy of insSensor

Create an insSensor object.

```
sensor = insSensor
```

```
sensor =
  insSensor with properties:
      MountingLocation: [0 0 0]          m
      RollAccuracy: 0.2                 deg
      PitchAccuracy: 0.2                deg
      YawAccuracy: 1                   deg
      PositionAccuracy: [1 1 1]         m
      VelocityAccuracy: 0.05            m/s
      AccelerationAccuracy: 0            m/s2
      AngularVelocityAccuracy: 0         deg/s
      TimeInput: 0
      RandomStream: 'Global stream'
```

Define the perturbation on the RollAccuracy property as three values with an equal possibility each.

```
values = {0.1 0.2 0.3}
```

```
values=1×3 cell array
  {[0.1000]}  {[0.2000]}  {[0.3000]}
```

```
probabilities = [1/3 1/3 1/3]
```

```
probabilities = 1×3
  0.3333  0.3333  0.3333
```

```
perturbations(sensor, 'RollAccuracy', 'Selection', values, probabilities)
```

```
ans=7x3 table
      Property      Type      Value
-----
"RollAccuracy"     "Selection" {1x3 cell} {[0.3333 0.3333 0.3333]}
"PitchAccuracy"    "None"      {[ NaN]}    {[ NaN]}
"YawAccuracy"      "None"      {[ NaN]}    {[ NaN]}
"PositionAccuracy" "None"      {[ NaN]}    {[ NaN]}
"VelocityAccuracy" "None"      {[ NaN]}    {[ NaN]}
"AccelerationAccuracy" "None"    {[ NaN]}    {[ NaN]}
"AngularVelocityAccuracy" "None"    {[ NaN]}    {[ NaN]}
```

Perturb the sensor object using the perturb function.

```
rng(2020)
perturb(sensor);
sensor

sensor =
  insSensor with properties:
    MountingLocation: [0 0 0]      m
    RollAccuracy: 0.5             deg
    PitchAccuracy: 0.2            deg
    YawAccuracy: 1                deg
    PositionAccuracy: [1 1 1]     m
    VelocityAccuracy: 0.05        m/s
    AccelerationAccuracy: 0        m/s2
    AngularVelocityAccuracy: 0     deg/s
    TimeInput: 0
    RandomStream: 'Global stream'
```

The RollAccuracy is perturbed to 0.5 deg.

### Perturb Waypoint Trajectory

Define a waypoint trajectory. By default, this trajectory contains two waypoints.

```
traj = waypointTrajectory

traj =
  waypointTrajectory with properties:
    SampleRate: 100
    SamplesPerFrame: 1
    Waypoints: [2x3 double]
    TimeOfArrival: [2x1 double]
    Velocities: [2x3 double]
    Course: [2x1 double]
    GroundSpeed: [2x1 double]
    ClimbRate: [2x1 double]
    Orientation: [2x1 quaternion]
    AutoPitch: 0
    AutoBank: 0
```

```
ReferenceFrame: 'NED'
```

Define perturbations on the `Waypoints` property and the `TimeOfArrival` property.

```
rng(2020);
perturbs1 = perturbations(traj, 'Waypoints', 'Normal', 1, 1)
```

```
perturbs1=2x3 table
  Property      Type      Value
  _____  _____  _____
  "Waypoints"   "Normal"   {[ 1]}   {[ 1]}
  "TimeOfArrival" "None"     {[NaN]}  {[NaN]}
```

```
perturbs2 = perturbations(traj, 'TimeOfArrival', 'Selection', {[0;1],[0;2]})
```

```
perturbs2=2x3 table
  Property      Type      Value
  _____  _____  _____
  "Waypoints"   "Normal"   {[ 1]}   {[ 1]}
  "TimeOfArrival" "Selection" {1x2 cell} {[0.5000 0.5000]}
```

Perturb the trajectory.

```
offsets = perturb(traj)
```

```
offsets=2x1 struct array with fields:
  Property
  Offset
  PerturbedValue
```

The `Waypoints` property and the `TimeOfArrival` property have changed.

```
traj.Waypoints
```

```
ans = 2x3
    1.8674    1.0203    0.7032
    2.3154   -0.3207    0.0999
```

```
traj.TimeOfArrival
```

```
ans = 2x1
     0
     2
```

## Input Arguments

**obj** — Object to be perturbed  
objects

Object to be perturbed, specified as an object. The objects that you can perturb include:

- `waypointTrajectory`
- `kinematicTrajectory`
- `geoTrajectory`
- `insSensor`
- `radarEmitter`
- `radarDataGenerator`

**property – Perturbable property**

property name

Perturbable property, specified as a property name. Use `perturbations` to obtain a full list of perturbable properties for the specified `obj`.

**values – Perturbation offset values**

*n*-element cell array of property values

Perturbation offset values, specified as an *n*-element cell array of property values. The function randomly draws the perturbation value for the property from the cell array based on the values' corresponding probabilities specified in the `probabilities` input.

**probabilities – Drawing probabilities for each perturbation value**

*n*-element array of nonnegative scalar

Drawing probabilities for each perturbation value, specified as an *n*-element array of nonnegative scalars, where *n* is the number of perturbation values provided in the `values` input. The sum of all elements must be equal to one.

For example, you can specify a series of perturbation value-probability pair as  $\{x_1, x_2, \dots, x_n\}$  and  $\{p_1, p_2, \dots, p_n\}$ , where the probability of drawing  $x_i$  is  $p_i$  ( $i = 1, 2, \dots, n$ ).

**mean – Mean of normal or truncated normal distribution**

scalar | vector | matrix

Mean of normal or truncated normal distribution, specified as a scalar, vector, or matrix. The dimension of `mean` must be compatible with the corresponding property that you perturb.

**deviation – Standard deviation of normal or truncated normal distribution**

nonnegative scalar | vector of nonnegative scalar | matrix of nonnegative scalar

Standard deviation of normal or truncated normal distribution, specified as a nonnegative scalar, vector of nonnegative scalars, or matrix of nonnegative scalars. The dimension of `deviation` must be compatible with the corresponding property that you perturb.

**lowerLimit – Lower limit of truncated normal distribution**

scalar | vector | matrix

Lower limit of the truncated normal distribution, specified as a scalar, vector, or matrix. The dimension of `lowerLimit` must be compatible with the corresponding property that you perturb.

**upperLimit – Upper limit of truncated normal distribution**

scalar | vector | matrix

Upper limit of the truncated normal distribution, specified as a scalar, vector, or matrix. The dimension of `upperLimit` must be compatible with the corresponding property that you perturb.

### **minVal — Minimum value of uniform distribution interval**

scalar | vector | matrix

Minimum value of the uniform distribution interval, specified as a scalar, vector, or matrix. The dimension of `minVal` must be compatible with the corresponding property that you perturb.

### **maxVal — Maximum value of uniform distribution interval**

scalar | vector | matrix

Maximum value of the uniform distribution interval, specified as a scalar, vector, or matrix. The dimension of `maxVal` must be compatible with the corresponding property that you perturb.

### **perturbFcn — Perturbation function**

function handle

Perturbation function, specified as a function handle. The function must have this syntax:

```
offset = myfun(propVal)
```

where `propVal` is the value of the property and `offset` is the perturbation offset for the property.

## **Output Arguments**

### **perturbs — Perturbations defined on object**

table of perturbation property

Perturbations defined on the object, returned as a table of perturbation properties. The table has three columns:

- **Property** — Property names.
- **Type** — Type of perturbations, returned as "None", "Selection", "Normal", "TruncatedNormal", "Uniform", or "Custom".
- **Value** — Perturbation values, returned as a cell array.

## **More About**

### **Specify Perturbation Distributions**

You can specify the distribution for the perturbation applied to a specific property.

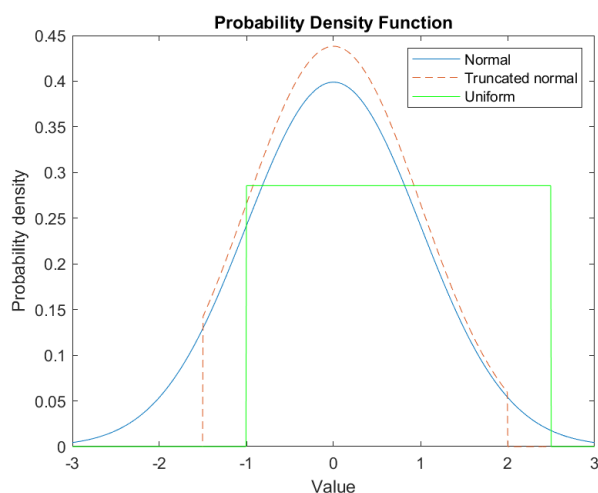
- **Selection distribution** — The function defines the perturbation offset as one of the specified values with the associated probability. For example, if you specify the values as `[1 2]` and specify the probabilities as `[0.7 0.3]`, then the `perturb` function adds an offset value of 1 to the property with a probability of 0.7 and add an offset value of 2 to the property with a probability of 0.3. Use selection distribution when you only want to perturb the property with a number of discrete values.
- **Normal distribution** — The function defines the perturbation offset as a value drawn from a normal distribution with the specified mean and standard deviation (or covariance). Normal distribution is the most commonly used distribution since it mimics the natural perturbation of parameters in most cases.

- Truncated normal distribution — The function defines the perturbation offset as a value drawn from a truncated normal distribution with the specified mean, standard deviation (or covariance), lower limit, and upper limit. Different from the normal distribution, the values drawn from a truncated normal distribution are truncated by the lower and upper limit. Use truncated normal distribution when you want to apply a normal distribution, but the valid values of the property are confined in an interval.
- Uniform distribution — The function defines the perturbation offset as a value drawn from a uniform distribution with the specified minimum and maximum values. All the values in the interval (specified by the minimum and maximum values) have the same probability of realization.
- Custom distribution — Customize your own perturbation function. The function must have this syntax:

```
offset = myfun(propVal)
```

where `propVal` is the value of the property and `offset` is the perturbation offset for the property.

This figure shows probability density functions for a normal distribution, a truncated normal distribution, and a uniform distribution, respectively.



## See Also

`perturb`

Introduced in R2021a

# waypointInfo

Get waypoint information table

## Syntax

```
trajectoryInfo = waypointInfo(trajectory)
```

## Description

`trajectoryInfo = waypointInfo(trajectory)` returns a table of waypoints, times of arrival, velocities, and orientation for the trajectory System object.

## Input Arguments

### **trajectory** — Object of waypointTrajectory

object

Object of the waypointTrajectory System object.

## Output Arguments

### **trajectoryInfo** — Trajectory information

table

Trajectory information, returned as a table with variables corresponding to set creation properties: Waypoints, TimeOfArrival, Velocities, and Orientation.

The trajectory information table always has variables `Waypoints` and `TimeOfArrival`. If the `Velocities` property is set during construction, the trajectory information table additionally returns velocities. If the `Orientation` property is set during construction, the trajectory information table additionally returns orientation.

## See Also

### **Objects**

waypointTrajectory

### **Functions**

lookupPose | perturbations | perturb

**Introduced in R2021a**

# lookupPose

Obtain pose information for certain time

## Syntax

```
[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(traj,sampleTimes)
```

## Description

[position,orientation,velocity,acceleration,angularVelocity] = lookupPose(traj,sampleTimes) returns the pose information of the waypoint trajectory at the specified sample times. If any sample time is beyond the duration of the trajectory, the corresponding pose information is returned as NaN.

## Input Arguments

### **traj** — Waypoint trajectory

waypointTrajectory object

Waypoint trajectory, specified as a waypointTrajectory object.

### **sampleTimes** — Sample times

$M$ -element vector of nonnegative scalar

Sample times in seconds, specified as an  $M$ -element vector of nonnegative scalars.

## Output Arguments

### **position** — Position in local navigation coordinate system (m)

$M$ -by-3 matrix

Position in the local navigation coordinate system in meters, returned as an  $M$ -by-3 matrix.

$M$  is specified by the sampleTimes input.

Data Types: double

### **orientation** — Orientation in local navigation coordinate system

$M$ -element quaternion column vector | 3-by-3-by- $M$  real array

Orientation in the local navigation coordinate system, returned as an  $M$ -by-1 quaternion column vector or a 3-by-3-by- $M$  real array.

Each quaternion or 3-by-3 rotation matrix is a frame rotation from the local navigation coordinate system to the current body coordinate system.

$M$  is specified by the sampleTimes input.

Data Types: double



**velocity — Velocity in local navigation coordinate system (m/s)***M*-by-3 matrix

Velocity in the local navigation coordinate system in meters per second, returned as an *M*-by-3 matrix.

*M* is specified by the `sampleTimes` input.

Data Types: `double`

**acceleration — Acceleration in local navigation coordinate system (m/s<sup>2</sup>)***M*-by-3 matrix

Acceleration in the local navigation coordinate system in meters per second squared, returned as an *M*-by-3 matrix.

*M* is specified by the `sampleTimes` input.

Data Types: `double`

**angularVelocity — Angular velocity in local navigation coordinate system (rad/s)***M*-by-3 matrix

Angular velocity in the local navigation coordinate system in radians per second, returned as an *M*-by-3 matrix.

*M* is specified by the `sampleTimes` input.

Data Types: `double`

**See Also****Objects**`waypointTrajectory`**Functions**`waypointInfo` | `perturbations` | `perturb`**Introduced in R2021a**

## radarEmitter

Radar signals and interferences generator

### Description

The `radarEmitter` System object creates an emitter to simulate radar emissions. You can use the `radarEmitter` object in a scenario that detects and tracks moving and stationary platforms. Construct a scenario using `radarScenario`.

A radar emitter changes the look angle between updates by stepping the mechanical and electronic position of the beam in increments of the angular span specified in the `FieldOfView` property. The radar scans the total region in azimuth and elevation defined by the radar mechanical and electronic scan limits, `MechanicalScanLimits` and `ElectronicScanLimits`, respectively. If the scan limits for azimuth or elevation are set to `[0 0]`, then no scanning is performed along that dimension for that scan mode. If the maximum mechanical scan rate for azimuth or elevation is set to zero, then no mechanical scanning is performed along that dimension.

To generate radar detections:

- 1 Create the `radarEmitter` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

### Creation

#### Syntax

```
emitter = radarEmitter(EmitterIndex)

emitter = radarEmitter(EmitterIndex,'No scanning')
emitter = radarEmitter(EmitterIndex,'Raster')
emitter = radarEmitter(EmitterIndex,'Rotator')
emitter = radarEmitter(EmitterIndex,'Sector')

emitter = radarEmitter( ___,Name,Value)
```

#### Description

`emitter = radarEmitter(EmitterIndex)` creates a radar emitter object with default property values.

`emitter = radarEmitter(EmitterIndex,'No scanning')` is a convenience syntax that creates a `radarEmitter` that stares along the radar antenna boresight direction. No mechanical or electronic scanning is performed. This syntax sets the `ScanMode` property to `'No scanning'`.

`emitter = radarEmitter(EmitterIndex,'Raster')` is a convenience syntax that creates a `radarEmitter` object that mechanically scans a raster pattern. The raster span is 90° in azimuth

from  $-45^\circ$  to  $+45^\circ$  and in elevation from the horizon to  $10^\circ$  above the horizon. See “Convenience Syntaxes” on page 4-606 for the properties set by this syntax.

`emitter = radarEmitter(EmitterIndex, 'Rotator')` is a convenience syntax that creates a `radarEmitter` object that mechanically scans  $360^\circ$  in azimuth by mechanically rotating the antenna at a constant rate. When you set `HasElevation` to `true`, the radar antenna mechanically points towards the center of the elevation field of view. See “Convenience Syntaxes” on page 4-606 for the properties set by this syntax.

`emitter = radarEmitter(EmitterIndex, 'Sector')` is a convenience syntax to create a `radarEmitter` object that mechanically scans a  $90^\circ$  azimuth sector from  $-45^\circ$  to  $+45^\circ$ . Setting `HasElevation` to `true`, points the radar antenna towards the center of the elevation field of view. You can change the `ScanMode` to `'Electronic'` to electronically scan the same azimuth sector. In this case, the antenna is not mechanically tilted in an electronic sector scan. Instead, beams are stacked electronically to process the entire elevation spanned by the scan limits in a single dwell. See “Convenience Syntaxes” on page 4-606 for the properties set by this syntax.

`emitter = radarEmitter( ____, Name, Value)` sets properties using one or more name-value pairs after all other input arguments. Enclose each property name in quotes. For example, `radarEmitter('CenterFrequency', 2e6)` creates a radar emitter creates detections in the emitter Cartesian coordinate system and has a maximum detection range of 200 meters. If you specify the emitter index using the `EmitterIndex` property, you can omit the `EmitterIndex` input.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

### EmitterIndex — Unique sensor identifier

positive integer

Unique emitter identifier, specified as a positive integer. When creating a `radarEmitter` system object, you must either specify the `EmitterIndex` as the first input argument in the creation syntax, or specify it as the value for the `EmitterIndex` property in the creation syntax.

Example: 2

Data Types: `double`

### UpdateRate — Emitter update rate

1 (default) | positive scalar

Emitter update rate, specified as a positive scalar. The emitter generates new emissions at intervals defined by the reciprocal of the `UpdateRate` property. This interval must be an integer multiple of the simulation time interval defined in `radarScenario`. Any update requested from the emitter between update intervals contains no emissions. Units are in hertz.

Example: 5

Data Types: `double`

**MountingLocation — Emitter location on platform**`[0 0 0]` (default) | 1-by-3 real-valued vector

Emitter location on platform, specified as a 1-by-3 real-valued vector. This property defines the coordinates of the emitter with respect to the platform origin. The default value specifies that the emitter origin is at the origin of its platform. Units are in meters.

Example: `[.2 0.1 0]`

Data Types: `double`

**MountingAngles — Orientation of emitter**`[0 0 0]` (default) | 3-element real-valued vector

Orientation of the emitter with respect to the platform, specified as a three-element real-valued vector. Each element of the vector corresponds to an intrinsic Euler angle rotation that carries the body axes of the platform to the emitter axes. The three elements define the rotations around the *z*, *y*, and *x* axes respectively, in that order. The first rotation rotates the platform axes around the *z*-axis. The second rotation rotates the carried frame around the rotated *y*-axis. The final rotation rotates carried frame around the carried *x*-axis. Units are in degrees.

Example: `[10 20 -15]`

Data Types: `double`

**FieldOfView — Fields of view of sensor**`[10;50]` | 2-by-1 vector of positive scalar

Fields of view of sensor, specified as a 2-by-1 vector of positive scalars in degree, `[azfov;elfov]`. The field of view defines the total angular extent spanned by the sensor. The azimuth filed of view `azfov` must lie in the interval (0,360]. The elevation filed of view `elfov` must lie in the interval (0,180].

Example: `[14;7]`

Data Types: `double`

**ScanMode — Scanning mode of radar**`'Mechanical'` (default) | `'Electronic'` | `'Mechanical and electronic'` | `'No scanning'`

Scanning mode of radar, specified as `'Mechanical'`, `'Electronic'`, `'Mechanical and electronic'`, or `'No scanning'`.

## Scan Modes

ScanMode	Purpose
'Mechanical'	The radar scans mechanically across the azimuth and elevation limits specified by the <code>MechanicalScanLimits</code> property. The scan direction increments by the radar field of view angle between dwells.
'Electronic'	The radar scans electronically across the azimuth and elevation limits specified by the <code>ElectronicScanLimits</code> property. The scan direction increments by the radar field of view angle between dwells.
'Mechanical and electronic'	The radar mechanically scans the antenna boresight across the mechanical scan limits and electronically scans beams relative to the antenna boresight across the electronic scan limits. The total field of regard scanned in this mode is the combination of the mechanical and electronic scan limits. The scan direction increments by the radar field of view angle between dwells.
'No scanning'	The radar beam points along the antenna boresight defined by the <code>mountingAngles</code> property.

Example: 'No scanning'

Data Types: char

### MaxMechanicalScanRate — Maximum mechanical scan rate

[75;75] (default) | nonnegative scalar | real-valued 2-by-1 vector with nonnegative entries

Maximum mechanical scan rate, specified as a nonnegative scalar or real-valued 2-by-1 vector with nonnegative entries.

When `HasElevation` is `true`, specify the scan rate as a 2-by-1 column vector of nonnegative entries, `[maxAzRate; maxElRate]`. `maxAzRate` is the maximum scan rate in azimuth and `maxElRate` is the maximum scan rate in elevation.

When `HasElevation` is `false`, specify the scan rate as a nonnegative scalar representing the maximum mechanical azimuth scan rate.

Scan rates set the maximum rate at which the radar can mechanically scan. The radar sets its scan rate to step the radar mechanical angle by the field of regard. If the required scan rate exceeds the maximum scan rate, the maximum scan rate is used. Units are degrees per second.

Example: [5,10]

### Dependencies

To enable this property, set the `ScanMode` property to 'Mechanical' or 'Mechanical and electronic'.

Data Types: double

### **MechanicalScanLimits — Angular limits of mechanical scan directions of radar**

[0 360; -10 0] (default) | real-valued 1-by-2 row vector | real-valued 2-by-2 matrix

Angular limits of mechanical scan directions of radar, specified as a real-valued 1-by-2 row vector or a real-valued 2-by-2 matrix. The mechanical scan limits define the minimum and maximum mechanical angles the radar can scan from its mounted orientation.

When `HasElevation` is true, the scan limits take the form [minAz maxAz; minEl maxEl]. `minAz` and `maxAz` represent the minimum and maximum limits of the azimuth angle scan. `minEl` and `maxEl` represent the minimum and maximum limits of the elevation angle scan. When `HasElevation` is false, the scan limits take the form [minAz maxAz]. If you specify the scan limits as a 2-by-2 matrix but set `HasElevation` to false, the second row of the matrix is ignored.

Azimuthal scan limits cannot span more than 360° and elevation scan limits must lie within the closed interval [-90° 90°]. Units are in degrees.

Example: [-90 90;0 85]

#### **Dependencies**

To enable this property, set the `ScanMode` property to 'Mechanical' or 'Mechanical and electronic'.

Data Types: double

### **MechanicalAngle — Current mechanical scan angle**

scalar | real-valued 2-by-1 vector

This property is read-only.

Current mechanical scan angle of radar, returned as a scalar or real-valued 2-by-1 vector. When `HasElevation` is true, the scan angle takes the form [Az;El]. `Az` and `El` represent the azimuth and elevation scan angles, respectively, relative to the mounted angle of the radar on the platform. When `HasElevation` is false, the scan angle is a scalar representing the azimuth scan angle.

#### **Dependencies**

To enable this property, set the `ScanMode` property to 'Mechanical' or 'Mechanical and electronic'.

Data Types: double

### **ElectronicScanLimits — Angular limits of electronic scan directions of radar**

[-45 45; -45 45] (default) | real-valued 1-by-2 row vector | real-valued 2-by-2 matrix

Angular limits of electronic scan directions of radar, specified as a real-valued 1-by-2 row vector or a real-valued 2-by-2 matrix. The electronic scan limits define the minimum and maximum electronic angles the radar can scan from its current mechanical direction.

When `HasElevation` is true, the scan limits take the form [minAz maxAz; minEl maxEl]. `minAz` and `maxAz` represent the minimum and maximum limits of the azimuth angle scan. `minEl` and `maxEl` represent the minimum and maximum limits of the elevation angle scan. When `HasElevation` is false, the scan limits take the form [minAz maxAz]. If you specify the scan limits as a 2-by-2 matrix but set `HasElevation` to false, the second row of the matrix is ignored.

Azimuthal scan limits and elevation scan limits must lie within the closed interval  $[-90^\circ \ 90^\circ]$ . Units are in degrees.

Example: `[-90 90; 0 85]`

#### Dependencies

To enable this property, set the `ScanMode` property to `'Electronic'` or `'Mechanical and electronic'`.

Data Types: `double`

#### ElectronicAngle — Current electronic scan angle

`electronic scalar` | `nonnegative scalar`

This property is read-only.

Current electronic scan angle of radar, returned as a scalar or 1-by-2 column vector. When `HasElevation` is `true`, the scan angle takes the form `[Az; El]`. `Az` and `El` represent the azimuth and elevation scan angles, respectively. When `HasElevation` is `false`, the scan angle is a scalar representing the azimuth scan angle.

#### Dependencies

To enable this property, set the `ScanMode` property to `'Electronic'` or `'Mechanical and electronic'`.

Data Types: `double`

#### LookAngle — Look angle of emitter

`scalar` | `real-valued 2-by-1 vector`

This property is read-only.

Look angle of emitter, specified as a scalar or real-valued 2-by-1 vector. Look angle is a combination of the mechanical angle and electronic angle depending on the `ScanMode` property. When `HasElevation` is `true`, the look angle takes the form `[Az; El]`. `Az` and `El` represent the azimuth and elevation look angles, respectively. When `HasElevation` is `false`, the look angle is a scalar representing the azimuth look angle.

ScanMode	LookAngle
<code>'Mechanical'</code>	<code>MechanicalAngle</code>
<code>'Electronic'</code>	<code>ElectronicAngle</code>
<code>'Mechanical and Electronic'</code>	<code>MechanicalAngle + ElectronicAngle</code>
<code>'No scanning'</code>	<code>0</code>

Data Types: `double`

#### HasElevation — Enable radar elevation scan and measurements

`false` (default) | `true`

Enable the radar to measure target elevation angles and to scan in elevation, specified as `false` or `true`. Set this property to `true` to model a radar emitter that can estimate target elevation and scan in elevation.

Data Types: `logical`

**EIRP — Effective isotropic radiated power**

100 (default) | scalar

Effective isotropic radiated power of the transmitter, specified as a scalar. EIRP is the root mean squared power input to a lossless isotropic antenna that gives the same power density in the far field as the actual transmitter. EIRP is equal to the power input to the transmitter antenna (in dBW) plus the transmitter isotropic antenna gain. Units are in dBi.

Data Types: double

**CenterFrequency — Center frequency of radar band**

positive scalar

Center frequency of radar band, specified as a positive scalar. Units are in hertz.

Example: 100e6

Data Types: double

**Bandwidth — Radar waveform bandwidth**

positive scalar

Radar waveform bandwidth, specified as a positive scalar. Units are in hertz.

Example: 100e3

Data Types: double

**WaveformType — Type of detected waveform**

0 (default) | nonnegative integer

Type of detected waveform, specified as a nonnegative integer.

Example: 1

Data Types: double

**ProcessingGain — Processing gain**

0 (default) | scalar

Processing gain when demodulating an emitted signal waveform, specified as a scalar. Processing gain is achieved by emitting a signal over a bandwidth which is greater than the minimum bandwidth necessary to send the information contained in the signal. Units are in dB.

Example: 20

Data Types: double

**Usage****Syntax**

```
radarsigs = emitter(platform,simTime)  
[radarsigs,config] = emitter(platform,simTime)
```



## Description

`radarsigs = emitter(platform, simTime)` creates radar signals, `radarsigs`, from emitter on the platform at the current simulation time, `simTime`. The emitter object can simultaneously generate signals from multiple emitters on the platform.

`[radarsigs, config] = emitter(platform, simTime)` also returns the emitter configurations, `config`, at the current simulation time.

## Input Arguments

### platform – emitter platform

object | structure

Emitter platform, specified as a platform object, `Platform`, or a platform structure:

Field	Description
PlatformID	Unique identifier for the platform, specified as a scalar positive integer. This is a required field which has no default value.
ClassID	User-defined integer used to classify the type of target, specified as a nonnegative integer. Zero is reserved for unclassified platform types and is the default value.
Position	Position of target in scenario coordinates, specified as a real-valued 1-by-3 vector. This is a required field. There is no default value. Units are in meters.
Velocity	Velocity of platform in scenario coordinates, specified as a real-valued 1-by-3 vector. Units are in meters per second. The default is <code>[0 0 0]</code> .
Speed	Speed of the platform in the scenario frame specified as a real scalar. When speed is specified, the platform velocity is aligned with its orientation. Specify either the platform speed or velocity, but not both. Units are in meters per second. The default is <code>0</code> .
Acceleration	Acceleration of the platform in scenario coordinates specified as a 1-by-3 row vector in meters per second-squared. The default is <code>[0 0 0]</code> .
Orientation	Orientation of the platform with respect to the local scenario NED coordinate frame, specified as a scalar quaternion or a 3-by-3 rotation matrix. Orientation defines the frame rotation from the local NED coordinate system to the current platform body coordinate system. Units are dimensionless. The default is <code>quaternion(1, 0, 0, 0)</code> .

Field	Description
AngularVelocity	Angular velocity of platform in scenario coordinates, specified as a real-valued 1-by-3 vector. The magnitude of the vector defines the angular speed. The direction defines the axis of clockwise rotation. Units are in degrees per second. The default is [0 0 0].
Signatures	Cell array of signatures defining the visibility of the platform to emitters and sensors in the scenario. The default is the cell {rcsSignature}.

**simTime — Current simulation time**

nonnegative scalar

Current simulation time, specified as a positive scalar. The `radarScenario` object calls the radar sensor at regular time intervals. The radar emitter generates new signals at intervals defined by the `UpdateInterval` property. The value of the `UpdateInterval` property must be an integer multiple of the simulation time interval. Updates requested from the emitter between update intervals contain no detections. Units are in seconds.

Example: 10.5

Data Types: double

**Output Arguments****radarsigs — Radar emissions**

array of radar emission objects

Radar emissions, returned as an array of `radarEmission` objects.

**config — Current emitter configuration**

structure array

Current emitter configurations, returned as an array of structures.

Field	Description
SensorIndex	Unique sensor index, returned as a positive integer.
IsValidTime	Valid detection time, returned as <code>true</code> or <code>false</code> . <code>IsValidTime</code> is <code>false</code> when detection updates are requested between update intervals specified by the update rate.
IsScanDone	<code>IsScanDone</code> is <code>true</code> when the sensor has completed a scan.
FieldOfView	Field of view of the sensor, returned as a 2-by-1 vector of positive real values, [ <code>azfov</code> ; <code>elfov</code> ]. <code>azfov</code> and <code>elfov</code> represent the field of view in azimuth and elevation, respectively.

MeasurementParameters	Sensor measurement parameters, returned as an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current sensor frame.
-----------------------	---

Data Types: struct

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

## Specific to radarEmitter

coverageConfig    Sensor and emitter coverage configuration  
 perturbations    Perturbation defined on object  
 perturb          Apply perturbations to object

## Common to All System Objects

step            Run System object algorithm  
 release        Release resources and allow changes to System object property values and input characteristics  
 reset          Reset internal states of System object

## Examples

### Model Radar Jammer

Create an emitter that stares from the front of a jammer.

Create a platform to mount the jammer on.

```
plat = struct( ...
    'PlatformID', 1, ...
    'Position', [0 0 0]);
```

Create an emitter that stares from the front of the jamming platform.

```
jammer = radarEmitter(1, 'No scanning');
```

Emit the jamming waveform.

```
time = 0;
sig = jammer(plat, time)

sig =
  radarEmission with properties:

      PlatformID: 1
      EmitterIndex: 1
```

```
OriginPosition: [0 0 0]
OriginVelocity: [0 0 0]
Orientation: [1x1 quaternion]
FieldOfView: [1 5]
CenterFrequency: 300000000
Bandwidth: 3000000
WaveformType: 0
ProcessingGain: 0
PropagationRange: 0
PropagationRangeRate: 0
EIRP: 100
RCS: 0
```

### Model Radar Emitter for Air Traffic Control Tower

Model an radar emitter for an air traffic control tower.

Simulate one full rotation of the tower.

```
rpm = 12.5;
scanrate = rpm*360/60;
fov = [1.4;5];
updaterate = scanrate/fov(1);
```

Create a `radarScenario` object to manage the motion of the platforms.

```
scene = radarScenario('UpdateRate', updaterate, ...
    'StopTime', 60/rpm);
```

Add a platform to the scenario to host the air traffic control tower.

```
tower = platform(scene);
```

Create an emitter that provides 360 degree surveillance.

```
radarTx = radarEmitter(1,'Rotator', ...
    'UpdateRate',updaterate, ...
    'MountingLocation',[0 0 -15], ...
    'MaxMechanicalScanRate',scanrate, ...
    'FieldOfView',fov);
```

Attach the emitter to the tower.

```
tower.Emitters = radarTx
```

```
tower =
```

```
Platform with properties:
```

```
PlatformID: 1
ClassID: 0
Position: [0 0 0]
Orientation: [0 0 0]
Dimensions: [1x1 struct]
Trajectory: [1x1 kinematicTrajectory]
PoseEstimator: [1x1 insSensor]
```

```

    Emitters: {[1x1 radarEmitter]}
    Sensors: {}
    Signatures: {[1x1 rcsSignature]}

```

Rotate the antenna and emit the radar waveform.

```

loggedData = struct('Time', zeros(0,1), ...
    'Orientation', quaternion.zeros(0, 1));
while advance(scene)
    time = scene.SimulationTime;
    txSig = emit(tower, time);
    loggedData.Time = [loggedData.Time; time];
    loggedData.Orientation = [loggedData.Orientation; ...
        txSig{1}.Orientation];
end

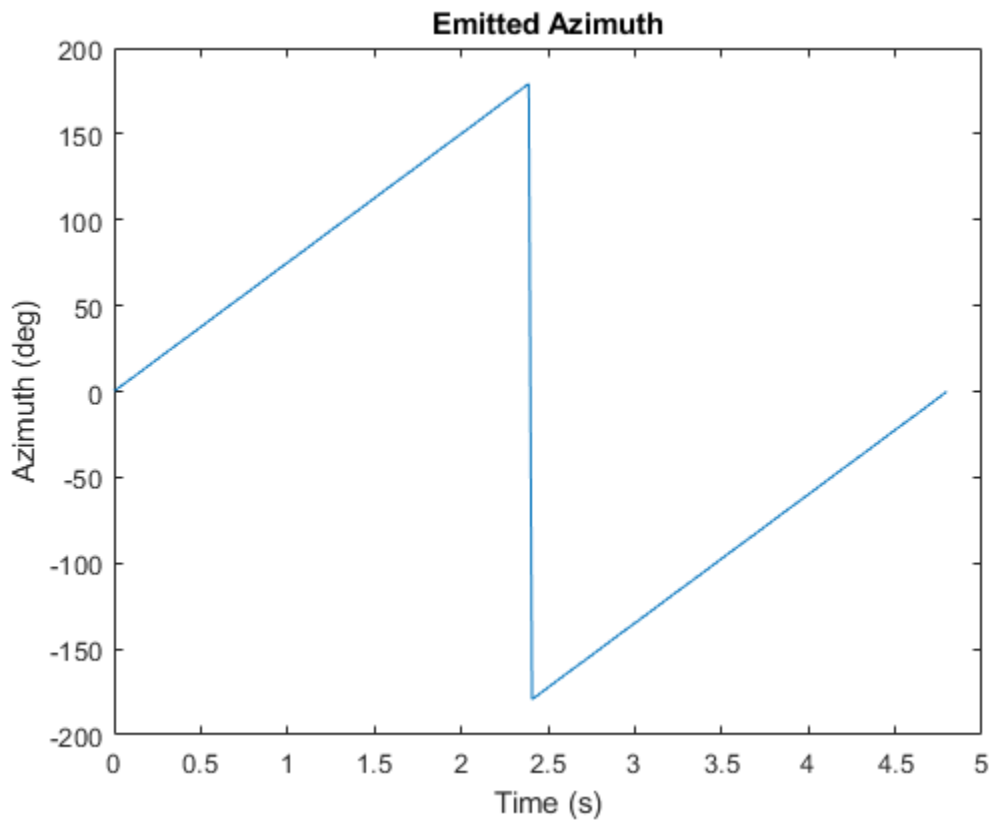
```

Plot the emitter azimuth direction.

```

angles = eulerd(loggedData.Orientation, 'zyx', 'frame');
plot(loggedData.Time, angles(:,1))
title('Emitted Azimuth')
xlabel('Time (s)')
ylabel('Azimuth (deg)')

```



## More About

### Convenience Syntaxes

The convenience syntaxes set several properties together to model a specific type of radar emitter.

#### No Scanning

Sets ScanMode to 'No scanning'.

#### Raster Scanning

This syntax sets these properties:

Property	Value
ScanMode	'Mechanical'
HasElevation	true
MaxMechanicalScanRate	[75;75]
MechanicalScanLimits	[-45 45; -10 0]
ElectronicScanLimits	[-45 45; -10 0]

You can change the ScanMode property to 'Electronic' to perform an electronic raster scan over the same volume as a mechanical scan.

#### Rotator Scanning

This syntax sets these properties:

Property	Value
ScanMode	'Mechanical'
FieldOfView	[1;10]
HasElevation	false or true
MechanicalScanLimits	[0 360; -10 0]
ElevationResolution	10/sqrt(12)

#### Sector Scanning

This syntax sets these properties:

Property	Value
ScanMode	'Mechanical'
FieldOfView	[1;10]
HasElevation	false
MechanicalScanLimits	[-45 45; -10 0]
ElectronicScanLimits	[-45 45; -10 0]
ElevationResolution	10/sqrt(12)

Changing the ScanMode property to 'Electronic' lets you perform an electronic raster scan over the same volume as a mechanical scan.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The object functions, `perturbations` and `perturb`, do not support code generation.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

`radarEmission` | `platform` | `targetPoses` | `emissionsInBody`

**Introduced in R2021a**

## rccSignature

Radar cross-section pattern

### Description

`rccSignature` creates a radar cross-section (RCS) signature object. You can use this object to model an angle-dependent and frequency-dependent radar cross-section pattern. The radar cross-section determines the intensity of reflected radar signal power from a target. The object models only non-polarized signals. The object support several Swerling fluctuation models.

### Creation

#### Syntax

```
rccsig = rccSignature  
rccsig = rccSignature(Name,Value)
```

#### Description

`rccsig = rccSignature` creates an `rccSignature` object with default property values.

`rccsig = rccSignature(Name,Value)` sets object properties using one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1,Value1, . . . ,NameN,ValueN`. Any unspecified properties take default values.

---

**Note** You can only set property values of `rccSignature` when constructing the object. The property values are not changeable after construction.

---

### Properties

#### Pattern — Sampled radar cross-section pattern

[10 10; 10 10] (default) |  $Q$ -by- $P$  real-valued matrix |  $Q$ -by- $P$ -by- $K$  real-valued array

Sampled radar cross-section (RCS) pattern, specified as a scalar, a  $Q$ -by- $P$  real-valued matrix, or a  $Q$ -by- $P$ -by- $K$  real-valued array. The pattern is an array of RCS values defined on a grid of elevation angles, azimuth angles, and frequencies. Azimuth and elevation are defined in the body frame of the target.

- $Q$  is the number of RCS samples in elevation.
- $P$  is the number of RCS samples in azimuth.
- $K$  is the number of RCS samples in frequency.

$Q$ ,  $P$ , and  $K$  usually match the length of the vectors defined in the `Elevation`, `Azimuth`, and `Frequency` properties, respectively, with these exceptions:



- To model an RCS pattern for an elevation cut (constant azimuth), you can specify the RCS pattern as a  $Q$ -by-1 vector or a 1-by- $Q$ -by- $K$  matrix. Then, the elevation vector specified in the `Elevation` property must have length 2.
- To model an RCS pattern for an azimuth cut (constant elevation), you can specify the RCS pattern as a 1-by- $P$  vector or a 1-by- $P$ -by- $K$  matrix. Then, the azimuth vector specified in the `Azimuth` property must have length 2.
- To model an RCS pattern for one frequency, you can specify the RCS pattern as a  $Q$ -by- $P$  matrix. Then, the frequency vector specified in the `Frequency` property must have length-2.

Example: [10,0;0,-5]

Data Types: double

### **Azimuth — Azimuth angles**

[ -180 180 ] (default) | length- $P$  real-valued vector

Azimuth angles used to define the angular coordinates of each column of the matrix or array, specified by the `Pattern` property. Specify the azimuth angles as a length- $P$  vector.  $P$  must be greater than two. Angle units are in degrees.

When the `Pattern` property defines an elevation cut, `Azimuth` must be a 2-element vector defining the minimum and maximum azimuth view angles over which the elevation cut is considered valid.

Example: [-45:0.5:45]

Data Types: double

### **Elevation — Elevation angles**

[ -90 90 ] (default) | length- $Q$  real-valued vector

Elevation angles used to define the coordinates of each row of the matrix or array, specified by the `Pattern` property. Specify the elevation angles as a length- $Q$  vector.  $Q$  must be greater than two. Angle units are in degrees.

When the `Pattern` property defines an azimuth cut, `Elevation` must be a 2-element vector defining the minimum and maximum elevation view angles over which the azimuth cut is considered valid.

Example: [-30:0.5:30]

Data Types: double

### **Frequency — Pattern frequencies**

[0 1e20] (default) |  $K$ -element vector of positive scalars

Frequencies used to define the applicable RCS for each page of the `Pattern` property, specified as a  $K$ -element vector of positive scalars.  $K$  is the number of RCS samples in frequency.  $K$  must be no less than two. Frequency units are in hertz.

When the `Pattern` property is a matrix, `Frequency` must be a 2-element vector defining the minimum and maximum frequencies over which the pattern values are considered valid.

Example: [0:0.1:30]

Data Types: double

### **FluctuationModel — Statistical signature fluctuation model**

'Swerling0' (default) | 'Swerling1' | 'Swerling3'

Fluctuation models, specified as 'Swerling0', 'Swerling1' or 'Swerling3'. Swerling cases 2 and 4 are not modeled as these are determined how the target is sample, not an inherent target property.

Model	Description
'Swerling0'	The target RCS is assumed to be non-fluctuating. In this case the instantaneous RCS signature value retrieved by the value method is deterministic. This model represents ideal radar targets with an RCS that remains constant in time across the range of aspect angles of interest, e.g., a conducting sphere and various corner reflectors.
'Swerling1'	The target is assumed to be made up of many independent scatterers of equal size. This model is typically used to represent aircraft. The instantaneous RCS signature value returned by the value method in this case is a random variable distributed according to the exponential distribution with a mean determined by the Pattern property.
'Swerling3'	The target is assumed to have one large dominant scatterer and several small scatterers. The RCS of the dominant scatterer equals $1 + \sqrt{2}$ times the sum of the RCS of other scatterers. This model can be used to represent helicopters and propeller driven aircraft. In this case the instantaneous RCS signature's value returned by the value method is a random variable distributed according to the 4th degree chi-square distribution with mean determined by the Pattern property.

Data Types: char | string

## Object Functions

value Radar cross-section at specified angle and frequency  
toStruct Convert to structure

## Examples

### Radar Cross-Section of Ellipsoid

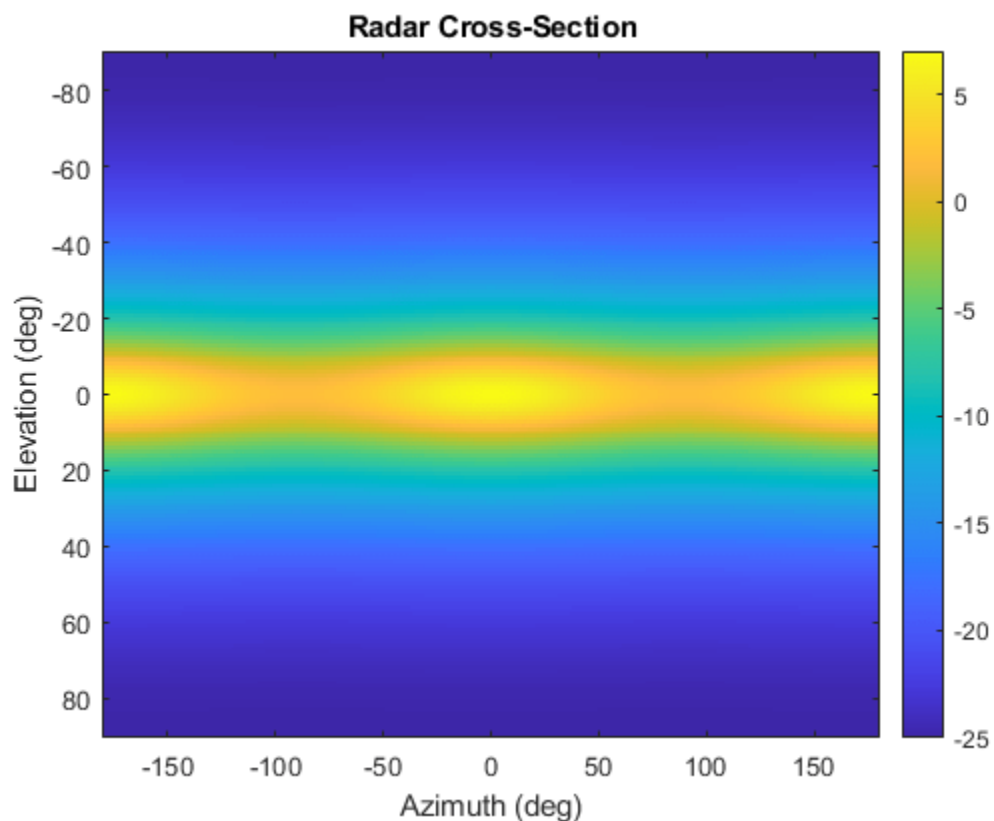
Specify the radar cross-section (RCS) of a triaxial ellipsoid and plot RCS values along an azimuth cut.

Specify the lengths of the axes of the ellipsoid. Units are in meters.

```
a = 0.15;
b = 0.20;
c = 0.95;
```

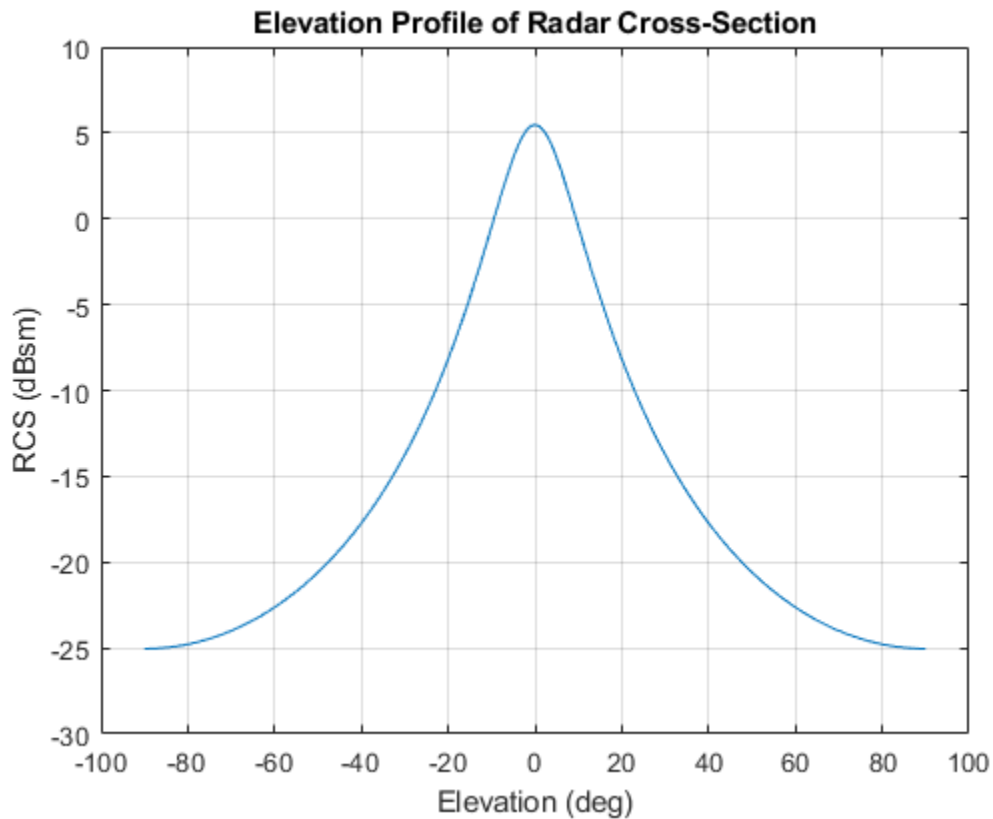
Create an RCS array. Specify the range of azimuth and elevation angles over which RCS is defined. Then, use an analytical model to compute the radar cross-section of the ellipsoid. Create an image of the RCS.

```
az = [-180:1:180];
el = [-90:1:90];
rsc = rsc_ellipsoid(a,b,c,az,el);
rscdb = 10*log10(rsc);
imagesc(az,el,rscdb)
title('Radar Cross-Section')
xlabel('Azimuth (deg)')
ylabel('Elevation (deg)')
colorbar
```



Create an rcsSignature object and plot an elevation cut at 30° azimuth.

```
rcssig = rcsSignature('Pattern',rscdb,'Azimuth',az,'Elevation',el,'Frequency',[300e6 300e6]);
rscdb1 = value(rcssig,30,el,300e6);
plot(el,rscdb1)
grid
title('Elevation Profile of Radar Cross-Section')
xlabel('Elevation (deg)')
ylabel('RCS (dBsm)')
```



```
function rcs = rcs_ellipsoid(a,b,c,az,el)
sinaz = sind(az);
cosaz = cosd(az);
sintheta = sind(90 - el);
costheta = cosd(90 - el);
denom = (a^2*(sintheta'.^2)*cosaz.^2 + b^2*(sintheta'.^2)*sinaz.^2 + c^2*(costheta'.^2)*ones(size(sintheta',2)));
rcs = (pi*a^2*b^2*c^2)./denom;
end
```

### RCS Distribution of Swerling 1 Target

Import the radar cross-section (RCS) measurements of a 1/5th scale Boeing 737. Load the RCS data into an `rcsSignature` object. Assume the RCS follows a Swerling 1 distribution.

```
load('RCSSignatureExampleData.mat','boeing737');
rcs = rcsSignature('Pattern',boeing737.RCSdBsm, ...
    'Azimuth', boeing737.Azimuth, 'Elevation',boeing737.Elevation, ...
    'Frequency',boeing737.Frequency, 'FluctuationModel', 'Swerling1');
```

Set the seed of the random number generator for reproducibility of example.

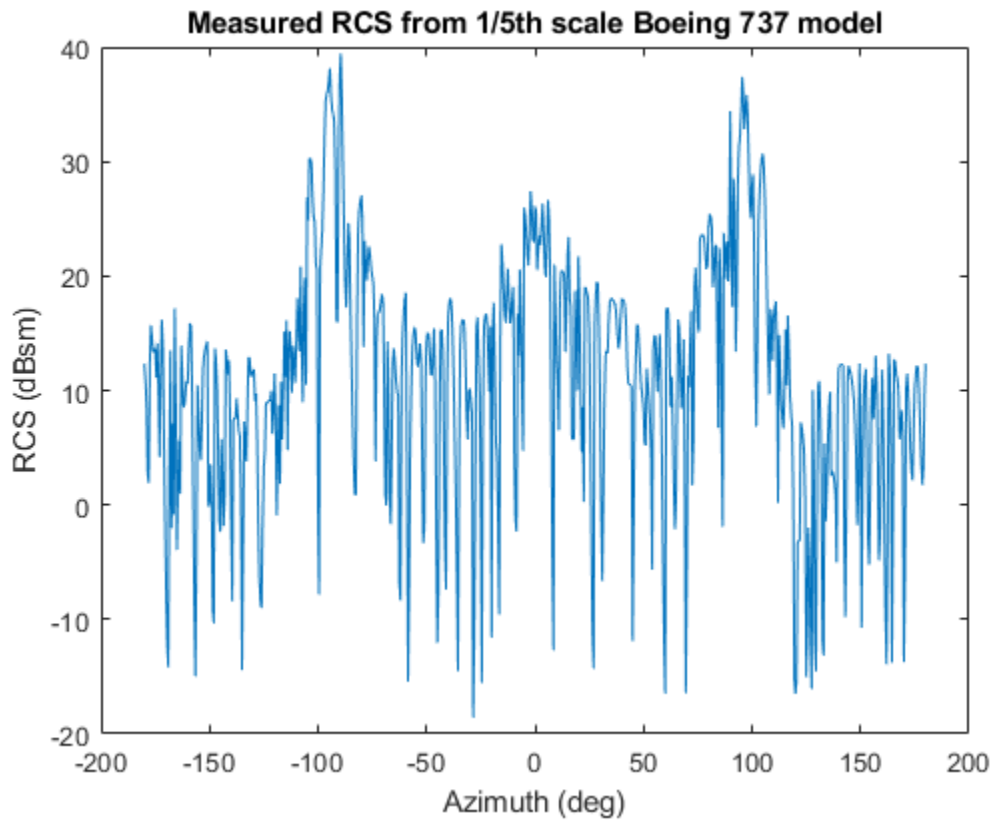
```
rng(3231)
```

Plot sample RCS versus azimuth angle.

```

plot(rcs.Azimuth,rsc.Pattern)
xlabel('Azimuth (deg)'); ylabel('RCS (dBsm)')
title('Measured RCS from 1/5th scale Boeing 737 model')

```



Construct an RCS histogram and display the mean value.

```

N = 1000;
val = zeros(1,N);
for k = 1:N
    [val(k),expval] = value(rcs,-5,0,800.0e6);
end

```

Convert to power units.

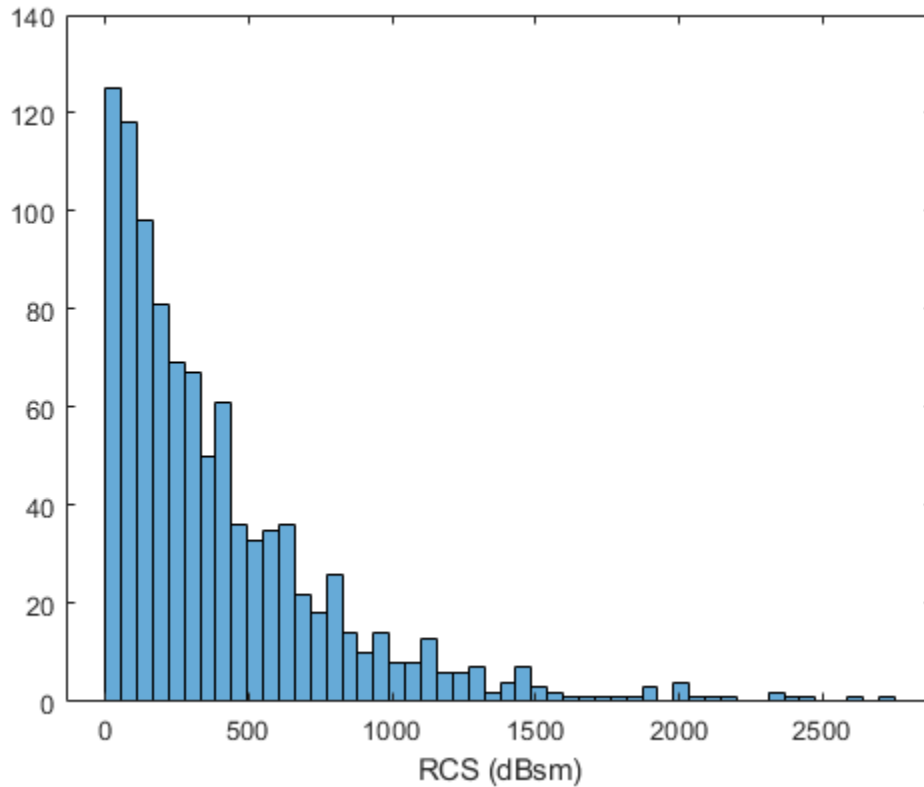
```
mean(db2pow(val))
```

```
ans = 406.9799
```

```

histogram(db2pow(val),50)
xlabel("RCS (dBsm)")

```



## References

[1] Richards, Mark A. *Fundamentals of Radar Signal Processing*. New York, McGraw-Hill, 2005.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

value | toStruct

**Introduced in R2021a**

## value

Radar cross-section at specified angle and frequency

### Syntax

```
rcsval = value(rcssig,az,el,freq)
[rcsval,expval] = value(rcssig,az,el,freq)
```

### Description

`rcsval = value(rcssig,az,el,freq)` returns the value, `rcsval`, of the radar cross-section (RCS) specified by the radar signature object, `rcssig`, computed at the specified azimuth `az`, elevation `el`, and frequency `freq`. If the specified azimuth and elevation is outside of the region in which the RCS signature is defined, the RCS value, `rcsval`, is returned as `-Inf` in dBsm.

`[rcsval,expval] = value(rcssig,az,el,freq)` returns the expected values of the radar cross-section.

### Input Arguments

#### **rcssig** — RCS signature object

`rcsSignature` object

Radar cross-section signature, specified as an `rcsSignature` object.

#### **az** — Azimuth angle

scalar | length-*M* real-valued vector

Azimuth angle, specified as scalar or length-*M* real-valued vector. Units are in degrees. The `az`, `el`, and `freq` arguments must have the same size. You can, however, specify one or two arguments as scalars, in which case the arguments are expanded to length-*M*.

Data Types: `double`

#### **el** — Elevation angle

scalar | length-*M* real-valued vector

Elevation angle, specified as scalar or length-*M* real-valued vector. The `az`, `el`, and `freq` arguments must have the same size. You can, however, specify one or two arguments as scalars, in which case the arguments are expanded to length-*M*. Units are in degrees.

Data Types: `double`

#### **freq** — RCS frequency

positive scalar | length-*M* vector with positive, real elements

RCS frequency, specified as a positive scalar or length-*M* vector with positive, real elements. The `az`, `el`, and `freq` arguments must have the same size. You can, however, specify one or two arguments as scalars, in which case the arguments are expanded to length-*M* vectors. Units are in Hertz.

Example: `100e6`

Data Types: double

## Output Arguments

### **rscval** — Radar cross-section

scalar | real-valued length- $M$  vector

Radar cross-section, returned as a scalar or real-valued length- $M$  vector. Units are in dBsm.

### **expval** — Expected values of radar cross section

scalar (default) | real-valued length- $M$  vector

Expected values of radar cross section, returned as a scalar or as a real-valued length- $M$  vector. The dimensions of `expval` are the same as `rscval`. Units are in dBsm.

Data Types: double

## Examples

### Radar Cross-Section of Ellipsoid

Specify the radar cross-section (RCS) of a triaxial ellipsoid and plot RCS values along an azimuth cut.

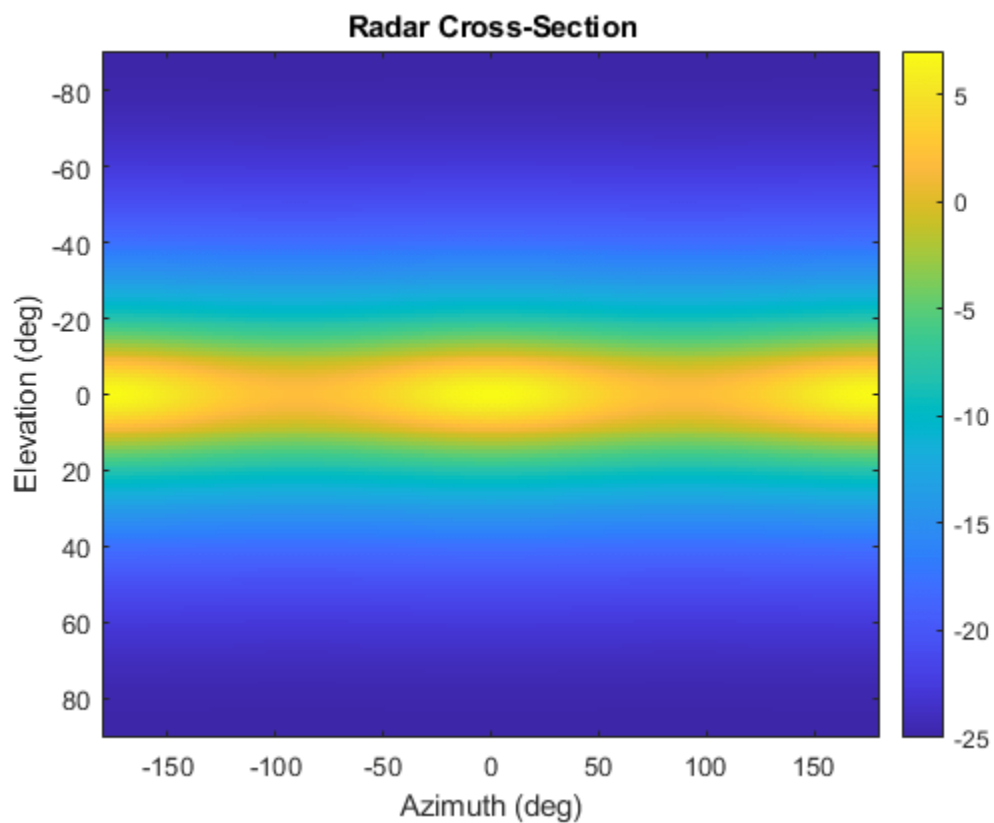
Specify the lengths of the axes of the ellipsoid. Units are in meters.

```
a = 0.15;  
b = 0.20;  
c = 0.95;
```

Create an RCS array. Specify the range of azimuth and elevation angles over which RCS is defined. Then, use an analytical model to compute the radar cross-section of the ellipsoid. Create an image of the RCS.

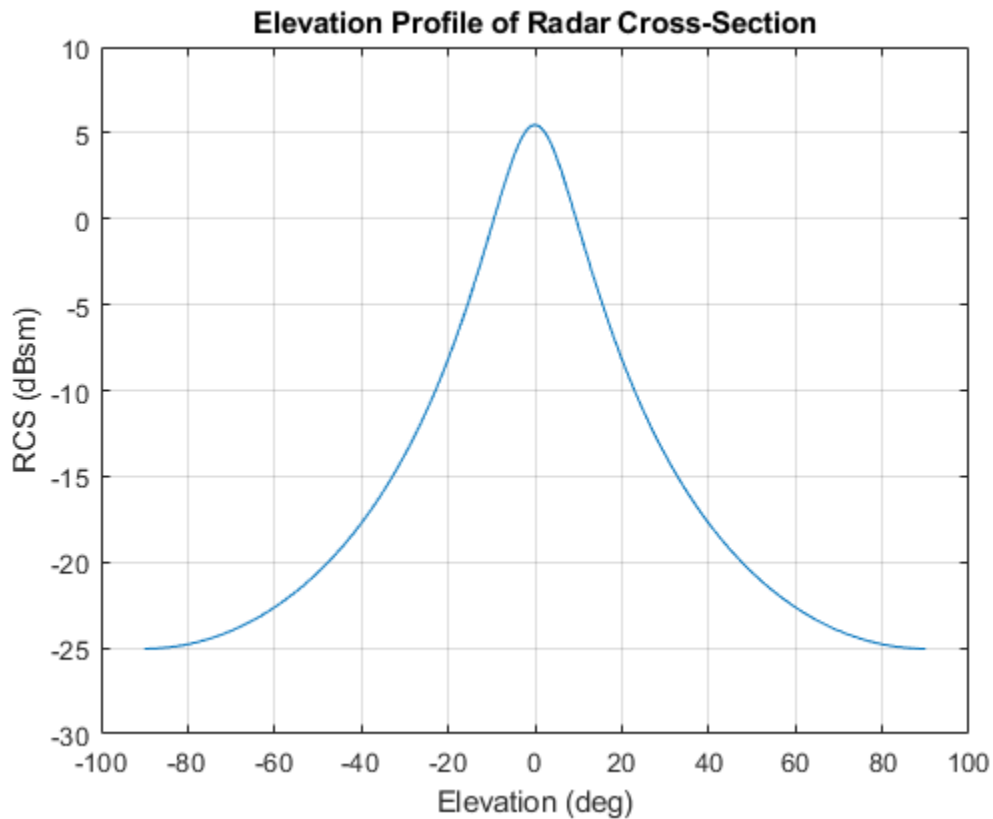
```
az = [-180:1:180];  
el = [-90:1:90];  
rsc = rsc_ellipsoid(a,b,c,az,el);  
rscdb = 10*log10(rsc);  
imagesc(az,el,rscdb)  
title('Radar Cross-Section')  
xlabel('Azimuth (deg)')  
ylabel('Elevation (deg)')  
colorbar
```





Create an `rccSignature` object and plot an elevation cut at 30° azimuth.

```
rccsig = rccSignature('Pattern',rcsdb,'Azimuth',az,'Elevation',el,'Frequency',[300e6 300e6]);  
rcsdbl = value(rccsig,30,el,300e6);  
plot(el,rcsdbl)  
grid  
title('Elevation Profile of Radar Cross-Section')  
xlabel('Elevation (deg)')  
ylabel('RCS (dBsm)')
```



```
function rcs = rcs_ellipsoid(a,b,c,az,el)
sinaz = sind(az);
cosaz = cosd(az);
sintheta = sind(90 - el);
costheta = cosd(90 - el);
denom = (a^2*(sintheta'.^2)*cosaz.^2 + b^2*(sintheta'.^2)*sinaz.^2 + c^2*(costheta'.^2)*ones(size(sintheta',2)));
rcs = (pi*a^2*b^2*c^2)./denom;
end
```

### RCS Distribution of Swerling 1 Target

Import the radar cross-section (RCS) measurements of a 1/5th scale Boeing 737. Load the RCS data into an `rcsSignature` object. Assume the RCS follows a Swerling 1 distribution.

```
load('RCSSignatureExampleData.mat','boeing737');
rcs = rcsSignature('Pattern',boeing737.RCSdBsm, ...
    'Azimuth', boeing737.Azimuth, 'Elevation',boeing737.Elevation, ...
    'Frequency',boeing737.Frequency, 'FluctuationModel', 'Swerling1');
```

Set the seed of the random number generator for reproducibility of example.

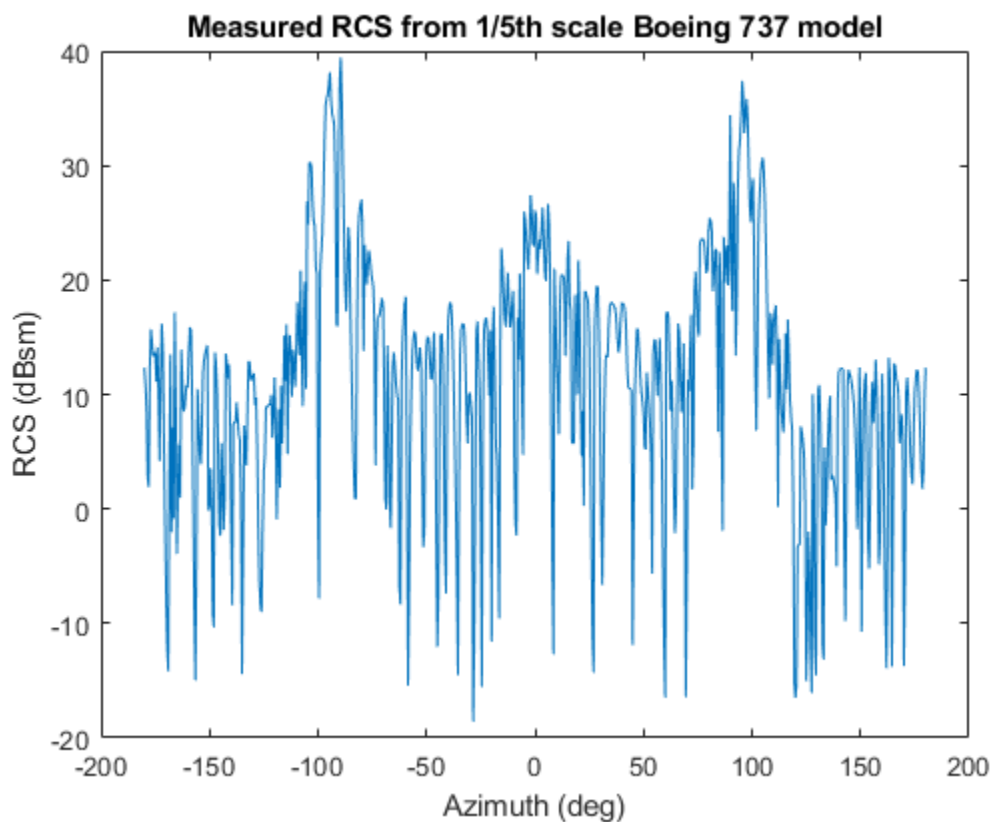
```
rng(3231)
```

Plot sample RCS versus azimuth angle.

```

plot(rcs.Azimuth,rcs.Pattern)
xlabel('Azimuth (deg)'); ylabel('RCS (dBsm)')
title('Measured RCS from 1/5th scale Boeing 737 model')

```



Construct an RCS histogram and display the mean value.

```

N = 1000;
val = zeros(1,N);
for k = 1:N
    [val(k),expval] = value(rcs,-5,0,800.0e6);
end

```

Convert to power units.

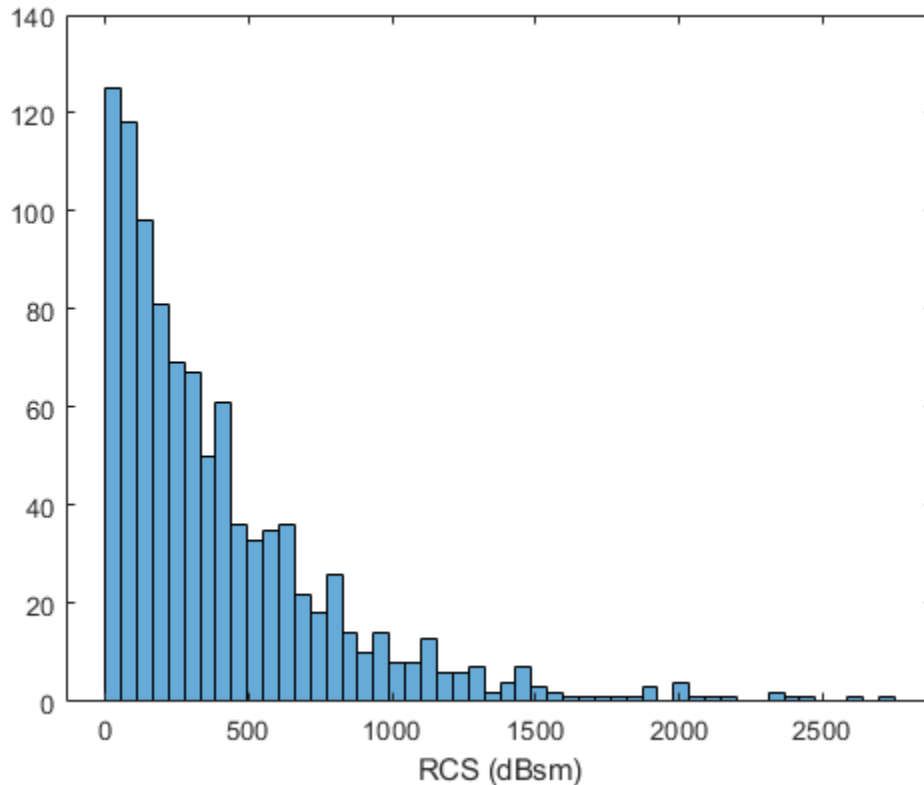
```
mean(db2pow(val))
```

```
ans = 406.9799
```

```

histogram(db2pow(val),50)
xlabel("RCS (dBsm)")

```



## Algorithms

The RCS signature, is first linearly interpolated at the specified azimuth, `az`, and elevation, `el`, view angles for the provided frequencies, `freq`. The interpolated signature is then used as an expected value of a probability distribution that generates a signature pattern value according to the RCS fluctuation model specified by the `FluctuationModel` property. `az` and `el` are specified in degrees and are defined in the body frame of the pattern. `freq` is in hertz.

If `FluctuationModel` is 'Swerling0', the returned pattern value is a deterministic constant equal to the interpolated signature.

If `FluctuationModel` is 'Swerling1', the returned pattern value is a random variable distributed according to an exponential distribution with a mean value equal to the interpolated signature.

If `FluctuationModel` is 'Swerling3', the returned pattern value is a random variable distributed according to a chi-square distribution with four degrees of freedom and a mean value equal to the interpolated signature.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

**See Also**

rcsSignature | toStruct

**Introduced in R2021a**

## toStruct

Convert to structure

### Syntax

```
rscStruct = toStruct(rscSig)
```

### Description

`rscStruct = toStruct(rscSig)` converts the `rscSignature` object `rscSig` to a structure `rscStruct`. The field names of the returned structure are the same as the property names of the `rscSignature` object.

### Examples

#### Convert rscSignature to Structure

Create a `rscSignature` object.

```
rscSig = rscSignature
```

```
rscSig =  
  rscSignature with properties:  
  
    FluctuationModel: Swerling0  
          Pattern: [2x2 double]  
        Azimuth: [-180 180]  
      Elevation: [2x1 double]  
    Frequency: [0 1.0000e+20]
```

Convert the signature to a structure.

```
rscStruct = toStruct(rscSig)  
  
rscStruct = struct with fields:  
    Pattern: [2x2 double]  
  Azimuth: [-180 180]  
Elevation: [2x1 double]  
Frequency: [0 1.0000e+20]
```

### Input Arguments

#### **rscSig** — RCS signature

`rscSignature` object

RCS signature, specified as an `rscSignature` object.

## **Output Arguments**

### **rcsStruct — RCS structure**

structure

RCS structure, returned as a structure.

**Introduced in R2021a**

# radarEmission

Emitted radar signal structure

## Description

The `radarEmission` class creates a radar emission object. This object contains all the properties that describe a signal radiated by a radar source.

## Creation

### Syntax

```
signal = radarEmission  
signal = radarEmission(Name, Value)
```

### Description

`signal = radarEmission` creates a `radarEmission` object with default properties. The object represents radar signals from emitters, channels, and sensors.

`signal = radarEmission(Name, Value)` sets object properties specified by one or more `Name, Value` pair arguments. `Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Properties

### **PlatformID** — Platform identifier

positive integer

Platform identifier, specified as a positive integer. The emitter is mounted on the platform with this ID. Each platform identifier is unique within a scenario.

Example: 5

Data Types: double

### **EmitterIndex** — Emitter identifier

positive integer

Emitter identifier, specified as a positive integer. Each emitter index is unique.

Example: 2

Data Types: double

### **OriginPosition** — Location of emitter

[0 0 0] (default) | 1-by-3 real-valued vector



Location of the emitter in scenario coordinates, specified as a 1-by-3 real-valued vector. Units are in meters.

Example: [100 -500 1000]

Data Types: double

#### **OriginVelocity — Velocity of emitter**

[0 0 0] (default) | 1-by-3 real-valued vector

Velocity of the emitter in scenario coordinates, specified as a 1-by-3 real-valued vector. Units are in meters per second.

Example: [0 -50 100]

Data Types: double

#### **Orientation — Orientation of emitter**

quaternion(1,0,0,0) (default) | quaternion | 3-by-3 real-valued orthogonal matrix

Orientation of the emitter in scenario coordinates, specified as a quaternion or 3-by-3 real-valued orthogonal matrix.

Example: eye(3)

Data Types: double

#### **FieldOfView — Field of view of emitter**

[180,180] | 2-by-1 vector of positive real values

Field of view of emitter, specified as a 2-by-1 vector of positive real values, [azfov, elfov]. The field of view defines the total angular extent of the signal emitted. The azimuth filed of view azfov must lie in the interval (0,360]. The elevation filed of view elfov must lie in the interval (0,180].

Example: [140;70]

Data Types: double

#### **EIRP — Effective isotropic radiated power**

0 (default) | scalar

Effective isotropic radiated power, specified as a scalar. Units are in dB.

Example: 10

Data Types: double

#### **RCS — Cumulative radar cross-section**

0 (default) | scalar

Cumulative radar cross-section, specified as a scalar. Units are in dBsm.

Example: 10

Data Types: double

#### **CenterFrequency — Center frequency of radar signal**

300e6 (default) | positive scalar

Center frequency of the signal, specified as a positive scalar. Units are in Hz.

Example: 100e6

Data Types: double

**Bandwidth — Half-power bandwidth of radar signal**

30e6 (default) | positive scalar

Half-power bandwidth of the radar signal, specified as a positive scalar. Units are in Hz.

Example: 5e3

Data Types: double

**WaveformType — Waveform type identifier**

0 (default) | nonnegative integer

Waveform type identifier, specified as a nonnegative integer.

Example: 5e3

Data Types: double

**ProcessingGain — Processing gain**

0 (default) | scalar

Processing gain associated with the signal waveform, specified as a scalar. Units are in dB.

Example: 10

Data Types: double

**PropagationRange — Distance signal propagates**

0 (default) | nonnegative scalar

Total distance over which the signal has propagated, specified as a nonnegative scalar. For direct-path signals, the range is zero. Units are in meters.

Example: 1000

Data Types: double

**PropagationRangeRate — Range rate of signal propagation path**

0 (default) | scalar

Total range rate for the path over which the signal has propagated, specified as a scalar. For direct-path signals, the range rate is zero. Units are in meters per second.

Example: 10

Data Types: double

**Examples****Create Radar Emission Object**

Create a radarEmission object with specified properties.

```
signal = radarEmission('PlatformID',10,'EmitterIndex',25, ...
    'OriginPosition',[100,3000,50],'EIRP',10,'CenterFrequency',200e6, ...
    'Bandwidth',10e3)
```

```
signal =
    radarEmission with properties:

        PlatformID: 10
        EmitterIndex: 25
        OriginPosition: [100 3000 50]
        OriginVelocity: [0 0 0]
        Orientation: [1x1 quaternion]
        FieldOfView: [180 180]
        CenterFrequency: 200000000
        Bandwidth: 10000
        WaveformType: 0
        ProcessingGain: 0
        PropagationRange: 0
        PropagationRangeRate: 0
        EIRP: 10
        RCS: 0
```

### Detect Radar Emission with radarDataGenerator

Create a radar emission and then detect the emission using a radarDataGenerator object.

First, create a radar emission.

```
orient = quaternion([180 0 0],'eulerd','zyx','frame');
rfSig = radarEmission('PlatformID',1,'EmitterIndex',1,'EIRP',100, ...
    'OriginPosition',[30 0 0],'Orientation',orient);
```

Then, create an ESM sensor using radarDataGenerator.

```
sensor = radarDataGenerator(1,'DetectionMode','ESM');
```

Detect the RF emission.

```
time = 0;
[dets,numDets,config] = sensor(rfSig,time)
```

```
dets = 1x1 cell array
    {1x1 objectDetection}
```

```
numDets = 1
```

```
config = struct with fields:
    SensorIndex: 1
    IsValidTime: 1
    IsScanDone: 0
    FieldOfView: [1 5]
    RangeLimits: [0 Inf]
    RangeRateLimits: [0 Inf]
    MeasurementParameters: [1x1 struct]
```

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

radarEmitter | radarChannel

**Introduced in R2021a**

# radarChannel

Free space propagation and reflection of radar signals

## Syntax

```
radarsigout = radarChannel(radarsigin,platforms)
radarsigout = radarChannel(radarsigin,platforms,'HasOcclusion',HasOcclusion)
```

## Description

`radarsigout = radarChannel(radarsigin,platforms)` returns radar signals, `radarsigout`, as combinations of the signals, `radarsigin`, that are reflected from the platforms, `platforms`.

`radarsigout = radarChannel(radarsigin,platforms,'HasOcclusion',HasOcclusion)` also allows you to specify whether to model occlusion from extended objects.

## Examples

### Reflect Radar Emission From Platform

Create a radar emission and a platform and reflect the emission from the platform.

Create a radar emission object.

```
radarSig = radarEmission('PlatformID',1,'EmitterIndex',1,'OriginPosition',[0 0 0]);
```

Create a platform structure.

```
platfm = struct('PlatformID',2,'Position',[10 0 0],'Signatures',rcsSignature());
```

Reflect the emission from the platform.

```
sigs = radarChannel(radarSig,platfm)
```

`sigs =`

radarEmission with properties:

```

    PlatformID: 1
    EmitterIndex: 1
    OriginPosition: [0 0 0]
    OriginVelocity: [0 0 0]
    Orientation: [1x1 quaternion]
    FieldOfView: [180 180]
    CenterFrequency: 300000000
    Bandwidth: 3000000
    WaveformType: 0
    ProcessingGain: 0
    PropagationRange: 0
    PropagationRangeRate: 0
    EIRP: 0
    RCS: 0
```

### **Reflect Radar Emission From Platform within Radar Scenario**

Create a radar scenario object.

```
scenario = radarScenario;
```

Create a radarEmitter object.

```
emitter = radarEmitter(1);
```

Mount the emitter on a platform within the scenario.

```
plat = platform(scenario, 'Emitters', emitter);
```

Add another platform to reflect the emitted signal.

```
target = platform(scenario);  
target.Trajectory.Position = [30 0 0];
```

Emit the signal using the emit object function of a platform.

```
txsigs = emit(plat, scenario.SimulationTime)
```

```
txsigs = 1x1 cell array  
        {1x1 radarEmission}
```

Reflect the signal from the platforms in the scenario.

```
sigs = radarChannel(txsigs, scenario.Platforms)
```

```
sigs=2x1 cell array  
        {1x1 radarEmission}  
        {1x1 radarEmission}
```

## **Input Arguments**

### **radarsigin — Input radar signals**

array of radarEmission objects

Input radar signals, specified as an array of radarEmission objects.

### **platforms — Reflector platforms**

cell array of Platform objects | array of Platform structures

Reflector platforms, specified as a cell array of Platform objects, or an array of Platform structures:

Field	Description
PlatformID	Unique identifier for the platform, specified as a scalar positive integer. This is a required field which has no default value.
ClassID	User-defined integer used to classify the type of target, specified as a nonnegative integer. Zero is reserved for unclassified platform types and is the default value.
Position	Position of target in scenario coordinates, specified as a real-valued 1-by-3 vector. This is a required field. There is no default value. Units are in meters.
Velocity	Velocity of platform in scenario coordinates, specified as a real-valued 1-by-3 vector. Units are in meters per second. The default is [0 0 0].
Speed	Speed of the platform in the scenario frame specified as a real scalar. When speed is specified, the platform velocity is aligned with its orientation. Specify either the platform speed or velocity, but not both. Units are in meters per second. The default is 0.
Acceleration	Acceleration of the platform in scenario coordinates specified as a 1-by-3 row vector in meters per second-squared. The default is [0 0 0].
Orientation	Orientation of the platform with respect to the local scenario NED coordinate frame, specified as a scalar quaternion or a 3-by-3 rotation matrix. Orientation defines the frame rotation from the local NED coordinate system to the current platform body coordinate system. Units are dimensionless. The default is <code>quaternion(1,0,0,0)</code> .
AngularVelocity	Angular velocity of platform in scenario coordinates, specified as a real-valued 1-by-3 vector. The magnitude of the vector defines the angular speed. The direction defines the axis of clockwise rotation. Units are in degrees per second. The default is [0 0 0].
Signatures	Cell array of signatures defining the visibility of the platform to emitters and sensors in the scenario. The default is the cell <code>{rcsSignature}</code> .

If you specify an array of platform structures, set a unique `PlatformID` for each platform and set the `Position` field for each platform. Any other fields not specified are assigned default values.

#### **HasOcclusion — Enable occlusion from extended objects**

`true` | `false`

Enable occlusion from extended objects, specified as `true` or `false`. Set `HasOcclusion` to `true` to model occlusion from extended objects. Two types of occlusion (self occlusion and inter object occlusion) are modeled. Self occlusion occurs when one side of an extended object occludes another side. Inter object occlusion occurs when one extended object stands in the line of sight of another extended object or a point target. Note that both extended objects and point targets can be occluded by extended objects, but a point target cannot occlude another point target or an extended object.

Set `HasOcclusion` to `false` to disable occlusion of extended objects. This will also disable the merging of objects whose detections share a common sensor resolution cell, which gives each object in the tracking scenario an opportunity to generate a detection.

Data Types: `logical`

## Output Arguments

### **radarsigout** — Reflected radar signals

array of `radarEmission` objects

Reflected radar signals, specified as an array of `radarEmission` objects.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`radarEmission` | `radarEmitter`

**Introduced in R2021a**



# theaterPlot

Plot objects, detections, and tracks in Scenario

## Description

The `theaterPlot` object is used to display a plot of a `radarScenario`. This type of plot can be used with sensors capable of detecting objects.

To display aspects of a scenario on a theater plot:

- 1 Create a `theaterPlot` object.
- 2 Create plotters for the aspects of the scenario that you want to plot.
- 3 Use the plotters with their corresponding plot functions to display those aspects on the theater plot.

This table shows the plotter functions to use based on the scenario aspect that you want to plot.

Scenario Aspect to Plot	Plotter Creation Function	Plotter Display Function
Sensor coverage areas	<code>coveragePlotter</code>	<code>plotCoverage</code>
Sensor detections	<code>detectionPlotter</code>	<code>plotDetection</code>
Object orientation	<code>orientationPlotter</code>	<code>plotOrientation</code>
Platform	<code>platformPlotter</code>	<code>plotPlatform</code>
Track	<code>trackPlotter</code>	<code>plotTrack</code>
Object trajectory	<code>trajectoryPlotter</code>	<code>plotTrajectory</code>

## Creation

### Syntax

```
tp = theaterPlot
tp = theaterPlot(Name, Value)
```

### Description

`tp = theaterPlot` creates a theater plot in a new figure.

`tp = theaterPlot(Name, Value)` creates a theater plot in a new figure with optional input "Properties" on page 4-633 specified by one or more `Name, Value` pair arguments. Properties can be specified in any order as `Name1, Value1, . . . , NameN, ValueN`. Enclose each property name in quotes.

## Properties

### Parent — Parent axes

`theaterPlot` handle

Parent axes, specified as a theaterPlot handle. If you do not specify Parent, then theaterPlot creates axes in a new figure.

### **Plotters — Plotters created for theater plot**

array of plotter objects

Plotters created for the theater plot, specified as an array of plotter objects.

### **XLimits — Limits of x-axis**

two-element row vector

Limits of the x-axis, specified as a two-element row vector, [x1,x2]. The values x1 and x2 are the lower and upper limits, respectively, for the theater plot display. If you do not specify the limits, then the default values for the Parent property are used.

Data Types: double

### **YLimits — Limits of y-axis**

two-element row vector

Limits of the y-axis, specified as a two-element row vector, [y1,y2]. The values y1 and y2 are the lower and upper limits, respectively, for the theater plot display. If you do not specify the limits, then the default values for the Parent property are used.

Data Types: double

### **ZLimits — Limits of z-axis**

two-element row vector

Limits of the z-axis, specified as a two-element row vector, [z1,z2]. The values z1 and z2 are the lower and upper limits, respectively, for the theater plot display. If you do not specify the limits, then the default values for the Parent property are used.

Data Types: double

### **AxesUnits — Unit of each axes**

["m" "m" "m"] (default) | three-element string array

Unit of each axes, specified as a three-element string array. Each element must be "m" or "km"

Data Types: string

## **Object Functions**

### **Plotter Creation**

coveragePlotter	Create coverage plotter
detectionPlotter	Create detection plotter
orientationPlotter	Create orientation plotter
platformPlotter	Create platform plotter
trackPlotter	Create track plotter
trajectoryPlotter	Create trajectory plotter

### **Plotter Display**

plotCoverage	Plot set of coverages in theater coverage plotter
--------------	---

---

plotDetection	Plot set of detections in theater detection plotter
plotOrientation	Plot set of orientations in orientation plotter
plotPlatform	Plot set of platforms in platform plotter
plotTrack	Plot set of tracks in theater track plotter
plotTrajectory	Plot set of trajectories in trajectory plotter

## Plotter Utilities

clearData	Clear data from specific plotter of theater plot
clearPlotterData	Clear plotter data from theater plot
findPlotter	Return array of plotters associated with theater plot

## Examples

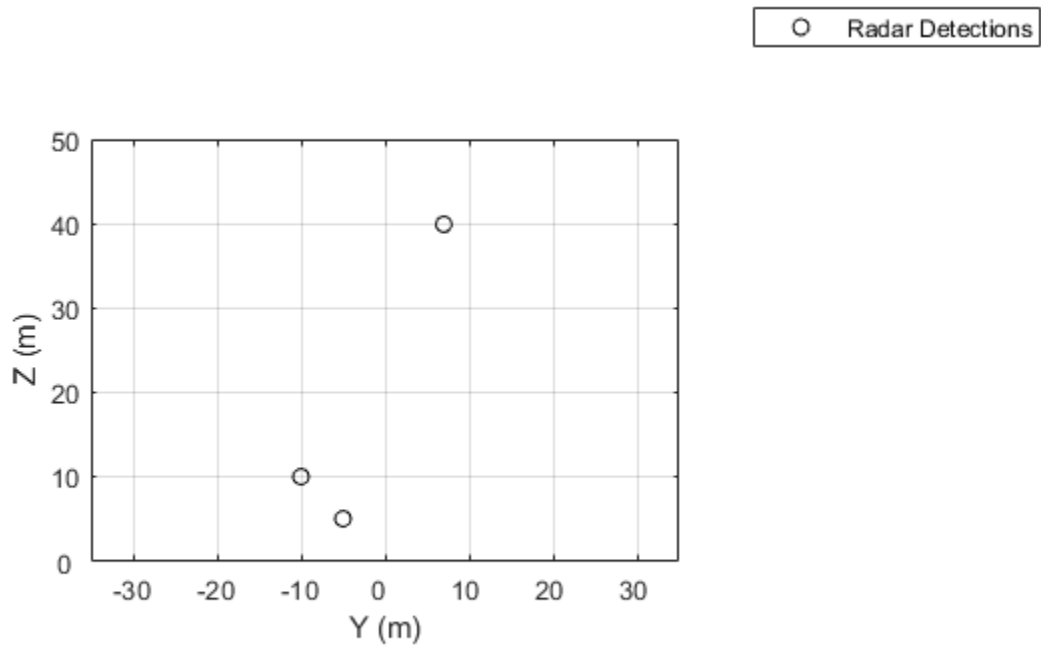
### Create and Display Theater Plot

Create a theater plot.

```
tp = theaterPlot('XLim',[0 90],'YLim',[-35 35],'ZLim',[0 50]);
```

Display radar detections with coordinates at (30, -5, 5), (50, -10, 10), and (40, 7, 40). Set the view so that you are looking on the yz-plane. Confirm the y- and z-coordinates of the radar detections are correct.

```
radarPlotter = detectionPlotter(tp,'DisplayName','Radar Detections');  
plotDetection(radarPlotter, [30 -5 5; 50 -10 10; 40 7 40])  
grid on  
view(90,0)
```



The view can be changed by opening the plot in a figure window and selecting **Tools > Rotate 3D** in the figure menu.

### Limitations

You cannot use the rectangle-zoom feature in the theaterPlot figure.

### See Also

radarScenario

**Introduced in R2021a**

# clearData

Clear data from specific plotter of theater plot

## Syntax

```
clearData(pl)
```

## Description

`clearData(pl)` clears data belonging to the plotter `pl` associated with a theater plot. This function clears data from plotters created by the following plotter methods:

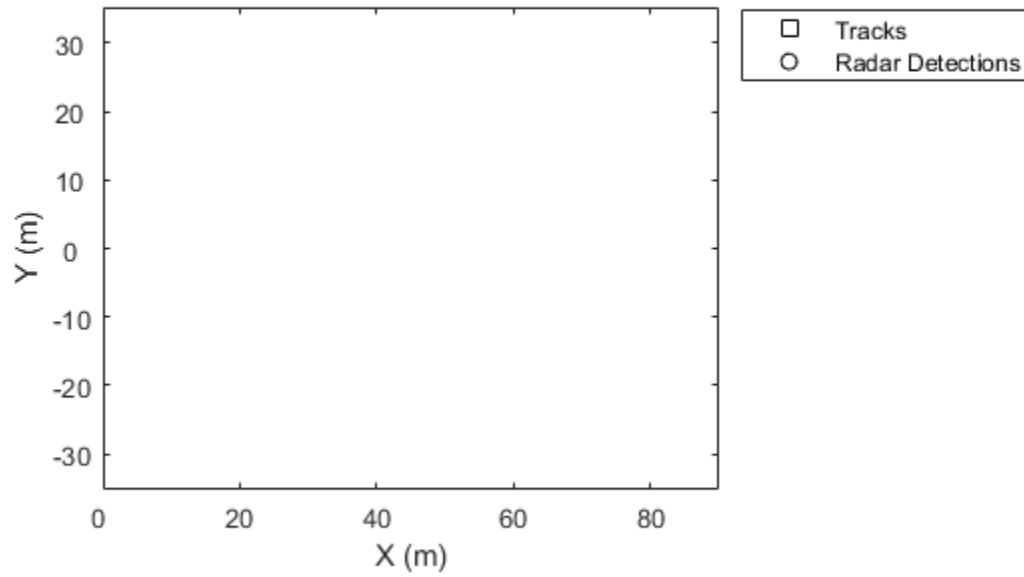
- `detectionPlotter`
- `orientationPlotter`
- `platformPlotter`
- `trackPlotter`
- `trajectoryPlotter`

## Examples

### Clear Specific Plotter Data

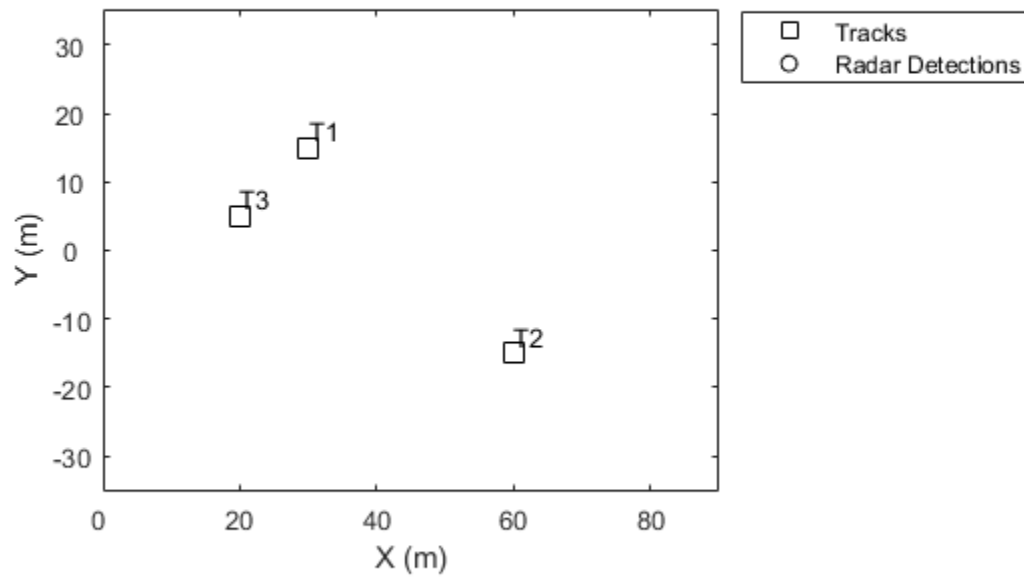
Create a theater plot. Add a track plotter and detection plotter to the theater plot.

```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35]);  
tPlotter = trackPlotter(tp,'DisplayName','Tracks');  
radarPlotter = detectionPlotter(tp,'DisplayName','Radar Detections');
```



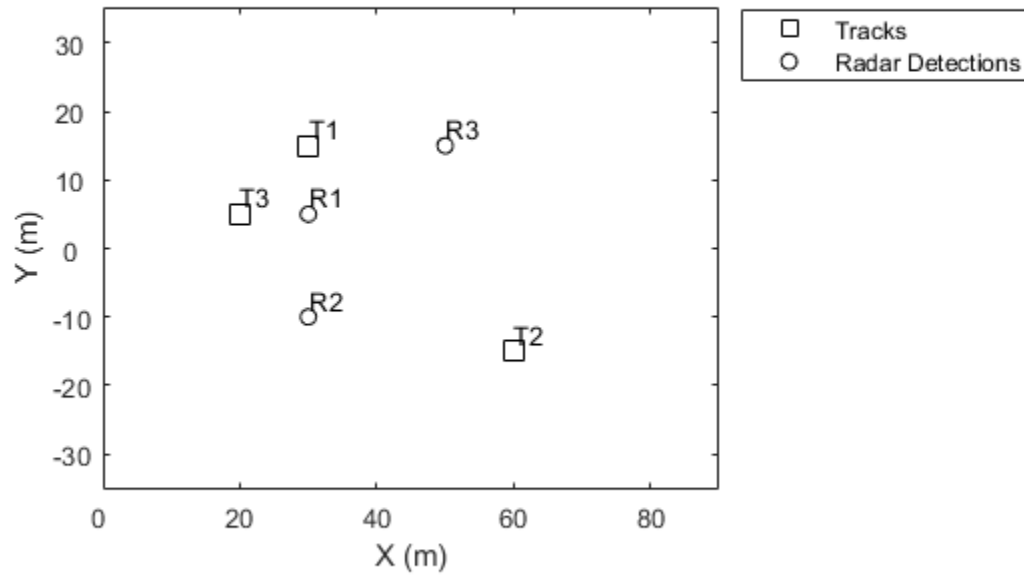
Plot a set of tracks in the track plotter.

```
trackPos = [30, 15, 1; 60, -15, 1; 20, 5, 1];  
trackLabels = {'T1', 'T2', 'T3'};  
plotTrack(tPlotter, trackPos, trackLabels)
```



Plot a set of detections in the detection plotter.

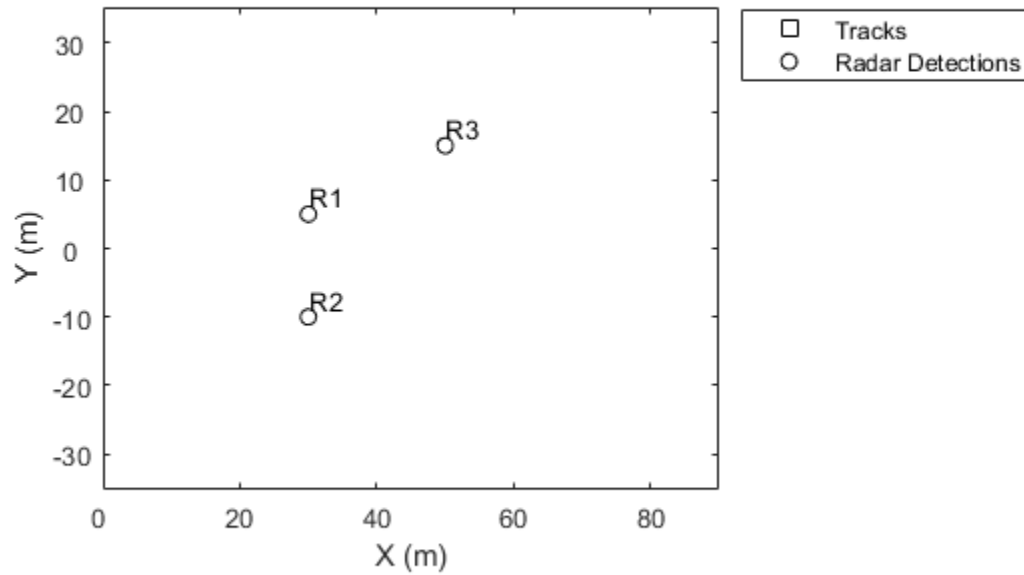
```
detPos = [30, 5, 4; 30, -10, 2; 50, 15, 1];  
detLabels = {'R1', 'R2', 'R3'};  
plotDetection(radarPlotter, detPos, detLabels)
```



Delete the track plotter data.

```
clearData(tPlotter)
```





## Input Arguments

### **p1** — Specific plotter belonging to theater plot

specific plotter of theater plot handle

Specific plotter belonging to a theater plot, specified as a plotter handle of theaterPlot.

## See Also

[clearPlotterData](#) | [theaterPlot](#) | [findPlotter](#)

**Introduced in R2021a**

## clearPlotterData

Clear plotter data from theater plot

### Syntax

```
clearPlotterData(tp)
```

### Description

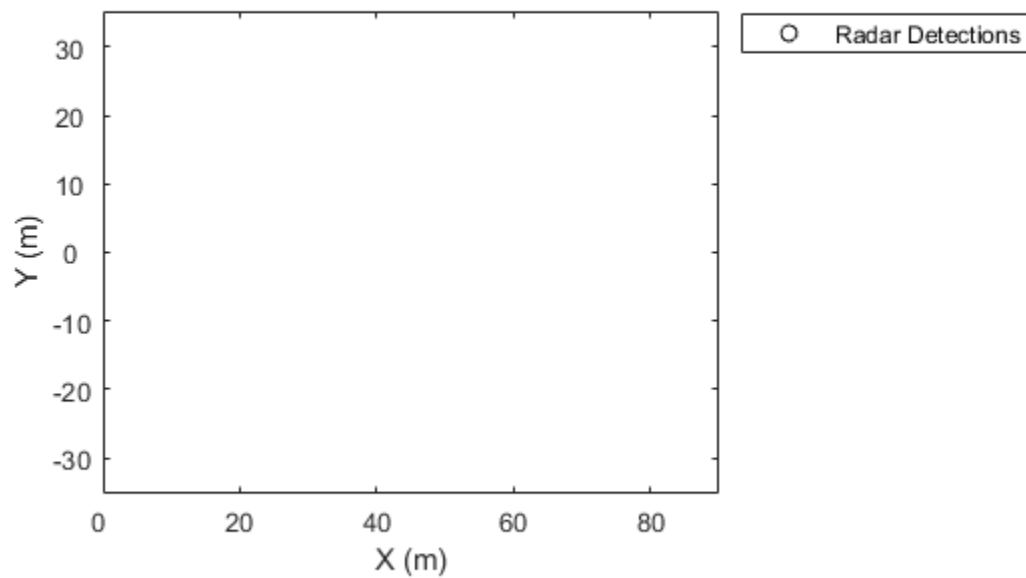
`clearPlotterData(tp)` clears data shown in the plot from all the plotters used in the theater plot, `tp`. Legend entries and coverage areas are not cleared from the plot.

### Examples

#### Clear Plotter Data from Theater Plot

Create a theater plot and a detection plotter.

```
tp = theaterPlot('XLim',[0, 90], 'YLim',[-35, 35], 'ZLim',[0, 10]);  
detectionPlotter(tp, 'DisplayName', 'Radar Detections');
```

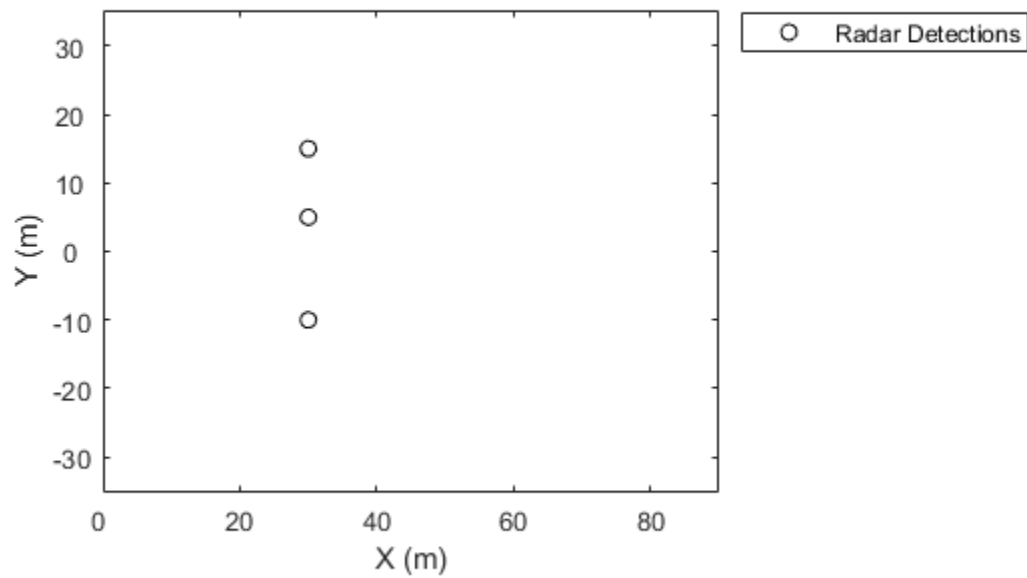


Use `findPlotter` to locate the plotter by its display name.

```
radarPlotter = findPlotter(tp, 'DisplayName', 'Radar Detections');
```

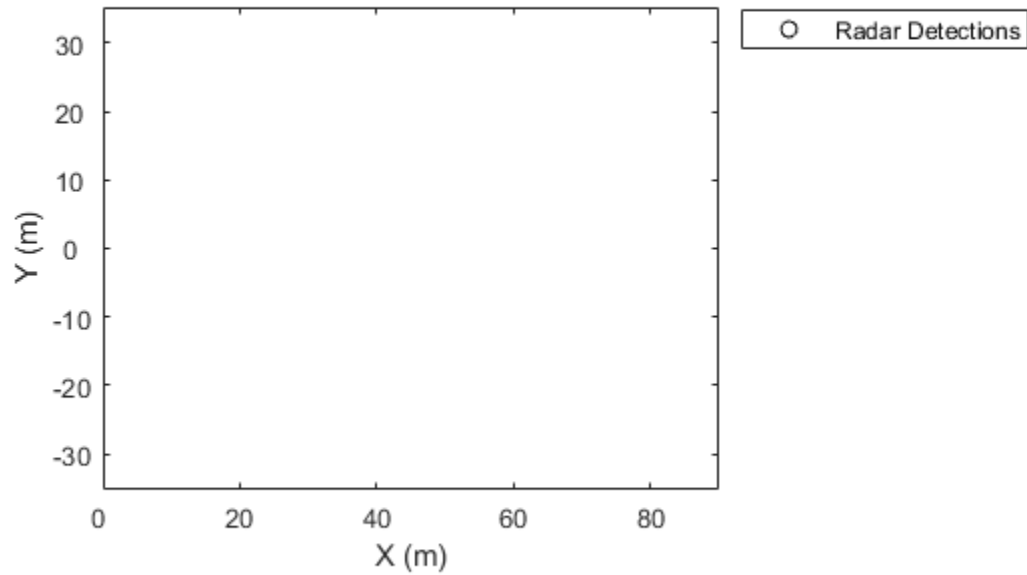
Plot three detections.

```
plotDetection(radarPlotter, [30, 5, 1; 30, -10, 2; 30, 15, 1]);
```



Clear data from the plot.

```
clearPlotterData(tp);
```



## Input Arguments

**tp** — Theater plot  
theaterPlot object

Theater plot, specified as a theaterPlot object.

## See Also

theaterPlot | findPlotter | clearData

**Introduced in R2021a**

# findPlotter

Return array of plotters associated with theater plot

## Syntax

```
p = findPlotter(tp)
p = findPlotter(tp,Name,Value)
```

## Description

`p = findPlotter(tp)` returns the array of plotters associated with the theater plot, `tp`.

---

**Note** In general, it is faster to use the plotters directly from the plotter creation methods of `theaterPlot`. Use `findPlotter` when it is otherwise inconvenient to use the plotter handles directly.

---

`p = findPlotter(tp,Name,Value)` specifies one or more `Name,Value` pair arguments required to match for the theater plot.

## Examples

### Find Plotter in Theater Plot

Create a theater plot and generate detection and platform plotters. Set the value of the `Tag` property of the detection plotter to `'radPlot'`.

```
tp = theaterPlot('XLim',[0, 90],'YLim',[-35, 35]);
detectionPlotter(tp,'DisplayName','Radar Detections','Tag','radPlot');
platformPlotter(tp,'DisplayName','Platforms');
```

Use `findPlotter` to locate the detection plotter based on its `Tag` property.

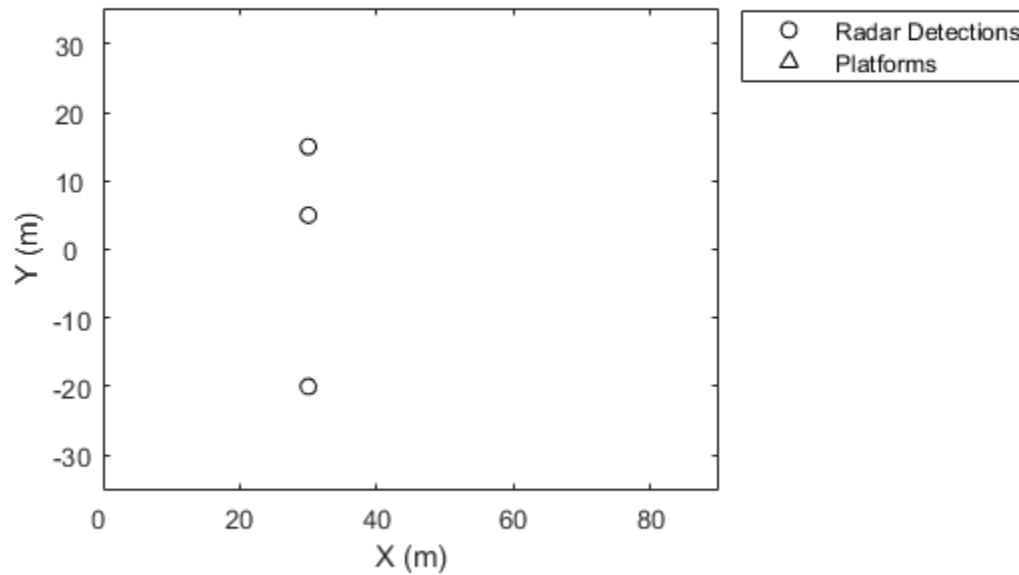
```
radarPlotter = findPlotter(tp,'Tag','radPlot')
```

```
radarPlotter =
  DetectionPlotter with properties:

    HistoryDepth: 0
      Marker: 'o'
    MarkerSize: 6
  MarkerEdgeColor: [0 0 0]
  MarkerFaceColor: 'none'
    FontSize: 10
    LabelOffset: [0 0 0]
  VelocityScaling: 1
      Tag: 'radPlot'
    DisplayName: 'Radar Detections'
```

Use the detection plotter to display the located objects.

```
plotDetection(radarPlotter, [30, 5, 0; 30, -20, 0; 30, 15, 0]);
```



## Input Arguments

### **tp** – Theater plot

theaterPlot object

Theater plot, specified as a theaterPlot object.

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'Tag', 'thisPlotter'

### **DisplayName** – Display name

character vector | string scalar

Display name of the plotter to find, specified as the comma-separated pair consisting of 'DisplayName' and a character vector or string scalar. `DisplayName` is the plotter name that appears in the legend. To match missing legend entries, specify `DisplayName` as ''.

**Tag — Tag of plotter**

character vector | string scalar

Tag of plotter to find, specified as the comma-separated pair consisting of 'Tag' a character vector or string scalar. By default, plotters have a `Tag` property with a default value of 'PlotterN', where *N* is an integer that corresponds to the *N*th plotter associated with the theater plot `tp`.

**See Also**

`theaterPlot` | `clearPlotterData` | `clearData`

**Introduced in R2021a**

## coveragePlotter

Create coverage plotter

### Syntax

```
cPlotter = coveragePlotter(tp)
cPlotter = coveragePlotter(tp,Name,Value)
```

### Description

`cPlotter = coveragePlotter(tp)` creates a `CoveragePlotter` object for use with the theater plot object, `tp`. Use the `plotCoverage` function to plot the sensor coverage via the created `CoveragePlotter` object.

`cPlotter = coveragePlotter(tp,Name,Value)` creates a `CoveragePlotter` object with additional options specified by one or more `Name,Value` pair arguments.

### Examples

#### Plot Coverage in Theater Plot

Create a theater plot and set the limits for its axes. Create a coverage plotter with `DisplayName` set to 'Sensor Coverage'.

```
tp = theaterPlot('XLim',[-40 40],'YLim',[-40 40],'ZLim',[-40 40]);
covp = coveragePlotter(tp,'DisplayName','Sensor Coverage');
```

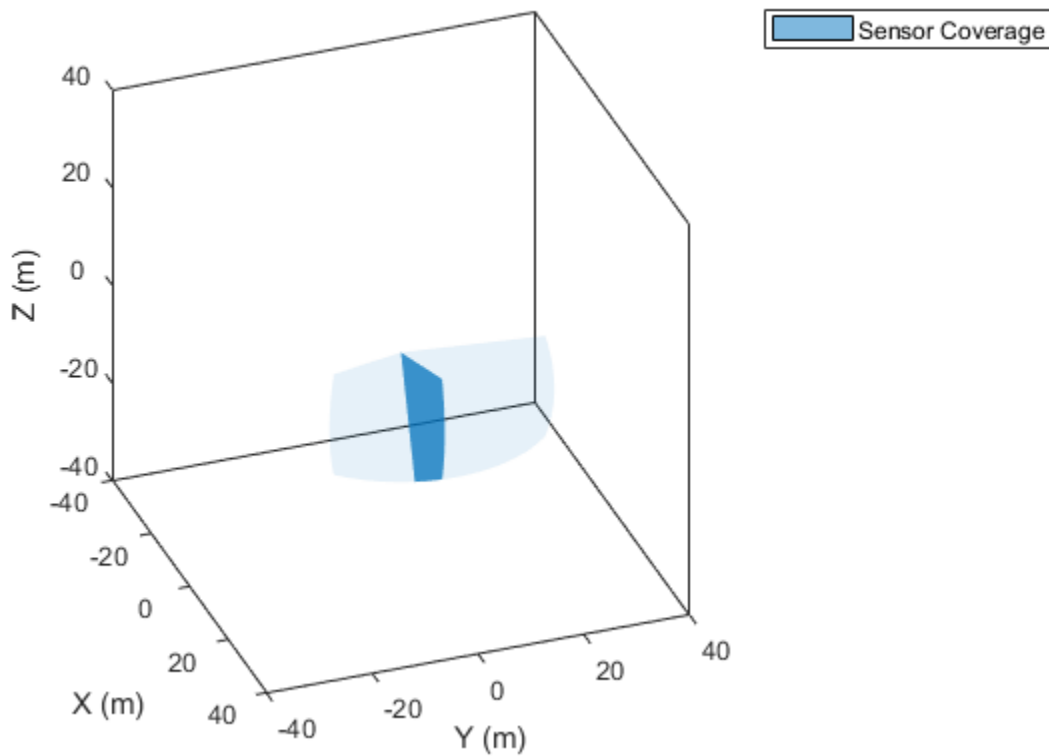
Set up the configuration of the sensors whose coverage is to be plotted.

```
sensor = struct('Index',1,'ScanLimits',[-45 45],'FieldOfView',[10;40],...
    'LookAngle',-10,'Range',30,'Position',zeros(1,3),'Orientation',zeros(1,3));
```

Plot the coverage using the `plotCoverage` function and visualize the results. The dark blue represents the current sensor beam, and the light blue represents the coverage area.

```
plotCoverage(covp,sensor)
view(70,30)
```





### Animate Sensor Coverage Plot

Create a theater plot and create a coverage plotter.

```
tp = theaterPlot('XLim',[-1e7 1e7],'YLim',[-1e7 1e7],'ZLim',[-2e6 1e6]);
covp = coveragePlotter(tp,'DisplayName','Sensor Coverage');
view(25,20)
```

Model a non-scanning radar and a raster scanning radar.

```
radarIndex = 1;
radar =fusionRadarSensor(radarIndex,'No Scanning','RangeLimits',[0 1e8]);
RasterIndex = 2;
raster = fusionRadarSensor(RasterIndex,'Raster','RangeLimits',[0 1e8]);
```

Create a target platform.

```
tgt = struct( ...
    'PlatformID', 1, ...
    'Position', [0 -50e3 -1e3], ...
    'Speed', -1e3);
```

Simulate sensors and visualize their scanning pattern.

```
time = 0;
timestep = 1;
```


```

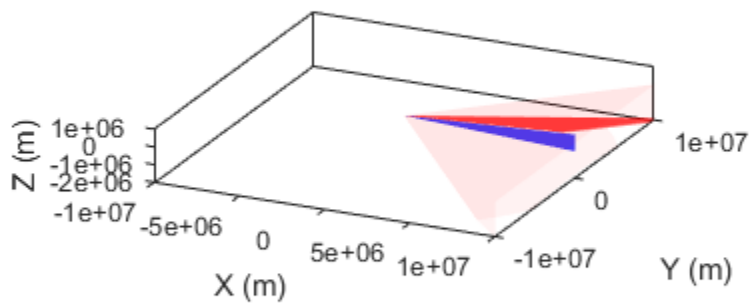
stopTime = 90;
while time < stopTime
    time = time+timestep;
    radar(tgt,time);
    raster(tgt,time);

    % Obtain sensor configuration using coverageConfig.
    radarcov = coverageConfig(radar);
    ircov = coverageConfig(raster);

    % Update plotter
    plotCoverage(covp,[radarcov,ircov],...
        [radarIndex, RasterIndex],...
        {'blue','red'}...
        );
    pause(0.03)
end

```

 Sensor Coverage



## Input Arguments

**tp** — Theater plot  
theaterPlot object

Theater plot, specified as a theaterPlot object.

## Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `'DisplayName', 'Radar1'`

### DisplayName — Plot name to display in legend

character vector | string scalar

Plot name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar. If no name is specified, no entry is shown.

Example: `'DisplayName', 'Radar Detections'`

### Color — Coverage area and sensor beam color

'auto' (default) | character vector | string scalar | RGB triplet | hexadecimal color code

Coverage area and sensor beam color, specified as a character vector, a string scalar, an RGB triplet, a hexadecimal color code, or `'auto'`. When a color is specified, the plotter draws all coverage areas and beams with the specified color. If the color is set to `'auto'`, the plotter uses the axis color order to assign colors to sensors based on their sensor indices.

### Alpha — Face alpha values of coverage area and sensor beam

[0.7 0.05] (default) | 2-element vector of nonnegative scalars

Face alpha values of the coverage area and the sensor beam, specified as a 2-element vector of nonnegative scalars. The first element is the value applied to the beam and the second element is the value applied to the coverage area.

### Tag — Tag associated with plotter

'PlotterN' (default) | character vector | string

Tag associated with the plotter, specified as a character vector or string. You can use the `findPlotter` function to identify plotters based on their tag. The default value is `'PlotterN'`, where `N` is an integer that corresponds to the `N`th plotter associated with the `theaterPlot`.

## Output Arguments

### cPlotter — Coverage plotter

CoveragePlotter object

Coverage plotter, returned as a `CoveragePlotter` object. You can modify this object by changing its property values. The property names correspond to the name-value pair arguments of the `coveragePlotter` function.

To plot the coverage, use the `plotCoverage` function.

## See Also

`plotCoverage` | `theaterPlot` | `clearData` | `clearPlotterData`

**Introduced in R2021a**

# plotCoverage

Plot set of coverages in theater coverage plotter

## Syntax

```
plotCoverage(cPlotter,configurations)
plotCoverage(cPlotter,configurations,indices,colors)
```

## Description

`plotCoverage(cPlotter,configurations)` specifies configurations of  $M$  sensors or emitters whose coverage areas and beams are plotted by the `CoveragePlotter` object, `cPlotter`. See `coveragePlotter` on how to create a `CoveragePlotter` object.

`plotCoverage(cPlotter,configurations,indices,colors)` specifies the color of each coverage and beam plot pair using a list of indices and colors.

## Examples

### Plot Coverage in Theater Plot

Create a theater plot and set the limits for its axes. Create a coverage plotter with `DisplayName` set to 'Sensor Coverage'.

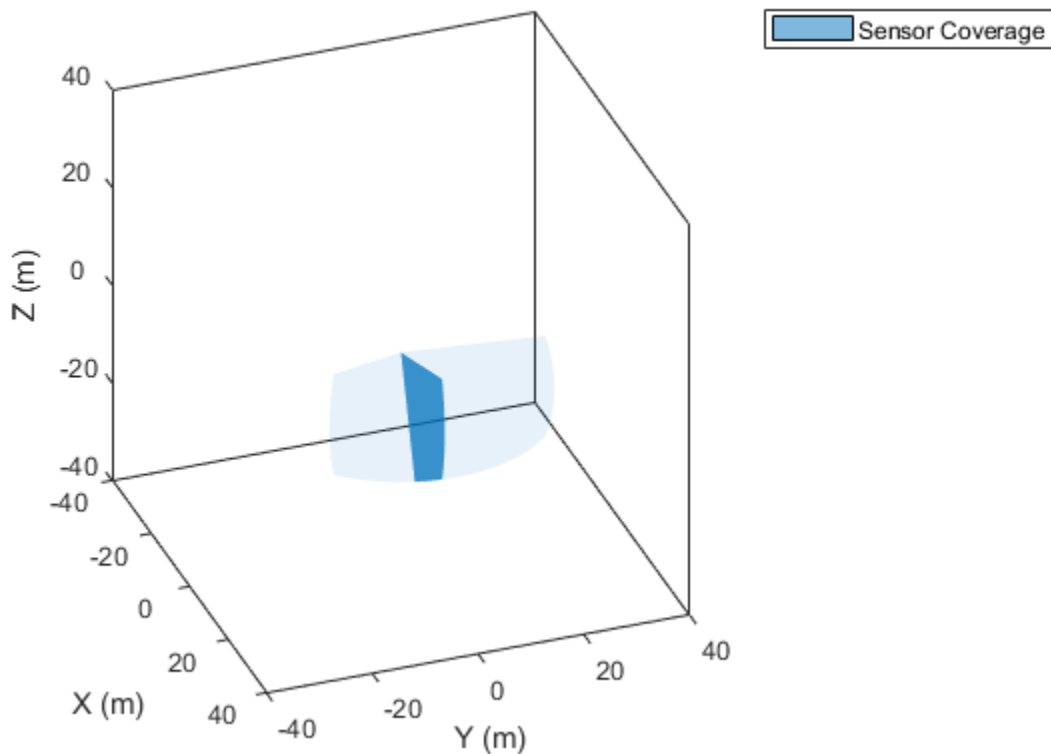
```
tp = theaterPlot('XLim',[-40 40],'YLim',[-40 40],'ZLim',[-40 40]);
covp = coveragePlotter(tp,'DisplayName','Sensor Coverage');
```

Set up the configuration of the sensors whose coverage is to be plotted.

```
sensor = struct('Index',1,'ScanLimits',[-45 45],'FieldOfView',[10;40],...
    'LookAngle',-10,'Range',30,'Position',zeros(1,3),'Orientation',zeros(1,3));
```

Plot the coverage using the `plotCoverage` function and visualize the results. The dark blue represents the current sensor beam, and the light blue represents the coverage area.

```
plotCoverage(covp,sensor)
view(70,30)
```



### Animate Sensor Coverage Plot

Create a theater plot and create a coverage plotter.

```
tp = theaterPlot('XLim',[-1e7 1e7],'YLim',[-1e7 1e7],'ZLim',[-2e6 1e6]);
covp = coveragePlotter(tp,'DisplayName','Sensor Coverage');
view(25,20)
```

Model a non-scanning radar and a raster scanning radar.

```
radarIndex = 1;
radar =fusionRadarSensor(radarIndex,'No Scanning','RangeLimits',[0 1e8]);
RasterIndex = 2;
raster = fusionRadarSensor(RasterIndex,'Raster','RangeLimits',[0 1e8]);
```

Create a target platform.

```
tgt = struct( ...
    'PlatformID', 1, ...
    'Position', [0 -50e3 -1e3], ...
    'Speed', -1e3);
```

Simulate sensors and visualize their scanning pattern.

```
time = 0;
timestep = 1;
```

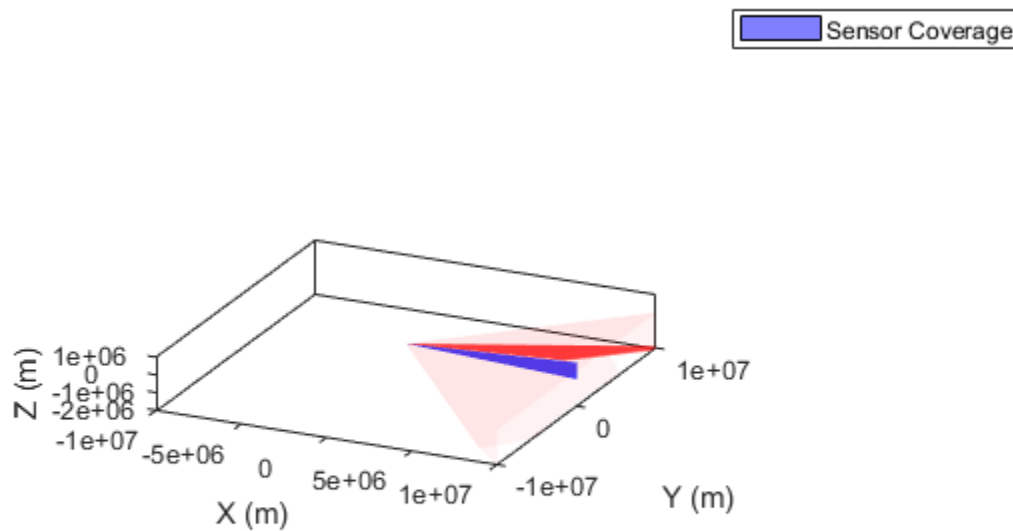
```

stopTime = 90;
while time < stopTime
    time = time+timestep;
    radar(tgt,time);
    raster(tgt,time);

    % Obtain sensor configuration using coverageConfig.
    radarcov = coverageConfig(radar);
    ircov = coverageConfig(raster);

    % Update plotter
    plotCoverage(covp,[radarcov,ircov],...
        [radarIndex, RasterIndex],...
        {'blue','red'}...
        );
    pause(0.03)
end

```



## Input Arguments

### cPlotter — Coverage plotter object

CoveragePlotter object

Coverage plotter object, created by the coveragePlotter function.

**configurations — Sensor or emitter configurations**

array of structures

Sensor or emitter configurations, specified as an array of structures. Each structure corresponds to the configuration of a sensor or emitter. The fields of each structure are:

**Fields of configurations**

Field	Description
Index	A unique integer to distinguish sensors or emitters.
LookAngle	The current boresight angles of the sensor or emitter, specified as: <ul style="list-style-type: none"> <li>• A scalar in degrees if scanning only in the azimuth direction.</li> <li>• A two-element vector [azimuth; elevation] in degrees if scanning both in the azimuth and elevation directions.</li> </ul>
FieldOfView	The field of view of the sensor or emitter, specified as a two-element vector [azimuth; elevation] in degrees.
ScanLimits	The minimum and maximum angles the sensor or emitter can scan from its Orientation. <ul style="list-style-type: none"> <li>• If the sensor or emitter can only scan in the azimuth direction, specify the limits as a 1-by-2 row vector [minAz, maxAz] in degrees.</li> <li>• If the sensor or emitter can also scan in the elevation direction, specify the limits as a 2-by-2 matrix [minAz, maxAz; minEl, maxEl] in degrees.</li> </ul>
Range	The range of the beam and coverage area of the sensor or emitter in meters.
Position	The origin position of the sensor or emitter, specified as a three-element vector [X, Y, Z] on the theater plot's axes.
Orientation	The rotation transformation from the scenario or global frame to the sensor or emitter mounting frame, specified as a rotation matrix, a quaternion, or three Euler angles in ZYX sequence.

**Tip** If either the value of Position field or the value of the Orientation field is NaN, the corresponding coverage area and beam will not be plotted.

**indices — Sensor or emitter indices***N*-element array of nonnegative integers



Sensor or emitter indices, specified as an  $N$ -element array of nonnegative integers. This argument allows you to specify the color of each coverage area and beam pair with the corresponding index.

Example: [1;2;4]

**colors – Coverage plotter colors**

$N$ -element array of character vector |  $N$ -element array of string scalar |  $N$ -element array of RGB triplet |  $N$ -element array of hexadecimal color code

Coverage plotter colors, specified as an  $N$ -element vector of character vectors, string scalars, RGB triplets, or hexadecimal color codes.  $N$  is the number of elements in the `indices` array. The coverage area and beam pair indexed by the  $i$ th element in the `indices` array is plotted with the color specified by the  $i$ th element of the `colors` array.

**See Also**

`coveragePlotter` | `theaterPlot` | `clearData` | `clearPlotterData`

**Introduced in R2021a**

## detectionPlotter

Create detection plotter

### Syntax

```
detPlotter = detectionPlotter(tp)
detPlotter = detectionPlotter(tp,Name,Value)
```

### Description

`detPlotter = detectionPlotter(tp)` creates a detection plotter for use with the theater plot `tp`.

`detPlotter = detectionPlotter(tp,Name,Value)` creates a detection plotter with additional options specified by one or more `Name, Value` pair arguments.

### Examples

#### Create and Update Detections for Theater Plot

Create a theater plot.

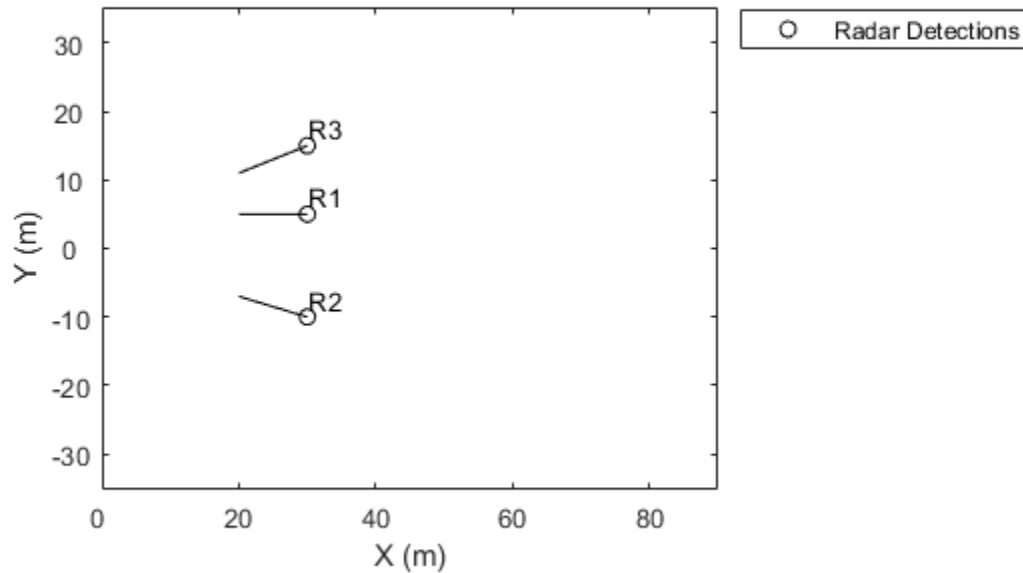
```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35],'ZLim',[1,10]);
```

Create a detection plotter with the name Radar Detections.

```
radarPlotter = detectionPlotter(tp,'DisplayName','Radar Detections');
```

Update the detection plotter with three detections labeled 'R1', 'R2', and 'R3' positioned in units of meters at (30, 5, 4), (30, -10, 2), and (30, 15, 1) with corresponding velocities (in m/s) of (-10, 0, 2), (-10, 3, 1), and (-10, -4, 1), respectively.

```
positions = [30, 5, 4; 30, -10, 2; 30, 15, 1];
velocities = [-10, 0, 2; -10, 3, 1; -10, -4, 1];
labels = {'R1','R2','R3'};
plotDetection(radarPlotter, positions, velocities, labels)
```



## Input Arguments

### tp — Theater plot

theaterPlot object

Theater plot, specified as a theaterPlot object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'MarkerSize',10`

### DisplayName — Plot name to display in legend

character vector | string scalar

Plot name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar. If no name is specified, no entry is shown.

Example: `'DisplayName','Radar Detections'`

**HistoryDepth — Number of previous updates to display**

0 (default) | nonnegative integer less than or equal to 10,000

Number of previous track updates to display, specified as the comma-separated pair consisting of 'HistoryDepth' and a nonnegative integer less than or equal to 10,000. If set to 0, then no previous updates are rendered.

**Marker — Marker symbol**

'o' (default) | character vector | string scalar

Marker symbol, specified as the comma-separated pair consisting of 'Marker' and one of these symbols.

Marker	Description	Resulting Marker
'o'	Circle	○
'+'	Plus sign	+
'*'	Asterisk	*
'.'	Point	•
'x'	Cross	×
'_'	Horizontal line	—
' '	Vertical line	
's'	Square	□
'd'	Diamond	◇
'^'	Upward-pointing triangle	△
'v'	Downward-pointing triangle	▽
'>'	Right-pointing triangle	▷
'<'	Left-pointing triangle	◁
'p'	Pentagram	☆
'h'	Hexagram	☆
'none'	No markers	Not applicable

**MarkerSize — Size of marker**

6 (default) | positive integer

Size of marker, specified as the comma-separated pair consisting of 'MarkerSize' and a positive integer in points.

**MarkerEdgeColor — Marker outline color**

'black' (default) | character vector | string scalar | RGB triplet | hexadecimal color code

Marker outline color, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a character vector, a string scalar, an RGB triplet, or a hexadecimal color code.

**MarkerFaceColor — Marker fill color**

'none' (default) | character vector | string scalar | RGB triplet | hexadecimal color code

Marker outline color, specified as the comma-separated pair consisting of 'MarkerFaceColor' and a character vector, a string scalar, an RGB triplet, a hexadecimal color code, or 'none'. The default is 'none'.

**FontSize — Font size for labeling platforms**

10 (default) | positive integer

Font size for labeling detections, specified as the comma-separated pair consisting of 'FontSize' and a positive integer that represents font point size.

**LabelOffset — Gap between label and positional point**

[0 0 0] (default) | three-element row vector

Gap between label and positional point it annotates, specified as the comma-separated pair consisting of 'LabelOffset' and a three-element row vector. Specify the [x y z] offset in meters.

**VelocityScaling — Scale factor for magnitude length of velocity vectors**

1 (default) | positive scalar

Scale factor for magnitude length of velocity vectors, specified as the comma-separated pair consisting of 'VelocityScaling' and a positive scalar. The plot renders the magnitude vector value as  $VK$ , where  $V$  is the magnitude of the velocity in meters per second, and  $K$  is the value of `VelocityScaling`.

**Tag — Tag to associate with the plotter**

'PlotterN' (default) | character vector | string scalar

Tag to associate with the plotter, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. The default value is 'PlotterN', where  $N$  is an integer that corresponds to the  $N$ th plotter associated with the `theaterPlot`.

Tags provide a way to identify plotter objects, for example when searching using `findPlotter`.

**See Also**

`theaterPlot` | `plotDetection` | `clearData` | `clearPlotterData`

**Introduced in R2021a**

## plotDetection

Plot set of detections in theater detection plotter

### Syntax

```
plotDetection(detPlotter,positions)
plotDetection(detPlotter,positions,velocities)
plotDetection(detPlotter,positions,___,labels)
plotDetection(detPlotter,positions,___,covariances)
```

### Description

`plotDetection(detPlotter,positions)` specifies positions of  $M$  detected objects whose positions are plotted by the detection plotter `detPlotter`. Specify the positions as an  $M$ -by-3 matrix, where each column of the matrix corresponds to the  $x$ -,  $y$ -, and  $z$ -coordinates of the detected object locations.

`plotDetection(detPlotter,positions,velocities)` also specifies the corresponding velocities of the detections. Velocities are plotted as line vectors emanating from the center positions of the detections. If specified, `velocities` must have the same dimensions as `positions`.

`plotDetection(detPlotter,positions,___,labels)` also specifies a cell vector of length  $M$  whose elements contain the text labels corresponding to the  $M$  detections specified in the positions matrix. If omitted, no labels are plotted.

`plotDetection(detPlotter,positions,___,covariances)` also specifies the covariances of the  $M$  detection uncertainties, where the covariances are a 3-by-3-by- $M$  matrix of covariances that are centered at the positions of each detection. The uncertainties are plotted as an ellipsoid

### Examples

#### Create and Update Detections for Theater Plot

Create a theater plot.

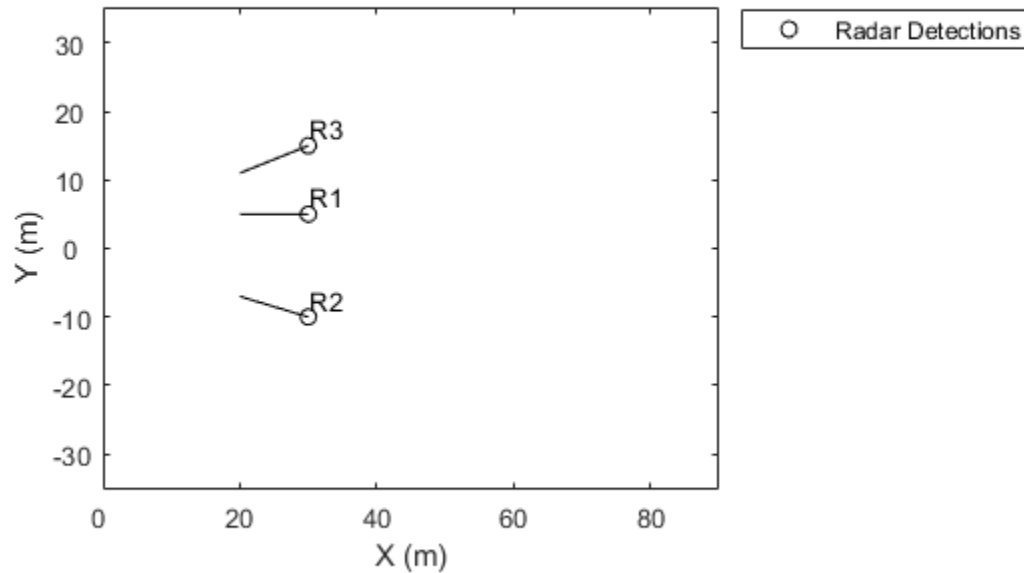
```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35],'ZLim',[1,10]);
```

Create a detection plotter with the name Radar Detections.

```
radarPlotter = detectionPlotter(tp,'DisplayName','Radar Detections');
```

Update the detection plotter with three detections labeled 'R1', 'R2', and 'R3' positioned in units of meters at (30, 5, 4), (30, -10, 2), and (30, 15, 1) with corresponding velocities (in m/s) of (-10, 0, 2), (-10, 3, 1), and (-10, -4, 1), respectively.

```
positions = [30, 5, 4; 30, -10, 2; 30, 15, 1];
velocities = [-10, 0, 2; -10, 3, 1; -10, -4, 1];
labels = {'R1','R2','R3'};
plotDetection(radarPlotter, positions, velocities, labels)
```



## Input Arguments

### **detPlotter** — Detection plotter

detectionPlotter object

Detection plotter, specified as a detectionPlotter object.

### **positions** — Detection positions

real-valued matrix

Detection positions, specified as an  $M$ -by-3 real-valued matrix, where  $M$  is the number of detections. Each column of the matrix corresponds to the  $x$ -,  $y$ -, and  $z$ -coordinates of the detection positions in meters.

### **velocities** — Detection velocities

real-valued matrix

Detection velocities, specified as an  $M$ -by-3 real-valued matrix, where  $M$  is the number of detections. Each column of the matrix corresponds to the  $x$ -,  $y$ -, and  $z$ -velocities of the detections. If specified, **velocities** must have the same dimensions as **positions**.

### **labels** — Detection labels

cell array

Detection labels, specified as a  $M$ -by-1 cell array of character vectors, where  $M$  is the number of detections. The input argument `labels` contains the text labels corresponding to the  $M$  detections specified in `positions`. If `labels` is omitted, no labels are plotted.

**covariances – Detection uncertainties**

real-valued array

Detection uncertainties of  $M$  tracked objects, specified as a 3-by-3-by- $M$  real-valued array of covariances. The covariances are centered at the positions of each detection and are plotted as an ellipsoid.

**See Also**

`theaterPlot` | `detectionPlotter` | `clearData` | `clearPlotterData`

**Introduced in R2021a**



# orientationPlotter

Create orientation plotter

## Syntax

```
oPlotter = orientationPlotter(tp)
oPlotter = orientationPlotter(tp,Name,Value)
```

## Description

`oPlotter = orientationPlotter(tp)` creates an orientation plotter for use with the theater plot `tp`.

`oPlotter = orientationPlotter(tp,Name,Value)` creates an orientation plotter with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Show Random Orientation

Create a theater plot object and a trajectory plotter.

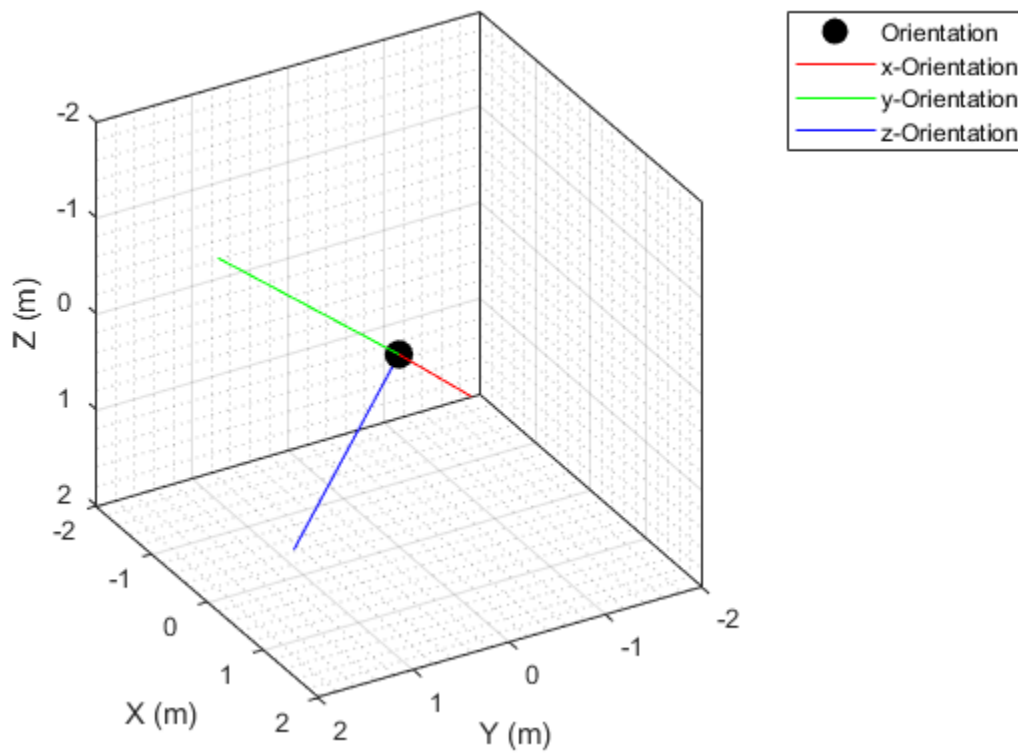
```
tp = theaterPlot('XLimit',[-2 2],'YLimit',[-2 2],'ZLimit',[-2 2]);
op = orientationPlotter(tp,'DisplayName','Orientation',...
    'LocalAxesLength',2);
```

Create some random rotations.

```
pose = randrot(20,1);
```

Loop through the pose information to animate the orientations.

```
for i=1:numel(pose)
    plotOrientation(op,pose(i))
    drawnow
end
```



## Input Arguments

### tp — Theater plot

theaterPlot object

Theater plot, specified as a theaterPlot object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'HistoryDepth',6

### DisplayName — Plot name to display in legend

character vector | string scalar

Plot name to display in legend, specified as the comma-separated pair consisting of 'DisplayName' and a character vector or string scalar. If no name is specified, no entry is shown.

Example: 'DisplayName','Radar Detections'

**HistoryDepth — Number of previous track updates to display**

0 (default) | nonnegative integer less than or equal to 100

Number of previous track updates to display, specified as the comma-separated pair consisting of 'HistoryDepth' and a nonnegative integer less than or equal to 100. If set to 0, then no previous updates are rendered.

**Marker — Marker symbol**

'o' (default) | character vector | string scalar

Marker symbol, specified as the comma-separated pair consisting of 'Marker' and one of these symbols.

Marker	Description	Resulting Marker
'o'	Circle	○
'+'	Plus sign	+
'*'	Asterisk	*
'.'	Point	•
'x'	Cross	×
'_'	Horizontal line	—
' '	Vertical line	
's'	Square	□
'd'	Diamond	◇
'^'	Upward-pointing triangle	△
'v'	Downward-pointing triangle	▽
'>'	Right-pointing triangle	▷
'<'	Left-pointing triangle	◁
'p'	Pentagram	☆
'h'	Hexagram	☆
'none'	No markers	Not applicable

**MarkerSize — Size of marker**

10 (default) | positive integer

Size of marker, specified in points as the comma-separated pair consisting of 'MarkerSize' and a positive integer.

**MarkerEdgeColor — Marker outline color**

'black' (default) | character vector | string scalar | RGB triplet | hexadecimal color code

Marker outline color, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a character vector, string scalar, an RGB triplet, or a hexadecimal color code. The default color is 'black'.

**MarkerFaceColor — Marker fill color**

'none' (default) | character vector | string scalar | RGB triplet | hexadecimal color code

Marker outline color, specified as the comma-separated pair consisting of 'MarkerFaceColor' and a character vector, a string scalar, an RGB triplet, a hexadecimal color code, or 'none'. The default is 'none'.

**FontSize — Font size for labeling tracks**

10 (default) | positive integer

Font size for labeling tracks, specified as the comma-separated pair consisting of 'FontSize' and a positive integer that represents font point size.

**LabelOffset — Gap between label and positional point**

[0 0 0] (default) | three-element row vector

Gap between label and positional point it annotates, specified as the comma-separated pair consisting of 'LabelOffset' and a three-element row vector. Specify the [x y z] offset in meters.

**LocalAxesLength — Length of line**

1 (default) | positive scalar

Length of line used to denote each of the local x-, y-, and z-axes of the given orientation, specified as the comma-separated pair consisting of 'LocalAxesLength' and a positive scalar. 'LocalAxesLength' is in meters.

**Tag — Tag to associate with the plotter**

'PlotterN' (default) | character vector | string scalar

Tag to associate with the plotter, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. The default value is 'PlotterN', where *N* is an integer that corresponds to the *N*th plotter associated with the theaterPlot.

Tags provide a way to identify plotter objects, for example when searching using findPlotter.

**See Also**

theaterPlot | plotOrientation | clearData | clearPlotterData

**Introduced in R2021a**

# plotOrientation

Plot set of orientations in orientation plotter

## Syntax

```
plotOrientation(oPlotter,orientations)
plotOrientation(oPlotter,roll,pitch,yaw)
plotOrientation(oPlotter, __ ,positions)
plotOrientation(oPlotter, __ ,positions,labels)
```

## Description

`plotOrientation(oPlotter,orientations)` specifies the orientations of  $M$  objects to show for the orientation plotter, `oPlotter`. The `orientations` argument can be either an  $M$ -by-1 array of quaternions, or a 3-by-3-by- $M$  array of rotation matrices.

`plotOrientation(oPlotter,roll,pitch,yaw)` specifies the orientations of  $M$  objects to show for the orientation plotter, `oPlotter`. The arguments `roll`, `pitch`, and `yaw` are  $M$ -by-1 vectors measured in degrees.

`plotOrientation(oPlotter, __ ,positions)` also specifies the positions of the objects as an  $M$ -by-3 matrix. Each column of `positions` corresponds to the  $x$ -,  $y$ -, and  $z$ -coordinates of the object locations, respectively.

`plotOrientation(oPlotter, __ ,positions,labels)` also specifies the labels as an  $M$ -by-1 cell array of character vectors that correspond to the  $M$  orientations.

## Examples

### Show Random Orientation

Create a theater plot object and a trajectory plotter.

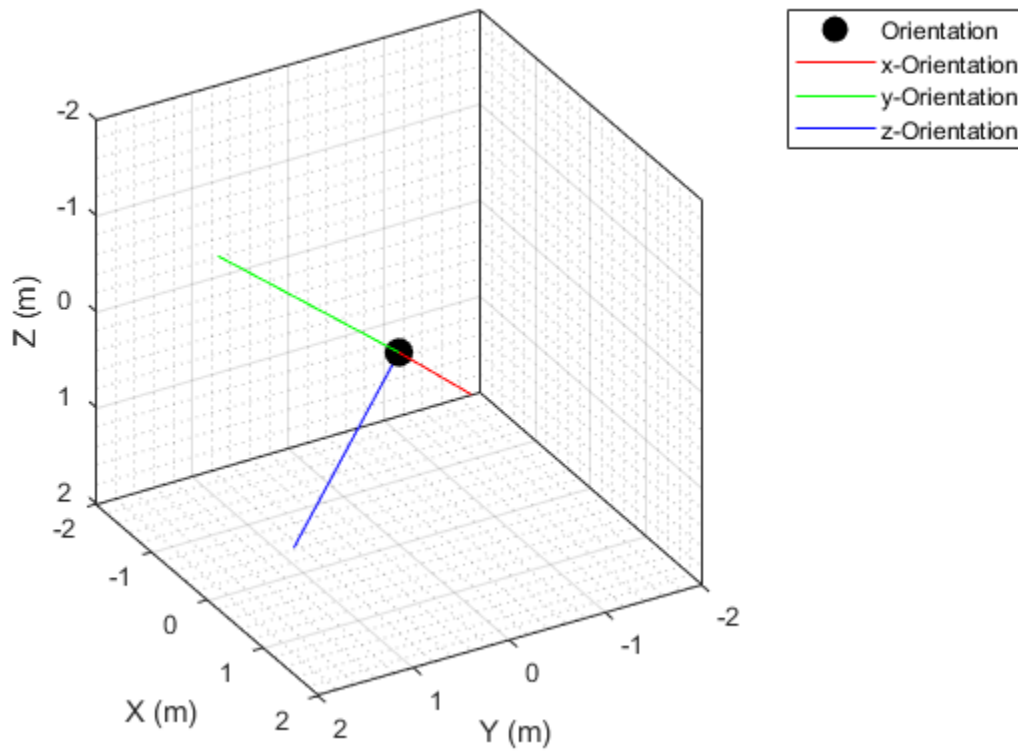
```
tp = theaterPlot('XLimit',[-2 2],'YLimit',[-2 2],'ZLimit',[-2 2]);
op = orientationPlotter(tp,'DisplayName','Orientation',...
    'LocalAxesLength',2);
```

Create some random rotations.

```
pose = randrot(20,1);
```

Loop through the pose information to animate the orientations.

```
for i=1:numel(pose)
    plotOrientation(op,pose(i))
    drawnow
end
```



## Input Arguments

### **oPlotter** – Orientation plotter

`orientationPlotter` object

Orientation plotter, specified as an `orientationPlotter` object.

### **orientations** – Orientations

quaternion array | real-valued array

Orientations of  $M$  objects, specified as either an  $M$ -by-1 array of quaternions, or a 3-by-3-by- $M$  array of rotation matrices.

### **roll, pitch, yaw** – Roll, pitch, yaw

real-valued vectors

Roll, pitch, and yaw angles defining the orientations of  $M$  objects, specified as  $M$ -by-1 vectors. Angles are measured in degrees.

### **positions** – Object positions

`[0 0 0]` (default) | real-valued matrix

Object positions, specified as an  $M$ -by-3 real-valued matrix, where  $M$  is the number of objects. Each column of the matrix corresponds to the  $x$ -,  $y$ -, and  $z$ -coordinates of the objects locations in meters. The default value of `positions` is at the origin.

**labels – Object labels**

cell array

Object labels, specified as a  $M$ -by-1 cell array of character vectors, where  $M$  is the number of objects. `labels` contains the text labels corresponding to the  $M$  objects specified in `positions`. If `labels` is omitted, no labels are plotted.

**See Also**

theaterPlot | orientationPlotter | clearData | clearPlotterData

**Introduced in R2021a**

## platformPlotter

Create platform plotter

### Syntax

```
pPlotter = platformPlotter(tp)
pPlotter = platformPlotter(tp,Name,Value)
```

### Description

`pPlotter = platformPlotter(tp)` creates a platform plotter for use with the theater plot, `tp`.

`pPlotter = platformPlotter(tp,Name,Value)` creates a platform plotter with additional options specified by one or more `Name, Value` pair arguments.

### Examples

#### Create and Update Theater Plot Platforms

Create a theater plot.

```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35],'ZLim',[1,10]);
```

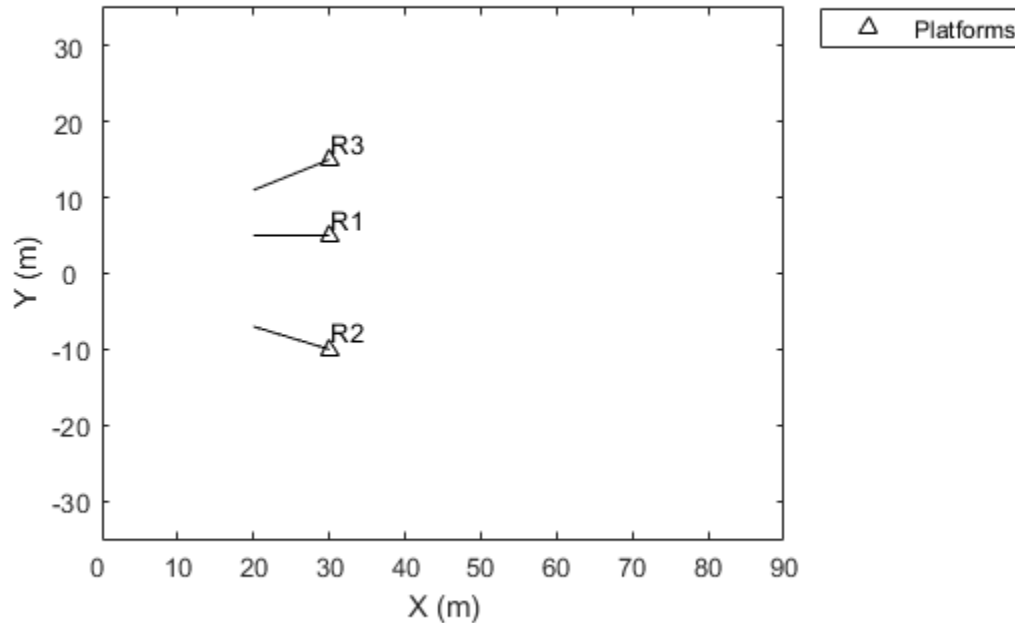
Create a platform plotter with the name 'Platforms'.

```
plotter = platformPlotter(tp,'DisplayName','Platforms');
```

Update the theater plot with three platforms labeled 'R1', 'R2', and 'R3'. Position the three platforms, in units of meters, at (30, 5, 4), (30, -10, 2), and (30, 15, 1), with corresponding velocities (in m/s) of (-10, 0, 2), (-10, 3, 1), and (-10, -4, 1), respectively.

```
positions = [30, 5, 4; 30, -10, 2; 30, 15, 1];
velocities = [-10, 0, 2; -10, 3, 1; -10, -4, 1];
labels = {'R1','R2','R3'};
plotPlatform(plotter, positions, velocities, labels);
```





## Input Arguments

### tp — Theater plot

theaterPlot object

Theater plot, specified as a theaterPlot object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'MarkerSize',10`

### DisplayName — Plot name to display in legend

character vector | string scalar

Plot name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar. If no name is specified, no entry is shown.

Example: `'DisplayName','Radar Detections'`

**Marker — Marker symbol**

'^' (default) | character vector | string scalar

Marker symbol, specified as the comma-separated pair consisting of 'Marker' and one of these values.

Marker	Description	Resulting Marker
'o'	Circle	○
'+'	Plus sign	+
'*'	Asterisk	*
'.'	Point	•
'x'	Cross	×
'_'	Horizontal line	—
' '	Vertical line	
's'	Square	□
'd'	Diamond	◇
'^'	Upward-pointing triangle	△
'v'	Downward-pointing triangle	▽
'>'	Right-pointing triangle	▷
'<'	Left-pointing triangle	◁
'p'	Pentagram	☆
'h'	Hexagram	☆
'none'	No markers	Not applicable

**MarkerSize — Size of marker**

6 | positive integer

Size of marker, specified as the comma-separated pair consisting of 'MarkerSize' and a positive integer in points.

**MarkerEdgeColor — Marker outline color**

'black' (default) | character vector | string scalar | RGB triplet | hexadecimal color code

Marker outline color, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a character vector, a string scalar, an RGB triplet, or a hexadecimal color code.

**MarkerFaceColor — Marker fill color**

'none' (default) | character vector | string scalar | RGB triplet | hexadecimal color code

Marker outline color, specified as the comma-separated pair consisting of 'MarkerFaceColor' and a character vector, a string scalar, an RGB triplet, a hexadecimal color code, or 'none'. The default is 'none'.

**FontSize — Font size for labeling platforms**

10 (default) | positive integer

Font size for labeling platforms, specified in font points size as the comma-separated pair consisting of 'FontSize' and a positive integer.

**LabelOffset — Gap between label and positional point**

[0 0 0] (default) | three-element row vector

Gap between label and positional point it annotates, specified as the comma-separated pair consisting of 'LabelOffset' and a three-element row vector. Specify the [x y z] offset in meters.

**VelocityScaling — Scale factor for magnitude length of velocity vectors**

1 (default) | positive scalar

Scale factor for magnitude length of velocity vectors, specified as the comma-separated pair consisting of 'VelocityScaling' and a positive scalar. The plot renders the magnitude vector value as  $VK$ , where  $V$  is the magnitude of the velocity in meters per second, and  $K$  is the value of `VelocityScaling`.

**Tag — Tag to associate with the plotter**

'PlotterN' (default) | character vector | string scalar

Tag to associate with the plotter, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. The default value is 'PlotterN', where  $N$  is an integer that corresponds to the  $N$ th plotter associated with the `theaterPlot`.

Tags provide a way to identify plotter objects, for example when searching using `findPlotter`.

**See Also**

`theaterPlot` | `plotPlatform` | `clearData` | `clearPlotterData`

**Introduced in R2021a**

## plotPlatform

Plot set of platforms in platform plotter

### Syntax

```
plotPlatform(platPlotter,positions)
plotPlatform(platPlotter,positions,velocities)
plotPlatform(platPlotter,positions,labels)
plotPlatform(platPlotter,positions,velocities,labels)
plotPlatform(platPlotter,positions, __,dimensions,orientations)
plotPlatform(platPlotter,positions, __,meshes,orientations)
```

### Description

`plotPlatform(platPlotter,positions)` specifies positions of  $M$  platforms whose positions are plotted by `platPlotter`. Specify the positions as an  $M$ -by-3 matrix, where each column of the matrix corresponds to the  $x$ -,  $y$ -, and  $z$ -coordinates of the platform locations.

`plotPlatform(platPlotter,positions,velocities)` also specifies the corresponding velocities of the platforms. Velocities are plotted as line vectors emanating from the positions of the platforms. If specified, velocities must have the same dimensions as positions.

`plotPlatform(platPlotter,positions,labels)` also specifies a cell vector of length  $M$  whose elements contain the text labels corresponding to the  $M$  platforms specified in the positions matrix. If omitted, no labels are plotted.

`plotPlatform(platPlotter,positions,velocities,labels)` specifies velocities and text labels corresponding to the  $M$  platforms specified in the positions matrix.

`plotPlatform(platPlotter,positions, __,dimensions,orientations)` specifies the dimension and orientation of each plotted platform.

`plotPlatform(platPlotter,positions, __,meshes,orientations)` specifies the extent of each platform using meshes.

Use of meshes requires Sensor Fusion and Tracking Toolbox.

### Examples

#### Create and Update Theater Plot Platforms

Create a theater plot.

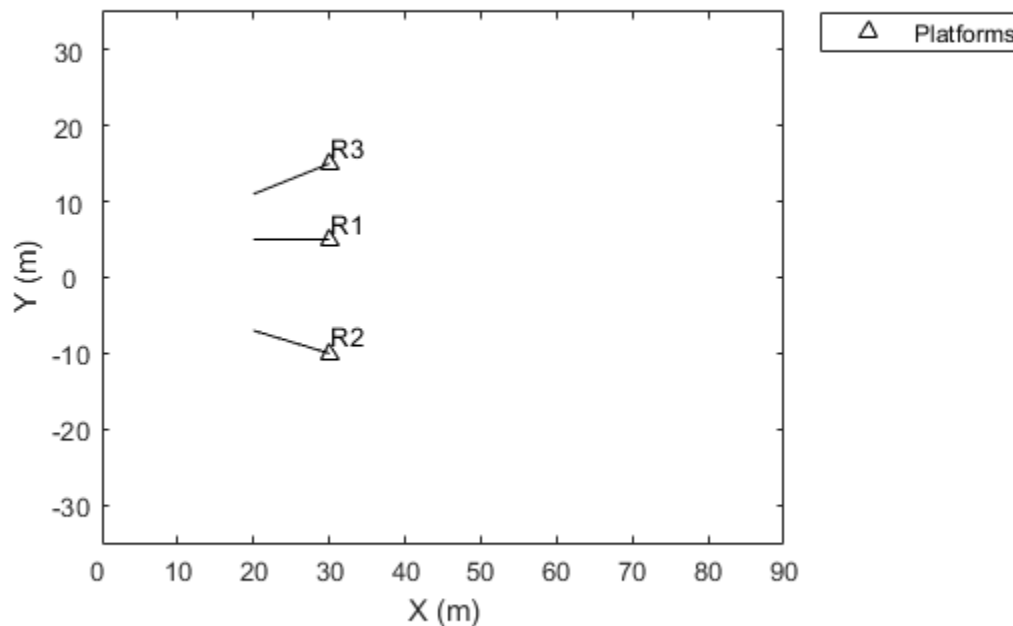
```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35],'ZLim',[1,10]);
```

Create a platform plotter with the name 'Platforms'.

```
plotter = platformPlotter(tp,'DisplayName','Platforms');
```

Update the theater plot with three platforms labeled, 'R1', 'R2', and 'R3'. Position the three platforms, in units of meters, at (30, 5, 4), (30, -10, 2), and (30, 15, 1), with corresponding velocities (in m/s) of (-10, 0, 2), (-10, 3, 1), and (-10, -4, 1), respectively.

```
positions = [30, 5, 4; 30, -10, 2; 30, 15, 1];
velocities = [-10, 0, 2; -10, 3, 1; -10, -4, 1];
labels = {'R1', 'R2', 'R3'};
plotPlatform(plotter, positions, velocities, labels);
```



## Input Arguments

### platPlotter — Platform plotter

platformPlotter object

Platform plotter, specified as a platformPlotter object.

### positions — Platform positions

real-valued matrix

Platform positions, specified as an  $M$ -by-3 real-valued matrix, where  $M$  is the number of platforms. Each column of the matrix corresponds to the  $x$ -,  $y$ -, and  $z$ -coordinates of the platform locations in meters.

### velocities — Platform velocities

$M$ -by-3 real-valued matrix

Platform velocities, specified as an  $M$ -by-3 real-valued matrix, where  $M$  is the number of platforms. Each column of the matrix corresponds to the  $x$ ,  $y$ , and  $z$  velocities of the platforms. If specified, `velocities` must have the same dimensions as `positions`.

### labels — Platform labels

cell array

Platform labels, specified as an  $M$ -by-1 cell array of character vectors, where  $M$  is the number of platforms. `labels` contains the text labels corresponding to the  $M$  platforms specified in `positions`. If `labels` is omitted, no labels are plotted.

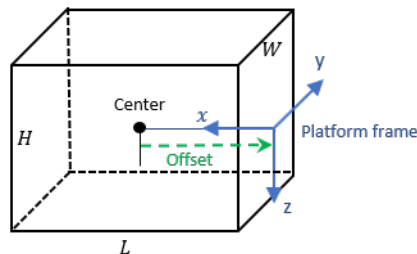
### dimensions — Platform dimensions

$M$ -by-1 array of dimension structure

Platform dimensions, specified as an  $M$ -by-1 array of dimension structures, where  $M$  is the number of platforms. The fields of each dimension structure are:

#### Fields of Dimensions

Fields	Description
Length	Dimension of a cuboid along the $x$ direction
Width	Dimension of a cuboid along the $y$ direction
Height	Dimension of a cuboid along the $z$ direction
OriginOffset	Position of the platform coordinate frame origin with respect to the cuboid center, specified as a vector of three elements



### meshes — Platform meshes

$M$ -element array of `extendedObjectMesh` object

Platform meshes, specified as an  $M$ -element array of `extendedObjectMesh` objects.

### orientations — Platform orientations

$3$ -by- $3$ -by- $M$  array of rotation matrix |  $M$ -element array of quaternion object

Platform orientations, specified as a  $3$ -by- $3$ -by- $M$  array of rotation matrices, or an  $M$ -element array of quaternion objects.

### See Also

`platformPlotter` | `theaterPlot`

Introduced in R2021a

# trackPlotter

Create track plotter

## Syntax

```
tPlotter = trackPlotter(tp)
tPlotter = trackPlotter(tp,Name,Value)
```

## Description

`tPlotter = trackPlotter(tp)` creates a track plotter for use with the theater plot `tp`.

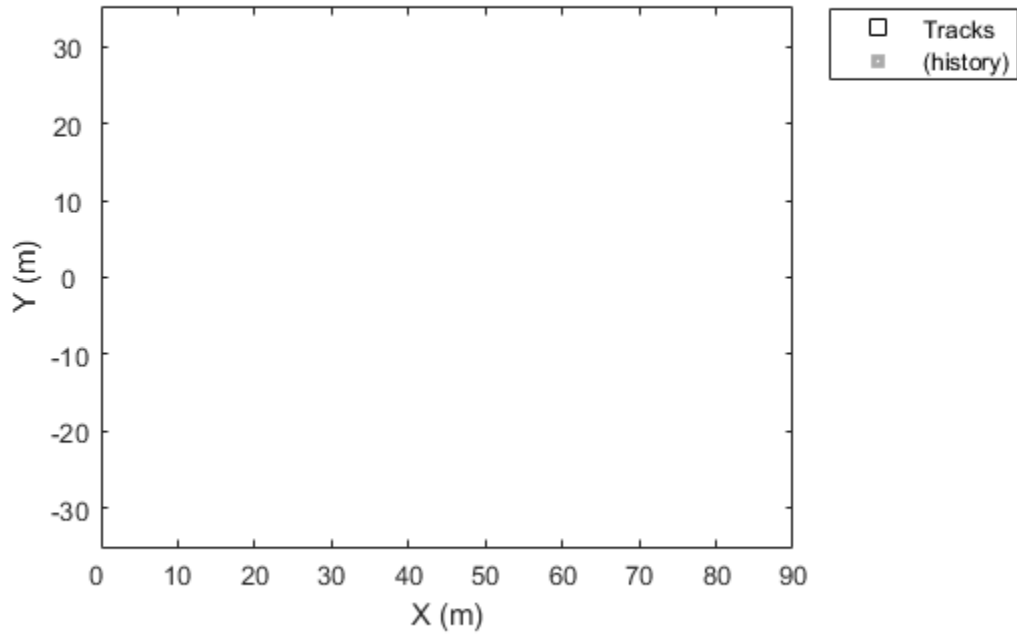
`tPlotter = trackPlotter(tp,Name,Value)` creates a track plotter with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Plot Tracks in Theater Plot

Create a theater plot. Create a track plotter with `DisplayName` set to 'Tracks' and with `HistoryDepth` set to 5.

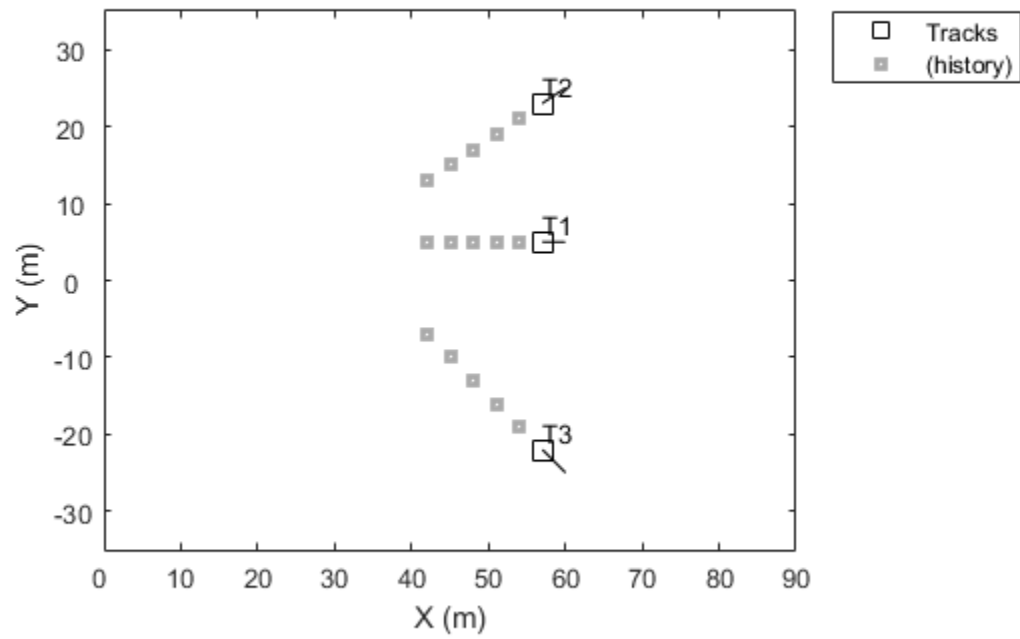
```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35]);
tPlotter = trackPlotter(tp,'DisplayName','Tracks','HistoryDepth',5);
```



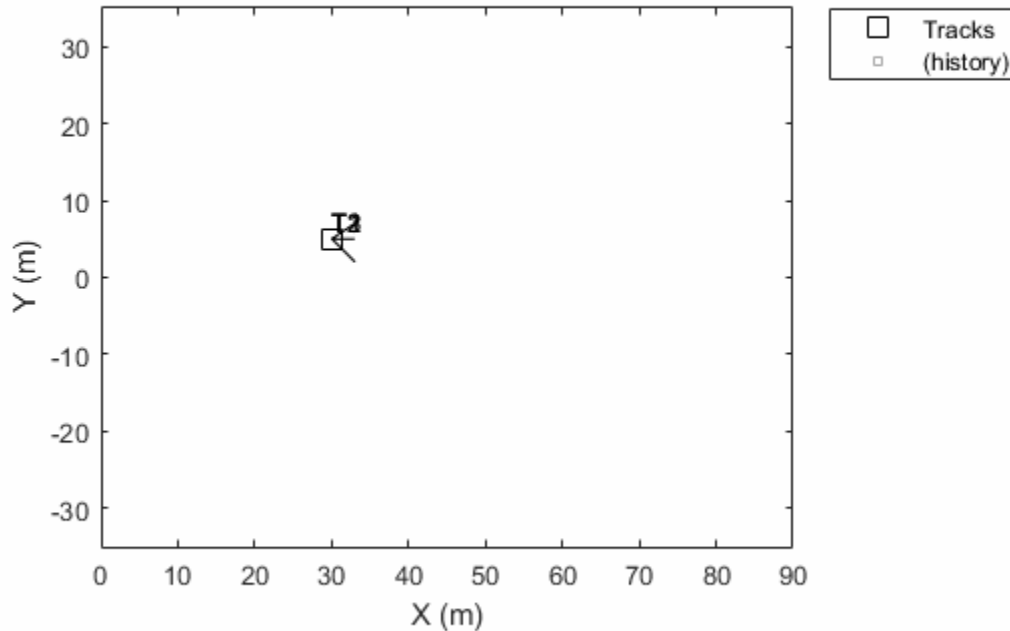
Update the track plotter with three tracks labeled 'T1', 'T2', and 'T3' with start positions in units of meters all starting at (30, 5, 1) with corresponding velocities (in m/s) of (3, 0, 1), (3, 2, 2) and (3, -3, 5), respectively. Update the tracks with the velocities for ten iterations.

```
positions = [30, 5, 1; 30, 5, 1; 30, 5, 1];
velocities = [3, 0, 1; 3, 2, 2; 3, -3, 5];
labels = {'T1', 'T2', 'T3'};
for i=1:10
    plotTrack(tPlotter, positions, velocities, labels)
    positions = positions + velocities;
end
```





This animation loops through all the generated plots.



## Input Arguments

### tp — Theater plot

theaterPlot object

Theater plot, specified as a theaterPlot object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'MarkerSize',10

### DisplayName — Plot name to display in legend

character vector | string scalar

Plot name to display in legend, specified as the comma-separated pair consisting of 'DisplayName' and a character vector or string scalar. If no name is specified, no entry is shown.

Example: 'DisplayName','Radar Detections'

### HistoryDepth — Number of previous track updates to display

0 (default) | nonnegative integer less than or equal to 10,000

Number of previous track updates to display, specified as the comma-separated pair consisting of 'HistoryDepth' and a nonnegative integer less than or equal to 10,000. If set to 0, then no previous updates are rendered.

### ConnectHistory — Connect tracks flag

'off' (default) | 'on'

Connect tracks flag, specified as either 'on' or 'off'. When set to 'on', tracks with the same label or track identifier between consecutive updates are connected with a line. This property can only be specified when creating the trackPlotter. The default is 'off'.

To use the trackIDs on page 4-0 input argument of plotTrack, 'ConnectHistory' must be 'on'. If trackIDs on page 4-0 is omitted when 'ConnectHistory' is 'on', then the track identifiers are derived from the labels input instead.

### ColorizeHistory — Colorize track history

'off' (default) | 'on'

Colorize track history, specified as either 'on' or 'off'. When set to 'on', tracks with the same label or track identifier between consecutive updates are connected with a line of a different color. This property can only be specified when creating the trackPlotter. The default is 'off'.






ColorizedHistory is applicable only when ConnectHistory is 'on'.

### Marker — Marker symbol

's' (default) | character vector | string scalar

Marker symbol, specified as the comma-separated pair consisting of 'Marker' and one of these symbols.

Marker	Description	Resulting Marker
'o'	Circle	○
'+'	Plus sign	+
'*'	Asterisk	*
'.'	Point	•
'x'	Cross	×
'_'	Horizontal line	—
' '	Vertical line	
's'	Square	□
'd'	Diamond	◇
'^'	Upward-pointing triangle	△

Marker	Description	Resulting Marker
'v'	Downward-pointing triangle	
'>'	Right-pointing triangle	
'<'	Left-pointing triangle	
'p'	Pentagram	
'h'	Hexagram	
'none'	No markers	Not applicable

**MarkerSize — Size of marker**

10 (default) | positive integer

Size of marker, specified as the comma-separated pair consisting of 'MarkerSize' and a positive integer in points.

**MarkerEdgeColor — Marker outline color**

'black' (default) | character vector | string scalar | RGB triplet | hexadecimal color code

Marker outline color, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a character vector, a string scalar, an RGB triplet, or a hexadecimal color code.

**MarkerFaceColor — Marker fill color**

'none' (default) | character vector | string scalar | RGB triplet | hexadecimal color code

Marker outline color, specified as the comma-separated pair consisting of 'MarkerFaceColor' and a character vector, a string scalar, an RGB triplet, a hexadecimal color code, or 'none'. The default is 'none'.

**FontSize — Font size for labeling tracks**

10 (default) | positive integer

Font size for labeling tracks, specified as the comma-separated pair consisting of 'FontSize' and a positive integer that represents font point size.

**LabelOffset — Gap between label and positional point**

[0 0 0] (default) | three-element row vector

Gap between label and positional point it annotates, specified as the comma-separated pair consisting of 'LabelOffset' and a three-element row vector. Specify the [x y z] offset in meters.

**VelocityScaling — Scale factor for magnitude length of velocity vectors**

1 (default) | positive scalar

Scale factor for magnitude length of velocity vectors, specified as the comma-separated pair consisting of 'VelocityScaling' and a positive scalar. The plot renders the magnitude vector value as  $VK$ , where  $V$  is the magnitude of the velocity in meters per second, and  $K$  is the value of VelocityScaling.

**Tag — Tag to associate with the plotter**

'PlotterN' (default) | character vector | string scalar

Tag to associate with the plotter, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. The default value is 'Plotter $N$ ', where  $N$  is an integer that corresponds to the  $N$ th plotter associated with the theaterPlot.

Tags provide a way to identify plotter objects, for example when searching using findPlotter.

**See Also**

theaterPlot | plotTrack | clearData | clearPlotterData

**Introduced in R2021a**

## plotTrack

Plot set of tracks in theater track plotter

### Syntax

```
plotTrack(tPlotter,positions)
plotTrack(tPlotter,positions,velocities)
plotTrack( ____,covariances)
plotTrack(tPlotter,positions, ____,labels)
plotTrack(tPlotter,positions, ____,labels,trackIDs)
plotTrack(tPlotter,positions, ____,dimensions,orientations)
```

### Description

`plotTrack(tPlotter,positions)` specifies positions of  $M$  tracked objects whose positions are plotted by the track plotter `tPlotter`. Specify the positions as an  $M$ -by-3 matrix, where each column of positions corresponds to the  $x$ -,  $y$ -, and  $z$ -coordinates of the object locations.

`plotTrack(tPlotter,positions,velocities)` also specifies the corresponding velocities of the objects. Velocities are plotted as line vectors emanating from the positions of the detections. If specified, `velocities` must have the same dimensions as `positions`. If unspecified, no velocity information is plotted.

`plotTrack( ____,covariances)` also specifies the covariances of the  $M$  track uncertainties. The input argument `covariances` is a 3-by-3-by- $M$  array of covariances that are centered at the track positions. The uncertainties are plotted as an ellipsoid. You can use this syntax with any of the previous syntaxes.

`plotTrack(tPlotter,positions, ____,labels)` also specifies the labels and positions of the  $M$  objects whose positions are estimated by a tracker. The input argument `labels` is an  $M$ -by-1 cell array of character vectors that correspond to the  $M$  detections specified in `positions`. If omitted, no labels are plotted.

`plotTrack(tPlotter,positions, ____,labels,trackIDs)` also specifies the unique track identifiers for each track when the 'ConnectHistory' on page 4-0 property of `tPlotter` is set to 'on'. The input argument `trackIDs` can be an  $M$ -by-1 array of unique integer values, an  $M$ -by-1 array of strings, or an  $M$ -by-1 cell array of unique character vectors.

If `trackIDs` is omitted when 'ConnectHistory' is 'on', then the track identifiers are derived from the labels input instead. The `trackIDs` input is ignored when 'ConnectHistory' is 'off'.

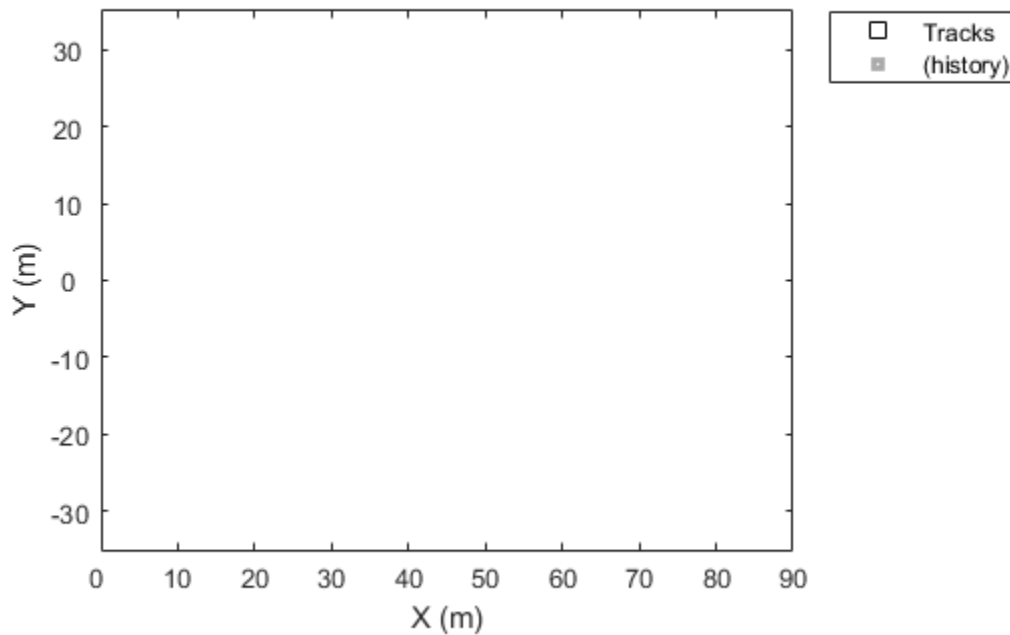
`plotTrack(tPlotter,positions, ____,dimensions,orientations)` specifies the dimension and orientation of each tracked object in the plot.

### Examples

### Plot Tracks in Theater Plot

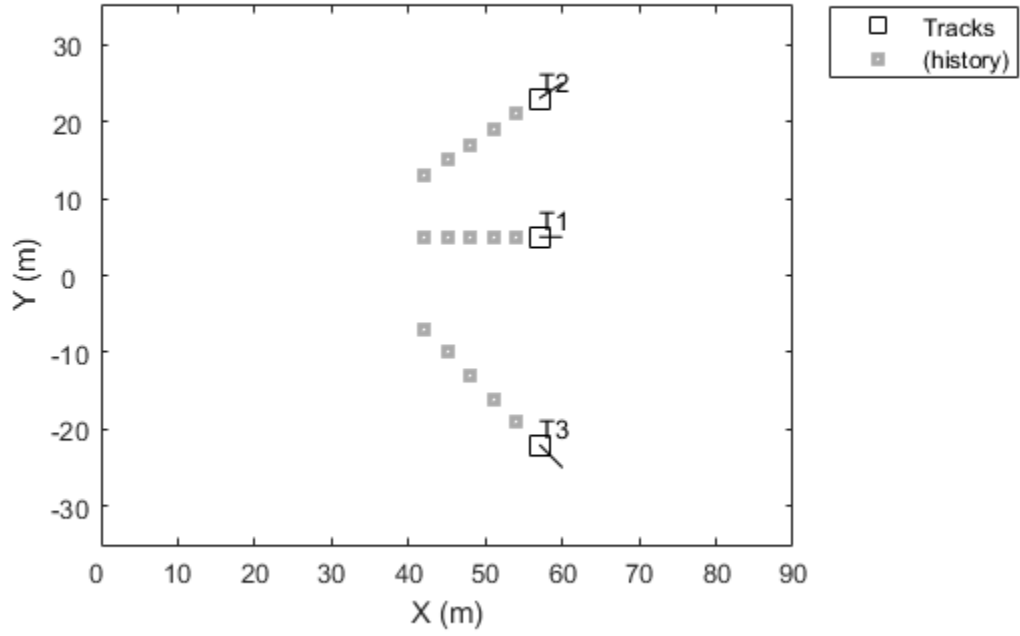
Create a theater plot. Create a track plotter with `DisplayName` set to 'Tracks' and with `HistoryDepth` set to 5.

```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35]);
tPlotter = trackPlotter(tp,'DisplayName','Tracks','HistoryDepth',5);
```



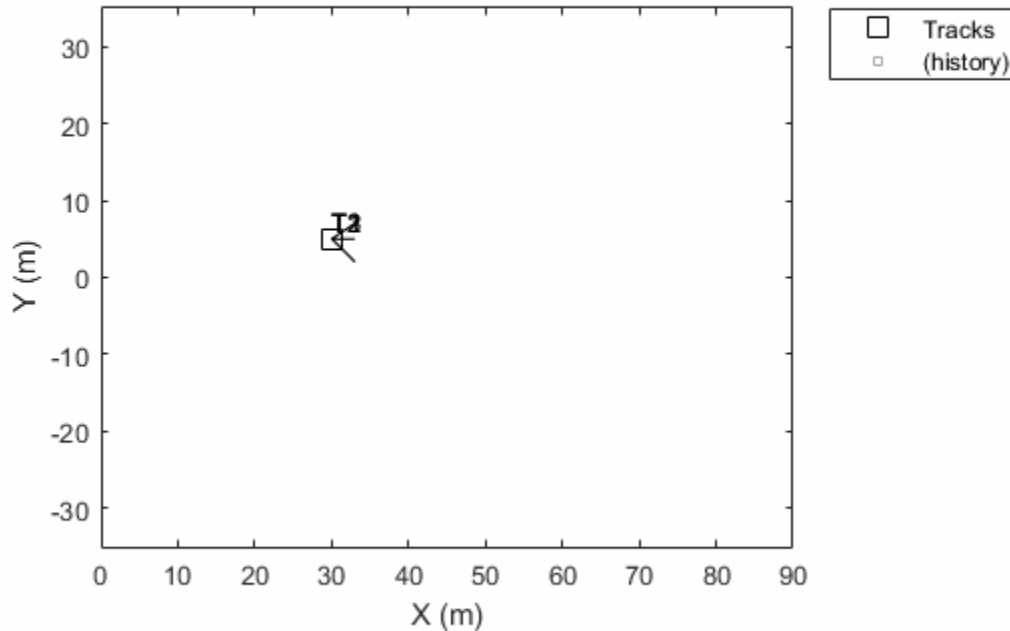
Update the track plotter with three tracks labeled 'T1', 'T2', and 'T3' with start positions in units of meters all starting at (30, 5, 1) with corresponding velocities (in m/s) of (3, 0, 1), (3, 2, 2) and (3, -3, 5), respectively. Update the tracks with the velocities for ten iterations.

```
positions = [30, 5, 1; 30, 5, 1; 30, 5, 1];
velocities = [3, 0, 1; 3, 2, 2; 3, -3, 5];
labels = {'T1','T2','T3'};
for i=1:10
    plotTrack(tPlotter, positions, velocities, labels)
    positions = positions + velocities;
end
```



This animation loops through all the generated plots.





### Plot Track Uncertainties

Create a theater plot. Create a track plotter with `DisplayName` set to 'Uncertain Track'.

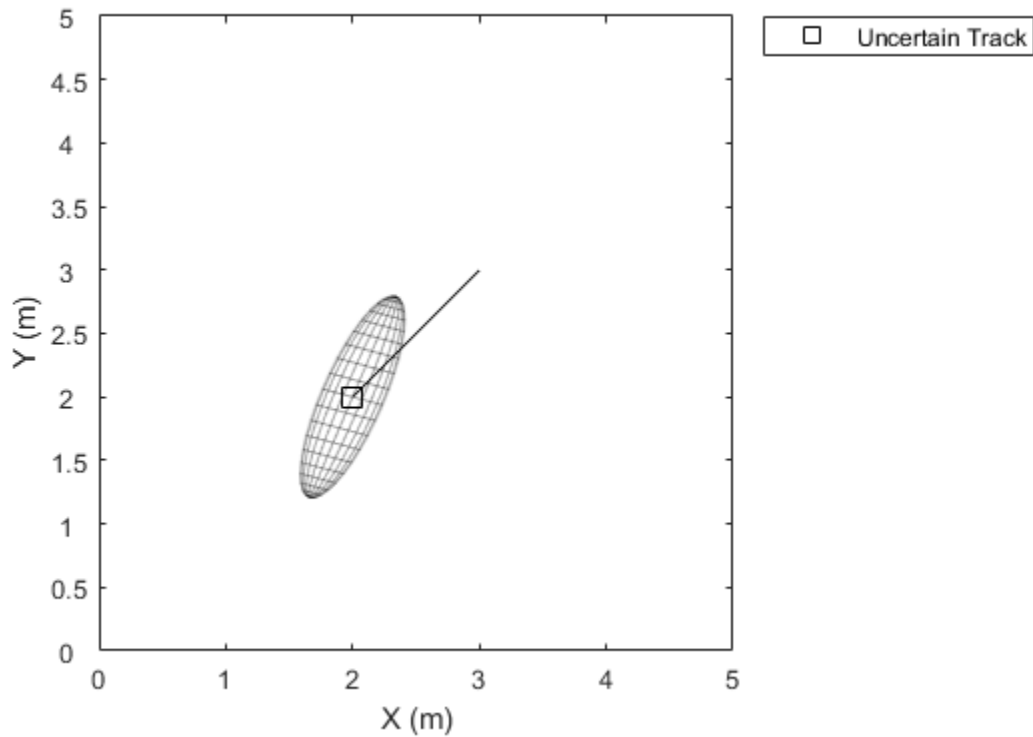
```
tp = theaterPlot('Xlim',[0 5], 'Ylim',[0 5]);
tPlotter = trackPlotter(tp, 'DisplayName', 'Uncertain Track');
```

Update the track plotter with a track at a position in meters (2,2,1) and velocity (in meters/second) of (1,1,3). Also create a random 3-by-3 covariance matrix representing track uncertainties. For purposes of reproducibility, set the random seed to the default value.

```
positions = [2, 2, 1];
velocities = [1, 1, 3];
rng default
covariances = randn(3,3);
```

Plot the track with the covariances plotted as an ellipsoid.

```
plotTrack(tPlotter, positions, velocities, covariances)
```



## Input Arguments

### **tPlotter** — Track plotter

trackPlotter object

Track plotter, specified as a trackPlotter object.

### **positions** — Tracked object positions

real-valued matrix

Tracked object positions, specified as an  $M$ -by-3 real-valued matrix, where  $M$  is the number of objects. Each column of **positions** corresponds to the  $x$ -,  $y$ -, and  $z$ -coordinates of the object locations in meters.

### **velocities** — Tracked object velocities

real-valued matrix

Tracked object velocities, specified as an  $M$ -by-3 real-valued matrix, where  $M$  is the number of objects. Each column of **velocities** corresponds to the  $x$ ,  $y$ , and  $z$  velocities of the objects. If specified, **velocities** must have the same dimensions as **positions**.

### **covariances** — Track uncertainties

real-valued array

Track uncertainties of  $M$  tracked objects, specified as a 3-by-3-by- $M$  real-valued array of covariances. The covariances are centered at the track positions, and are plotted as an ellipsoid.

### Labels — Tracked object labels

cell array

Tracked object labels, specified as a  $M$ -by-1 cell array of character vectors, where  $M$  is the number of objects. The argument `labels` contains the text labels corresponding to the  $M$  objects specified in `positions`. If `labels` is omitted, no labels are plotted.

### trackIDs — Unique track identifiers

integer vector | string array | cell array

Unique track identifiers for the  $M$  tracked objects, specified as an  $M$ -by-1 integer vector, an  $M$ -by-1 array of strings, or an  $M$ -by-1 cell array of character vectors. The elements of `trackIDs` must be unique.

The `trackIDs` input is ignored when the property 'ConnectHistory' of `tPlotter` is 'off'. If `trackIDs` is omitted when 'ConnectHistory' is 'on', then the track identifiers are derived from the labels input instead.

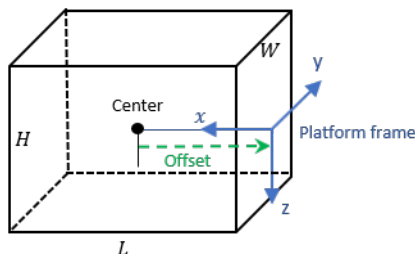
### dimensions — Platform dimensions

$M$ -by-1 array of dimension structure

Platform dimensions, specified as an  $M$ -by-1 array of dimension structures, where  $M$  is the number of platforms. The fields of each dimension structure are:

#### Fields of Dimensions

Fields	Description
Length	Dimension of a cuboid along the $x$ direction
Width	Dimension of a cuboid along the $y$ direction
Height	Dimension of a cuboid along the $z$ direction
OriginOffset	Position of the platform coordinate frame origin with respect to the cuboid center, specified as a vector of three elements



### orientations — Platform orientations

3-by-3-by- $M$  array of rotation matrix |  $M$ -element array of quaternion object

Platform orientations, specified as a 3-by-3-by- $M$  array of rotation matrices, or an  $M$ -element array of quaternion objects.

**See Also**

`theaterPlot` | `trackPlotter` | `clearData` | `clearPlotterData`

**Introduced in R2021a**

# trajectoryPlotter

Create trajectory plotter

## Syntax

```
trajPlotter = trajectoryPlotter(tp)
trajPlotter = trajectoryPlotter(tp,Name,Value)
```

## Description

`trajPlotter = trajectoryPlotter(tp)` creates a trajectory plotter for use with the theater plot `tp`.

`trajPlotter = trajectoryPlotter(tp,Name,Value)` creates a trajectory plotter with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Moving Platform on Trajectory in radarScenario

This example shows how to create an animation of a platform moving on a trajectory.

First, create a `radarScenario` and add waypoints for a trajectory.

```
ts = radarScenario;
height = 100;
d = 1;
wayPoints = [ ...
    -30  -25  height;
    -30   25-d height;
    -30+d 25  height;
    -10-d 25  height;
    -10   25-d height;
    -10  -25+d height;
    -10+d -25  height;
    10-d -25  height;
    10   -25+d height;
    10   25-d height;
    10+d 25  height;
    30-d 25  height;
    30   25-d height;
    30  -25+d height;
    30  -25  height];
```

Specify a time for each waypoint.

```
elapsedTime = linspace(0,10,size(wayPoints,1));
```

Next, create a platform in the tracking scenario and add trajectory information using the `trajectory` method.

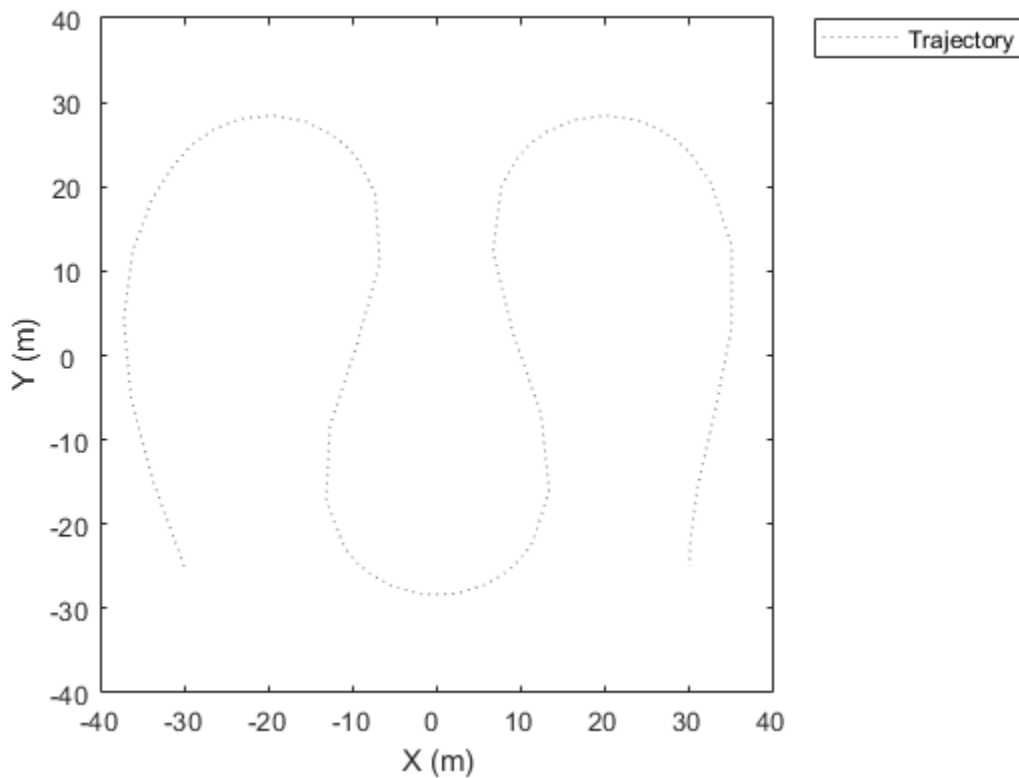
```
target = platform(ts);
traj = waypointTrajectory('Waypoints',wayPoints,'TimeOfArrival',elapsedTime);
target.Trajectory = traj;
```

Record the tracking scenario to retrieve the platform's trajectory.

```
r = record(ts);
pposes = [r(:).Poses];
pposition = vertcat(pposes.Position);
```

Create a theater plot to display the recorded trajectory.

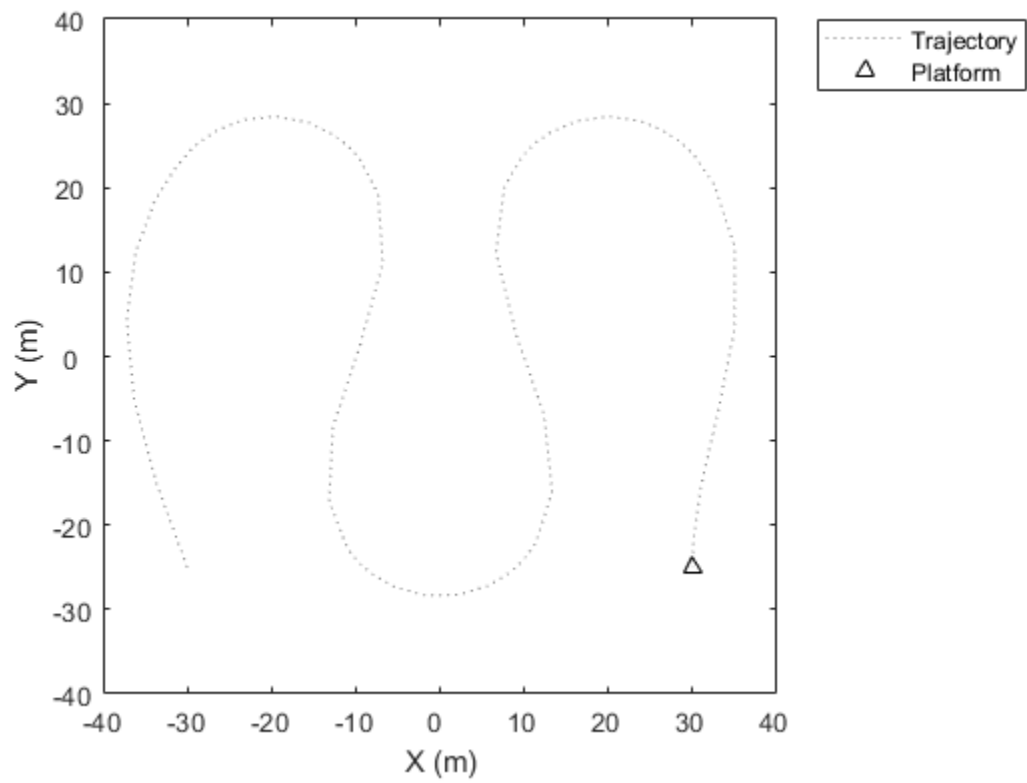
```
tp = theaterPlot('XLim',[-40 40],'YLim',[-40 40]);
trajPlotter = trajectoryPlotter(tp,'DisplayName','Trajectory');
plotTrajectory(trajPlotter,{pposition})
```



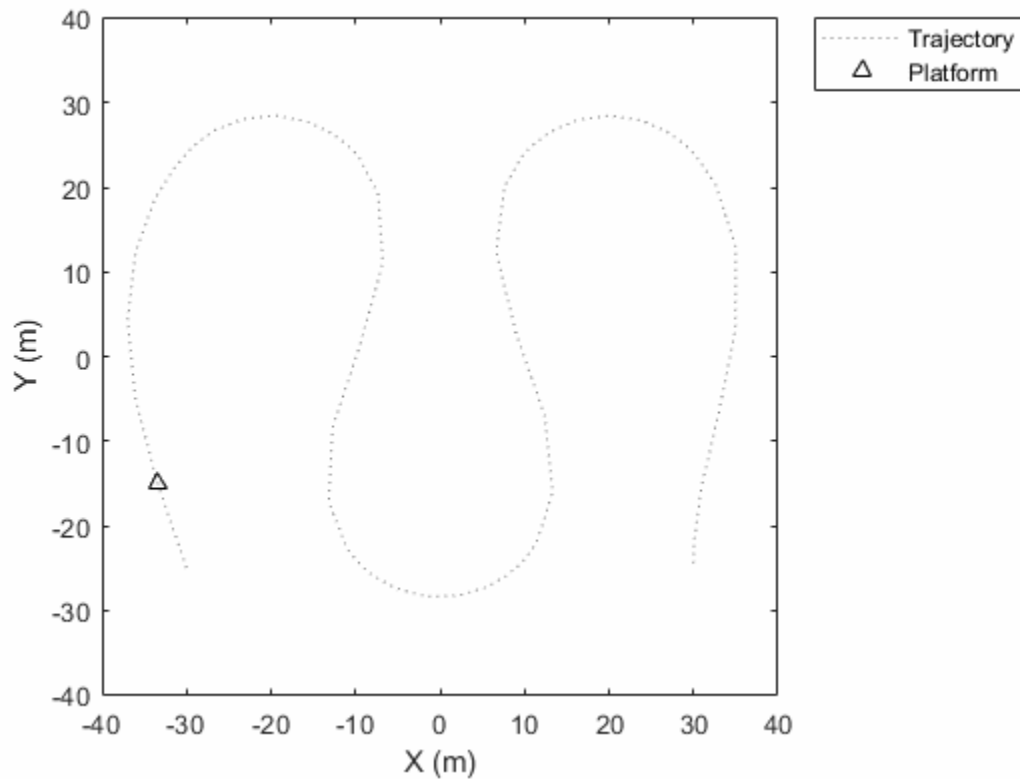
Animate using the platformPlotter.

```
restart(ts);
trajPlotter = platformPlotter(tp,'DisplayName','Platform');

while advance(ts)
    p = pose(target,'true');
    plotPlatform(trajPlotter, p.Position);
    pause(0.1)
end
```



This animation loops through all the generated plots.



## Input Arguments

### tp — Theater plot

theaterPlot object

Theater plot, specified as a theaterPlot object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `'LineStyle','--'`

### DisplayName — Plot name to display in legend

character vector | string scalar

Plot name to display in legend, specified as the comma-separated pair consisting of `'DisplayName'` and a character vector or string scalar. If no name is specified, no entry is shown.

Example: `'DisplayName','Radar Detections'`



**Color – Trajectory color**

'gray' (default) | character vector | string scalar | RGB triplet | hexadecimal color code

Trajectory color, specified as the comma-separated pair consisting of 'Color' and a character vector, a string scalar, an RGB triplet, or a hexadecimal color code.

**LineStyle – Line style**

':' (default) | '-' | '--' | '-.'

Line style used to plot the trajectory, specified as one of these values.

Value	Description
':'	Dotted line (default)
'-'	Solid line
'--'	Dashed line
'-.'	Dash-dotted line

**LineWidth – Line width**

0.5 (default) | positive scalar

Line width of the trajectory, specified in points size as the comma-separated pair consisting of 'LineWidth' and a positive scalar.

**Tag – Tag to associate with the plotter**

'PlotterN' (default) | character vector | string scalar

Tag to associate with the plotter, specified as the comma-separated pair consisting of 'Tag' and a character vector or string scalar. The default value is 'PlotterN', where *N* is an integer that corresponds to the *N*th plotter associated with the theaterPlot.

Tags provide a way to identify plotter objects, for example when searching using findPlotter.

**See Also**

theaterPlot | plotTrajectory | clearData | clearPlotterData

**Introduced in R2021a**

## plotTrajectory

Plot set of trajectories in trajectory plotter

### Syntax

```
plotTrajectory(trajPlotter, trajCoordList)
```

### Description

`plotTrajectory(trajPlotter, trajCoordList)` specifies the trajectories to show in the trajectory plotter, `trajPlotter`. The input argument `trajCoordList` is a cell array of  $M$ -by-3 matrices, where  $M$  is the number of points in the trajectory. Each matrix in `trajCoordList` can have a different number of rows. The first, second, and third columns of each matrix correspond to the  $x$ -,  $y$ -, and  $z$ -coordinates of a curve through  $M$  points that represent the corresponding trajectory.

### Examples

#### Moving Platform on Trajectory in radarScenario

This example shows how to create an animation of a platform moving on a trajectory.

First, create a `radarScenario` and add waypoints for a trajectory.

```
ts = radarScenario;
height = 100;
d = 1;
wayPoints = [ ...
    -30  -25  height;
    -30   25-d height;
    -30+d 25  height;
    -10-d 25  height;
    -10   25-d height;
    -10  -25+d height;
    -10+d -25  height;
    10-d -25  height;
    10   -25+d height;
    10   25-d height;
    10+d 25  height;
    30-d 25  height;
    30   25-d height;
    30  -25+d height;
    30  -25  height];
```

Specify a time for each waypoint.

```
elapsedTime = linspace(0,10,size(wayPoints,1));
```

Next, create a platform in the tracking scenario and add trajectory information using the `trajectory` method.

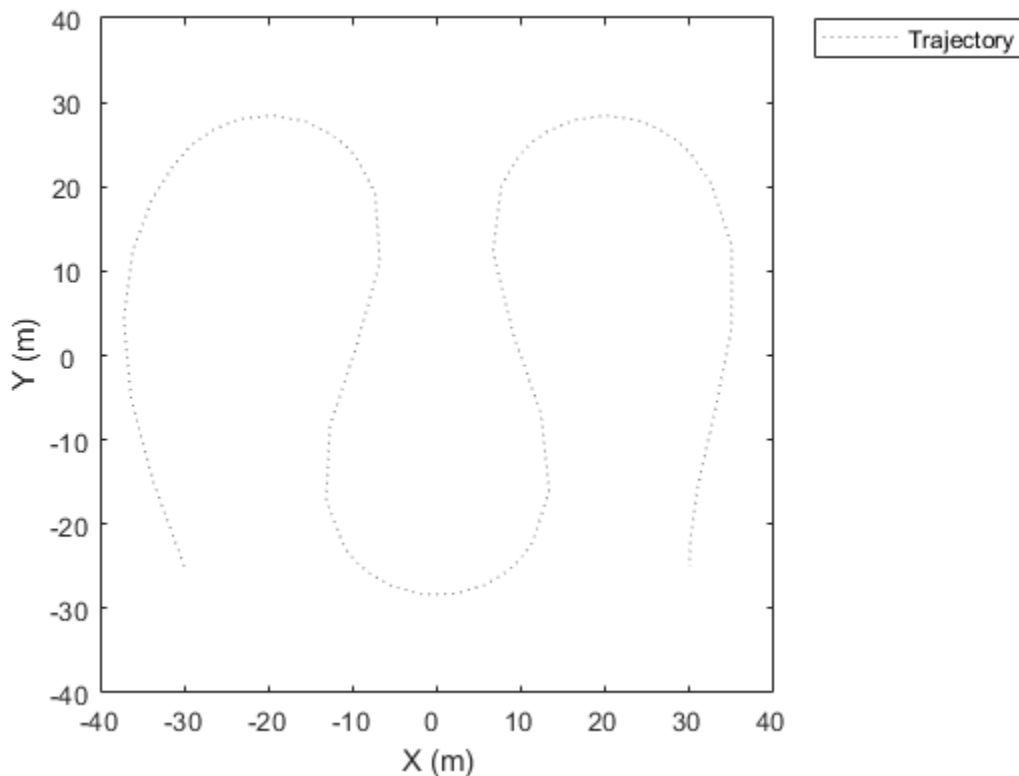
```
target = platform(ts);
traj = waypointTrajectory('Waypoints',wayPoints,'TimeOfArrival',elapsedTime);
target.Trajectory = traj;
```

Record the tracking scenario to retrieve the platform's trajectory.

```
r = record(ts);
pposes = [r(:).Poses];
pposition = vertcat(pposes.Position);
```

Create a theater plot to display the recorded trajectory.

```
tp = theaterPlot('XLim',[-40 40],'YLim',[-40 40]);
trajPlotter = trajectoryPlotter(tp,'DisplayName','Trajectory');
plotTrajectory(trajPlotter,{pposition})
```

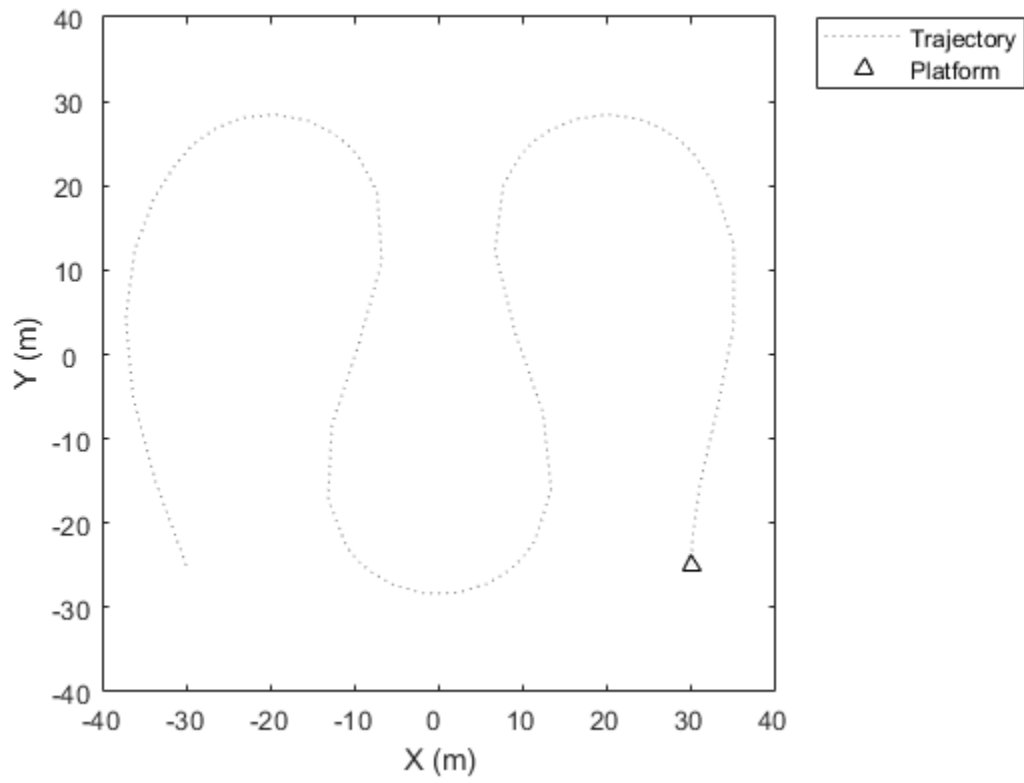


Animate using the platformPlotter.

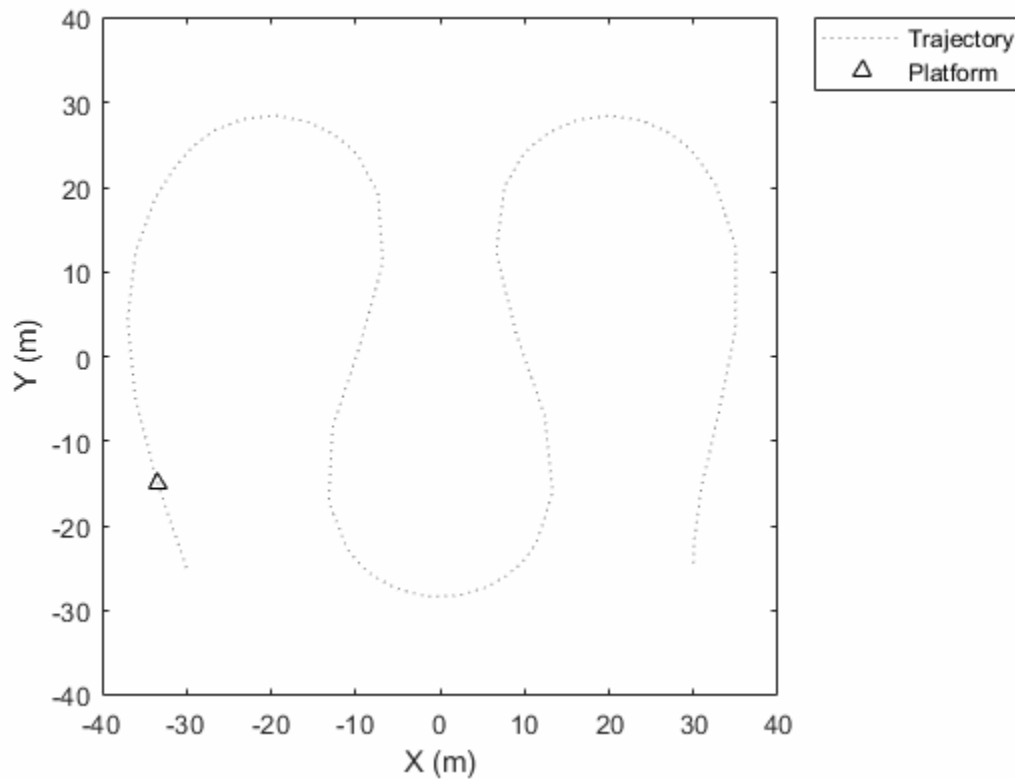
```
restart(ts);
trajPlotter = platformPlotter(tp,'DisplayName','Platform');
```

```
while advance(ts)
    p = pose(target,'true');
    plotPlatform(trajPlotter, p.Position);
    pause(0.1)
```

```
end
```



This animation loops through all the generated plots.



## Input Arguments

### **trajPlotter** — Trajectory plotter

trajectoryPlotter object

Trajectory plotter, specified as a trajectoryPlotter object.

### **trajCoordList** — Coordinates of trajectories

cell array

Coordinates of trajectories to show, specified as a cell array of  $M$ -by-3 matrices, where  $M$  is the number of points in the trajectory. Each matrix in `trajCoordList` can have a different number of rows. The first, second, and third columns of each matrix correspond to the  $x$ -,  $y$ -, and  $z$ -coordinates of a curve through  $M$  points that represent the corresponding trajectory.

Example: `coordList = {[1 2 3; 4 5 6; 7,8,9];[4 2 1; 4 3 1];[4 4 4; 3 1 2; 9 9 9; 1 0 2]}` specifies three different trajectories.

## See Also

trajectoryPlotter | theaterPlot | clearData | clearPlotterData

Introduced in R2021a

## trackHistoryLogic

Confirm and delete tracks based on recent track history

### Description

The `trackHistoryLogic` object determines if a track should be confirmed or deleted based on the track history. A track should be confirmed if there are at least  $M_c$  hits in the recent  $N_c$  updates. A track should be deleted if there are at least  $M_d$  misses in the recent  $N_d$  updates.

The confirmation and deletion decisions contribute to the track management by a `radarTracker` object.

### Creation

#### Syntax

```
logic = trackHistoryLogic
logic = trackHistoryLogic(Name, Value, ...)
```

#### Description

`logic = trackHistoryLogic` creates a `trackHistoryLogic` object with default confirmation and deletion thresholds.

`logic = trackHistoryLogic(Name, Value, ...)` specifies the properties of the track history logic object using one or more `Name, Value` pair arguments. Any unspecified properties take default values.

### Properties

#### ConfirmationThreshold — Confirmation threshold

[2 3] (default) | positive integer scalar | 2-element vector of positive integers

Confirmation threshold, specified as a positive integer scalar or 2-element vector of positive integers. If the logic score is above this threshold, the track is confirmed. `ConfirmationThreshold` has the form  $[M_c N_c]$ , where  $M_c$  is the number of hits required for confirmation in the recent  $N_c$  updates. When specified as a scalar, then  $M_c$  and  $N_c$  have the same value.

Example: [3 5]

Data Types: `single` | `double`

#### DeletionThreshold — Deletion threshold

[6 6] (default) | positive integer scalar | 2-element vector of positive integers

Deletion threshold, specified as a positive integer scalar or 2-element vector of positive integers. If the logic score is above this threshold, the track is deleted. `DeletionThreshold` has the form  $[M_d N_d]$ , where  $M_d$  is the number of misses required for deletion in the recent  $N_d$  updates. When specified as a scalar, then  $M_d$  and  $N_d$  have the same value.

Example: [5 5]

Data Types: single | double

## History — Track history

logical vector

This property is read-only.

Track history, specified as a logical vector of length  $N$ , where  $N$  is the larger of the second element in the `ConfirmationThreshold` and the second element in the `DeletionThreshold`. The first element is the most recent update. A `true` value indicates a hit and a `false` value indicates a miss.

## Object Functions

<code>init</code>	Initialize track logic with first hit
<code>hit</code>	Update track logic with subsequent hit
<code>miss</code>	Update track logic with miss
<code>checkConfirmation</code>	Check if track should be confirmed
<code>checkDeletion</code>	Check if track should be deleted
<code>output</code>	Get current state of track logic
<code>reset</code>	Reset state of track logic
<code>sync</code>	Synchronize trackHistoryLogic objects
<code>clone</code>	Create copy of track logic

## Examples

### Create and Update History-Based Logic

Create a history-based logic. Specify confirmation threshold values  $M_c$  and  $N_c$  as the vector [3 5]. Specify deletion threshold values  $M_d$  and  $N_d$  as the vector [6 7].

```
historyLogic = trackHistoryLogic('ConfirmationThreshold',[3 5], ...
    'DeletionThreshold',[6 7])
```

```
historyLogic =
    trackHistoryLogic with properties:
        ConfirmationThreshold: [3 5]
        DeletionThreshold: [6 7]
        History: [0 0 0 0 0 0 0]
```

Initialize the logic, which records a hit as the first update to the logic.

```
init(historyLogic)
history = historyLogic.History;
disp(['History: ', num2str(history), '.']);
```

```
History: [1 0 0 0 0 0 0].
```

Update the logic four more times, where only the odd updates register a hit. The confirmation flag is `true` by the end of the fifth update, because three hits ( $M_c$ ) are counted in the most recent five updates ( $N_c$ ).

```

for i = 2:5
    isOdd = logical(mod(i,2));
    if isOdd
        hit(historyLogic)
    else
        miss(historyLogic)
    end

    history = historyLogic.History;
    confFlag = checkConfirmation(historyLogic);
    delFlag = checkDeletion(historyLogic,true,i);
    disp(['History: [',num2str(history),']. Confirmation Flag: ',num2str(confFlag), ...
        '. Deletion Flag: ',num2str(delFlag)']);
end

History: [0 1 0 0 0 0 0]. Confirmation Flag: 0. Deletion Flag: 0
History: [1 0 1 0 0 0 0]. Confirmation Flag: 0. Deletion Flag: 0
History: [0 1 0 1 0 0 0]. Confirmation Flag: 0. Deletion Flag: 0
History: [1 0 1 0 1 0 0]. Confirmation Flag: 1. Deletion Flag: 0

```

Update the logic with a miss six times. The deletion flag is true by the end of the fifth update, because six misses ( $Md$ ) are counted in the most recent seven updates ( $Nd$ ).

```

for i = 1:6
    miss(historyLogic);

    history = historyLogic.History;
    confFlag = checkConfirmation(historyLogic);
    delFlag = checkDeletion(historyLogic);
    disp(['History: [',num2str(history),']. Confirmation Flag: ',num2str(confFlag), ...
        '. Deletion Flag: ',num2str(delFlag)']);
end

History: [0 1 0 1 0 1 0]. Confirmation Flag: 0. Deletion Flag: 0
History: [0 0 1 0 1 0 1]. Confirmation Flag: 0. Deletion Flag: 0
History: [0 0 0 1 0 1 0]. Confirmation Flag: 0. Deletion Flag: 0
History: [0 0 0 0 1 0 1]. Confirmation Flag: 0. Deletion Flag: 0
History: [0 0 0 0 0 1 0]. Confirmation Flag: 0. Deletion Flag: 1
History: [0 0 0 0 0 0 1]. Confirmation Flag: 0. Deletion Flag: 1

```

## References

- [1] Blackman, S., and R. Popoli. *Design and Analysis of Modern Tracking Systems*. Boston, MA: Artech House, 1999.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

### See Also

radarTracker



**Introduced in R2021a**

## checkConfirmation

Check if track should be confirmed

### Syntax

```
tf = checkConfirmation(historyLogic)
```

### Description

`tf = checkConfirmation(historyLogic)` returns a flag that is `true` when at least  $M_c$  out of  $N_c$  recent updates of the track history logic object `historyLogic` are `true`.

### Examples

#### Check Confirmation of History-Based Logic

Create a history-based logic. Specify confirmation threshold values  $M_c$  and  $N_c$  as the vector `[2 3]`. Specify deletion threshold values  $M_d$  and  $N_d$  as the vector `[3 3]`.

```
historyLogic = trackHistoryLogic('ConfirmationThreshold',[2 3], ...
    'DeletionThreshold',[3 3])
```

```
historyLogic =
    trackHistoryLogic with properties:
```

```
    ConfirmationThreshold: [2 3]
    DeletionThreshold: [3 3]
    History: [0 0 0]
```

Initialize the logic, which records a hit as the first update to the logic. The confirmation flag is `false` because the number of hits is less than two ( $M_c$ ).

```
init(historyLogic)
history = output(historyLogic);
confFlag = checkConfirmation(historyLogic);
disp(['History: ',num2str(history),']. Confirmation Flag: ',num2str(confFlag)]);
```

```
History: [1 0 0]. Confirmation Flag: 0
```

Update the logic with a hit. The confirmation flag is `true` because two hits ( $M_c$ ) are counted in the most recent three updates ( $N_c$ ).

```
hit(historyLogic)
history = output(historyLogic);
confFlag = checkConfirmation(historyLogic);
disp(['History: ',num2str(history),']. Confirmation Flag: ',num2str(confFlag)]);
```

```
History: [1 1 0]. Confirmation Flag: 1
```

## Input Arguments

### **historyLogic** — Track history logic

trackHistoryLogic

Track history logic, specified as a trackHistoryLogic object.

## Output Arguments

### **tf** — Track should be confirmed

true | false

Track should be confirmed, returned as true or false.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

trackHistoryLogic

**Introduced in R2021a**

## checkDeletion

Check if track should be deleted

### Syntax

```
tf = checkDeletion(historyLogic)
tf = checkDeletion(historyLogic,tentativeTrack,age)
```

### Description

`tf = checkDeletion(historyLogic)` returns a flag that is true when at least  $Md$  out of  $Nd$  recent updates of the track history logic object `historyLogic` are false.

`tf = checkDeletion(historyLogic,tentativeTrack,age)` returns a flag that is true when the track is tentative and there are not enough detections to allow it to confirm. Use the logical flag `tentativeTrack` to indicate if the track is tentative and provide `age` as a numeric scalar.

### Examples

#### Check Deletion of History-Based Logic

Create a history-based logic. Specify confirmation threshold values  $M_c$  and  $N_c$  as the vector [2 3]. Specify deletion threshold values  $M_d$  and  $N_d$  as the vector [4 5].

```
historyLogic = trackHistoryLogic('ConfirmationThreshold',[2 3], ...
    'DeletionThreshold',[4 5])
```

```
historyLogic =
    trackHistoryLogic with properties:

        ConfirmationThreshold: [2 3]
        DeletionThreshold: [4 5]
        History: [0 0 0 0 0]
```

Initialize the logic, which records a hit as the first update to the logic. The confirmation flag is false because the number of hits is less than two ( $M_c$ ).

```
init(historyLogic)
history = output(historyLogic);
checkConfirmation(historyLogic)
```

```
ans = logical
     0
```

```
delFlag = checkDeletion(historyLogic);
disp(['History: ',num2str(history),']. Deletion Flag: ',num2str(delFlag)]);
```

```
History: [1 0 0 0 0]. Deletion Flag: 1
```

Update the logic with a hit. The confirmation flag is `true` because two hits ( $M_c$ ) are counted in the most recent three updates ( $N_c$ ).

```
hit(historyLogic)
history = output(historyLogic);
checkConfirmation(historyLogic)
```

```
ans = logical
     1
```

```
delFlag = checkDeletion(historyLogic);
disp(['History: ',num2str(history),']. Deletion Flag: ',num2str(delFlag)]);
```

```
History: [1 1 0 0 0]. Deletion Flag: 0
```

```
miss(historyLogic)
history = output(historyLogic);
checkConfirmation(historyLogic)
```

```
ans = logical
     1
```

```
delFlag = checkDeletion(historyLogic);
disp(['History: ',num2str(history),']. Deletion Flag: ',num2str(delFlag)]);
```

```
History: [0 1 1 0 0]. Deletion Flag: 0
```

```
miss(historyLogic)
history = output(historyLogic);
delFlag = checkDeletion(historyLogic);
checkConfirmation(historyLogic)
```

```
ans = logical
     0
```

```
disp(['History: ',num2str(history),']. Deletion Flag: ',num2str(delFlag)]);
```

```
History: [0 0 1 1 0]. Deletion Flag: 0
```

### Check Deletion of Tentative Track

Create a history-based logic. Specify confirmation threshold values  $M_c$  and  $N_c$  as the vector [2 3]. Specify deletion threshold values  $M_d$  and  $N_d$  as the vector [4 5].

```
historyLogic = trackHistoryLogic('ConfirmationThreshold',[2 3], ...
    'DeletionThreshold',5)
```

```
historyLogic =
    trackHistoryLogic with properties:
```

```
ConfirmationThreshold: [2 3]
DeletionThreshold: [5 5]
History: [0 0 0 0 0]
```

Initialize the logic, which records a hit as the first update to the logic. Then, record two misses.

```
init(historyLogic)
miss(historyLogic)
miss(historyLogic)
history = output(historyLogic)

history = 1x5 logical array

    0    0    1    0    0
```

The confirmation flag is `false` because the number of hits in the most recent 3 updates ( $N_c$ ) is less than 2 ( $M_c$ ).

```
confirmationFlag = checkConfirmation(historyLogic)

confirmationFlag = logical
    0
```

Check the deletion flag as if the track were not tentative. The deletion flag is `false` because the number of misses in the most recent 5 updates ( $N_m$ ) is less than 4 ( $M_c$ ).

```
deletionFlag = checkDeletion(historyLogic)

deletionFlag = logical
    0
```

Recheck the deletion flag, treating the track as tentative with an age of 3. The tentative deletion flag is `true` because there are not enough detections to allow the track to confirm.

```
tentativeDeletionFlag = checkDeletion(historyLogic,true,3)

tentativeDeletionFlag = logical
    1
```

## Input Arguments

### **historyLogic** — Track history logic

`trackHistoryLogic`

Track history logic, specified as a `trackHistoryLogic` object.

### **tentativeTrack** — Track is tentative

`false` | `true`

Track is tentative, specified as `false` or `true`. Use `tentativeTrack` to indicate if the track is tentative.

### **age** — Number of updates

numeric scalar

Number of updates since track initialization, specified as a numeric scalar.

## Output Arguments

### **tf** — Track can be deleted

true | false

Track can be deleted, returned as true or false.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

trackHistoryLogic

**Introduced in R2021a**

## clone

Create copy of track logic

### Syntax

```
clonedLogic = clone(logic)
```

### Description

`clonedLogic = clone(logic)` returns a copy of the current track logic object, `logic`.

### Examples

#### Clone Track History Logic

Create a history-based logic. Specify confirmation threshold values  $M_c$  and  $N_c$  as the vector [3 5]. Specify deletion threshold values  $M_d$  and  $N_d$  as the vector [6 7].

```
historyLogic = trackHistoryLogic('ConfirmationThreshold',[3 5], ...  
    'DeletionThreshold',[6 7])
```

```
historyLogic =  
    trackHistoryLogic with properties:  
  
    ConfirmationThreshold: [3 5]  
    DeletionThreshold: [6 7]  
    History: [0 0 0 0 0 0 0]
```

Initialize the logic, which records a hit as the first update to the logic.

```
init(historyLogic)
```

Update the logic four more times, where only the odd updates register a hit.

```
for i = 2:5  
    isOdd = logical(mod(i,2));  
    if isOdd  
        hit(historyLogic)  
    else  
        miss(historyLogic)  
    end  
end
```

Get the current state of the logic.

```
history = output(historyLogic)
```

```
history = 1x7 logical array  
  
    1    0    1    0    1    0    0
```



Create a copy of the logic. The clone has the same confirmation threshold, deletion threshold, and history as the original history logic.

```
clonedLogic = clone(historyLogic)

clonedLogic =
  trackHistoryLogic with properties:

    ConfirmationThreshold: [3 5]
    DeletionThreshold: [6 7]
    History: [1 0 1 0 1 0 0]
```

## Input Arguments

### **logic** — Track history logic

trackHistoryLogic object

Track history logic, specified as a trackHistoryLogic object.

## Output Arguments

### **clonedLogic** — Cloned track logic

trackHistoryLogic object

Cloned track logic, returned as a trackHistoryLogic object.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

trackHistoryLogic

**Introduced in R2021a**

## hit

Update track logic with subsequent hit

### Syntax

```
hit(historyLogic)
```

### Description

`hit(historyLogic)` updates the track history with a hit.

### Examples

#### Update History Logic with Hit

Create a history-based logic with the default confirmation and deletion thresholds.

```
historyLogic = trackHistoryLogic;
```

Initialize the logic, which records a hit as the first update to the logic. The first element of the 'History' property, which indicates the most recent update, is 1.

```
init(historyLogic)
history = historyLogic.History;
disp(['History: ', num2str(history), '.']);
```

```
History: [1 0 0 0 0 0].
```

Update the logic with a hit. The first two elements of the 'History' property are 1.

```
hit(historyLogic)
history = historyLogic.History;
disp(['History: ', num2str(history), '.']);
```

```
History: [1 1 0 0 0 0].
```

### Input Arguments

#### **historyLogic** — Track history logic

`trackHistoryLogic`

Track history logic, specified as a `trackHistoryLogic` object.

### Extended Capabilities

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

trackHistoryLogic

**Introduced in R2021a**

## init

Initialize track logic with first hit

### Syntax

```
init(historyLogic)
```

### Description

`init(historyLogic)` initializes the track history logic with the first hit.

### Examples

#### Initialize History-Based Logic

Create a history-based logic with default confirmation and deletion thresholds.

```
historyLogic = trackHistoryLogic

historyLogic =
    trackHistoryLogic with properties:
        ConfirmationThreshold: [2 3]
        DeletionThreshold: [6 6]
        History: [0 0 0 0 0 0]
```

Initialize the logic, which records a hit as the first update to the logic.

```
init(historyLogic)
history = historyLogic.History;
disp(['History: ', num2str(history), '.']);

History: [1 0 0 0 0 0].
```

### Input Arguments

#### **historyLogic** — Track history logic

`trackHistoryLogic` object

Track history logic, specified as a `trackHistoryLogic` object.

### Extended Capabilities

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

trackHistoryLogic

**Introduced in R2021a**

## miss

Update track logic with miss

### Syntax

```
miss(historyLogic)
```

### Description

`miss(historyLogic)` updates the track history with a miss.

### Examples

#### Update History Logic with Miss

Create a history-based logic with the default confirmation and deletion thresholds.

```
historyLogic = trackHistoryLogic;
```

Initialize the logic, which records a hit as the first update to the logic. The first element of the 'History' property, which indicates the most recent update, is 1.

```
init(historyLogic)
history = historyLogic.History;
disp(['History: ', num2str(history), '.']);
```

```
History: [1 0 0 0 0 0].
```

Update the logic with a miss. The first element of the 'History' property is 0.

```
miss(historyLogic)
history = historyLogic.History;
disp(['History: ', num2str(history), '.']);
```

```
History: [0 1 0 0 0 0].
```

### Input Arguments

#### **historyLogic** — Track history logic

`trackHistoryLogic`

Track history logic, specified as a `trackHistoryLogic` object.

### Extended Capabilities

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

trackHistoryLogic

**Introduced in R2021a**

## output

Get current state of track logic

### Syntax

```
history = output(historyLogic)
```

### Description

`history = output(historyLogic)` returns the recent history updates of the track history logic object, `historyLogic`.

### Examples

#### Get Recent History of History-Based Logic

Create a history-based logic. Specify confirmation threshold values  $M_c$  and  $N_c$  as the vector [3 5]. Specify deletion threshold values  $M_d$  and  $N_d$  as the vector [6 7].

```
historyLogic = trackHistoryLogic('ConfirmationThreshold',[3 5], ...  
    'DeletionThreshold',[6 7]);
```

Get the recent history of the logic. The history vector has a length of 7, which is the greater of  $N_c$  and  $N_d$ . All values are 0 because the logic is not initialized.

```
h = output(historyLogic)
```

```
h = 1x7 logical array
```

```
    0    0    0    0    0    0    0
```

Initialize the logic, then get the recent history of the logic. The first element, which indicates the most recent update, is 1.

```
init(historyLogic);  
h = output(historyLogic)
```

```
h = 1x7 logical array
```

```
    1    0    0    0    0    0    0
```

Update the logic with a hit, then get the recent history of the logic.

```
hit(historyLogic);  
h = output(historyLogic)
```

```
h = 1x7 logical array
```

```
    1    1    0    0    0    0    0
```



## Input Arguments

### **historyLogic** — Track history logic

trackHistoryLogic

Track history logic, specified as a trackHistoryLogic object.

## Output Arguments

### **history** — Recent history

logical vector

Recent track history of historyLogic, returned as a logical vector. The length of the vector is the same as the length of the History property of the historyLogic. The first element is the most recent update. A true value indicates a hit and a false value indicates a miss.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

trackHistoryLogic

**Introduced in R2021a**

## reset

Reset state of track logic

### Syntax

```
reset(logic)
```

### Description

`reset(logic)` resets the track logic object, `logic`.

### Examples

#### Reset Track History Logic

Create a history-based logic using the default confirmation threshold and deletion threshold. Get the current state of the logic. The current and maximum score are both 0.

```
historyLogic = trackHistoryLogic;  
history = output(historyLogic)
```

```
history = 1x6 logical array
```

```
  0  0  0  0  0  0
```

Initialize the logic, then get the current state of the logic.

```
volume = 1.3;  
beta = 0.1;  
init(historyLogic);  
history = output(historyLogic)
```

```
history = 1x6 logical array
```

```
  1  0  0  0  0  0
```

Reset the logic, then get the current state of the logic.

```
reset(historyLogic)  
history = output(historyLogic)
```

```
history = 1x6 logical array
```

```
  0  0  0  0  0  0
```

## Input Arguments

### **logic** — Track history logic

trackHistoryLogic object

Track history logic, specified as a trackHistoryLogic object.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

trackHistoryLogic

**Introduced in R2021a**

## sync

Synchronize trackHistoryLogic objects

### Syntax

```
sync(historyLogic1,historyLogic2)
```

### Description

sync(historyLogic1,historyLogic2) synchronizes historyLogic1 based on historyLogic2 so that they have the same history value.

### Examples

#### Synchronize Two trackHistoryLogic Objects

Create two trackHistoryLogic objects.

```
logic1 = trackHistoryLogic
```

```
logic1 =  
  trackHistoryLogic with properties:  
    ConfirmationThreshold: [2 3]  
    DeletionThreshold: [6 6]  
    History: [0 0 0 0 0 0]
```

```
logic2 = trackHistoryLogic('ConfirmationThreshold',[3 3],'DeletionThreshold',[5 6])
```

```
logic2 =  
  trackHistoryLogic with properties:  
    ConfirmationThreshold: [3 3]  
    DeletionThreshold: [5 6]  
    History: [0 0 0 0 0 0]
```

Initialize logic2 with a hit.

```
init(logic2)  
logic2
```

```
logic2 =  
  trackHistoryLogic with properties:  
    ConfirmationThreshold: [3 3]  
    DeletionThreshold: [5 6]  
    History: [1 0 0 0 0 0]
```

Synchronize logic1 to logic2.

```
sync(logic1,logic2);
logic1

logic1 =
  trackHistoryLogic with properties:

    ConfirmationThreshold: [2 3]
    DeletionThreshold: [6 6]
    History: [1 0 0 0 0 0]
```

## Input Arguments

### **historyLogic1 — Track history logic**

trackHistoryLogic object

Track history logic, specified as a trackHistoryLogic object.

### **historyLogic2 — Track history logic**

trackHistoryLogic object

Track history logic, specified as a trackHistoryLogic object.

### **Introduced in R2021a**

# objectTrack

Single object track report

## Description

objectTrack captures the track information of a single object. objectTrack is the standard output format for trackers.

## Creation

### Syntax

```
track = objectTrack  
track = objectTrack(Name, Value)
```

### Description

track = objectTrack creates an objectTrack object with default property values. An objectTrack object contains information like the age and state of a single track.

track = objectTrack(Name, Value) allows you to set properties using one or more name-value pairs. Enclose each property name in single quotes.

## Properties

### TrackID — Unique track identifier

1 (default) | nonnegative integer

Unique track identifier, specified as a nonnegative integer. This property distinguishes different tracks.

Example: 2

### BranchID — Unique track branch identifier

0 (default) | nonnegative integer

Unique track branch identifier, specified as a nonnegative integer. This property distinguishes different track branches.

Example: 1

### SourceIndex — Index of source track reporting system

1 (default) | nonnegative integer

Index of source track reporting system, specified as a nonnegative integer. This property identifies the source that reports the track.

Example: 3

**UpdateTime — Update time of track**`0` (default) | nonnegative real scalar

Time at which the track was updated by a tracker, specified as a nonnegative real scalar.

Example: 1.2

Data Types: `single` | `double`

**Age — Number of times track was updated**`1` (default) | positive integer

Number of times the track was updated, specified as a positive integer. When a track is initialized, its Age is equal to 1. Any subsequent update with a hit or miss increases the track Age by 1.

Example: 2

**State — Current state of track**`zeros(6,1)` (default) | real-valued  $N$ -element vector

The current state of the track at the UpdateTime, specified as a real-valued  $N$ -element vector, where  $N$  is the dimension of the state. The format of track state depends on the model used to track the object. For example, for 3-D constant velocity model used with `constvel`, the state vector is  $[x; v_x; y; v_y; z; v_z]$ .

Example: `[1 0.2 3 0.2]`

Data Types: `single` | `double`

**StateCovariance — Current state uncertainty covariance of track**`eye(6,6)` (default) | real positive semidefinite symmetric  $N$ -by- $N$  matrix

The current state uncertainty covariance of the track, specified as a real positive semidefinite symmetric  $N$ -by- $N$  matrix, where  $N$  is the dimension of state specified in the State property.

Data Types: `single` | `double`

**StateParameters — Parameters of the track state reference frame**`struct()` (default) | structure | structure array

Parameters of the track state reference frame, specified as a structure or a structure array. Use this property to define the track state reference frame and how to transform the track from the source coordinate system to the fuser coordinate system.

**ObjectClassID — Object class identifier**`0` (default) | nonnegative integer

Object class identifier, specified as a nonnegative integer. This property distinguishes between different user-defined types of objects. For example, you can use 1 for objects of type "car", and 2 for objects of type "pedestrian". 0 is reserved for unknown classification.

Example: 3

**TrackLogic — Track confirmation and deletion logic type**`'History'` (default) | `'Integrated'` | `'Score'`

Confirmation and deletion logic type, specified as:

- 'History' - Track confirmation and deletion is based on the number of times the track has been assigned to a detection in the latest tracker updates.
- 'Score' - Track confirmation and deletion is based on a log-likelihood track score. A high score means that the track is more likely to be valid. A low score means that the track is more likely to be a false alarm.
- 'Integrated' - Track confirmation and deletion is based on the integrated probability of track existence.

**TrackLogicState — State of track logic**

1-by- $M$  logical vector | 1-by-2 real-valued vector | nonnegative scalar

The current state of the track logic type. Based on the logic type specified in the `TrackLogic` property, the logic state is specified as:

- 'History' - A 1-by- $M$  logical vector, where  $M$  is the number of latest track logical states recorded. `true` (1) values indicate hits, and `false` (0) values indicate misses. For example, `[1 0 1 1 1]` represents four hits and one miss in the last five updates. The default value for logic state is 1.
- 'Score' - A 1-by-2 real-valued vector, `[cs, ms]`. `cs` is the current score, and `ms` is the maximum score. The default value is `[0, 0]`.
- 'Integrated' - A nonnegative scalar. The scalar represents the integrated probability of existence of the track. The default value is 0.5.

**IsConfirmed — Indicate if track is confirmed**

`true` (default) | `false`

Indicate if the track is confirmed, specified as `true` or `false`.

Data Types: `logical`

**IsCoasted — Indicate if track is coasted**

`false` (default) | `true`

Indicate if the track is coasted, specified as `true` or `false`. A track is coasted if its latest update is based on prediction instead of correction using detections.

Data Types: `logical`

**IsSelfReported — Indicate if track is self reported**

`true` (default) | `false`

Indicate if the track is self reported, specified as `true` or `false`. A track is self reported if it is reported from internal sources (sensors, trackers, or fusers). To limit the propagation of rumors in a tracking system, use the value `false` if the track was updated by an external source.

Example: `false`

Data Types: `logical`

**ObjectAttributes — Object attributes**

`struct()` (default) | structure

Object attributes passed by the tracker, specified as a structure.



## Object Functions

toStruct Convert objectTrack object to struct

## Examples

### Create Track Report using objectTrack

Create a report of a track using objectTrack.

```
x = (1:6)';  
P = diag(1:6);  
track = objectTrack('State',x,'StateCovariance',P);  
disp(track)
```

objectTrack with properties:

```
    TrackID: 1  
    BranchID: 0  
    SourceIndex: 1  
    UpdateTime: 0  
    Age: 1  
    State: [6x1 double]  
    StateCovariance: [6x6 double]  
    StateParameters: [1x1 struct]  
    ObjectClassID: 0  
    TrackLogic: 'History'  
    TrackLogicState: 1  
    IsConfirmed: 1  
    IsCoasted: 0  
    IsSelfReported: 1  
    ObjectAttributes: [1x1 struct]
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

- The TrackLogic property can only be set during construction.

## See Also

objectDetection

**Introduced in R2021a**

## toStruct

Convert objectTrack object to struct

### Syntax

```
S = toStruct(objTrack)
```

### Description

`S = toStruct(objTrack)` converts an array of objectTrack objects, `objTrack`, to an array of structures whose fields are equivalent to the properties of `objTrack`.

### Examples

#### Convert objectTrack to Struct

Create a report of a track using objectTrack.

```
x = (1:6)';  
P = diag(1:6);  
track = objectTrack('State', x, 'StateCovariance', P)
```

```
track =  
  objectTrack with properties:  
  
      TrackID: 1  
      BranchID: 0  
      SourceIndex: 1  
      UpdateTime: 0  
      Age: 1  
      State: [6x1 double]  
      StateCovariance: [6x6 double]  
      StateParameters: [1x1 struct]  
      ObjectClassID: 0  
      TrackLogic: 'History'  
      TrackLogicState: 1  
      IsConfirmed: 1  
      IsCoasted: 0  
      IsSelfReported: 1  
      ObjectAttributes: [1x1 struct]
```

Convert the track object to a structure.

```
S = toStruct(track)  
  
S = struct with fields:  
      TrackID: 1  
      BranchID: 0  
      SourceIndex: 1  
      UpdateTime: 0
```

```
    Age: 1
    State: [6x1 double]
StateCovariance: [6x6 double]
StateParameters: [1x1 struct]
    ObjectClassID: 0
    TrackLogic: 'History'
TrackLogicState: 1
    IsConfirmed: 1
    IsCoasted: 0
    IsSelfReported: 1
ObjectAttributes: [1x1 struct]
```

## Input Arguments

### **objTrack** — Reports of object track

array of `objectTrack` object

Reports of object tracks, specified as an array of `objectTrack` objects.

## Output Arguments

### **S** — Structures converted from `objectTrack`

array of structure

Structures converted from `objectTrack`, returned as an array of structures. The dimension of the returned structure is same with the dimension of the `objTrack` input. The fields of each structure are equivalent to the properties of `objectTrack`.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`objectTrack`

**Introduced in R2021a**

# objectDetection

Report for single object detection

## Description

An `objectDetection` object contains an object detection report that was obtained by a sensor for a single object. You can use the `objectDetection` output as the input to trackers such as `radarTracker`.

## Creation

### Syntax

```
detection = objectDetection(time, measurement)
detection = objectDetection( ___, Name, Value)
```

### Description

`detection = objectDetection(time, measurement)` creates an object detection at the specified time from the specified measurement.

`detection = objectDetection( ___, Name, Value)` creates a detection object with properties specified as one or more `Name, Value` pair arguments. Any unspecified properties have default values. You cannot specify the `Time` or `Measurement` properties using `Name, Value` pairs.

### Input Arguments

#### **time** — Detection time

nonnegative real scalar

Detection time, specified as a nonnegative real scalar. This argument sets the `Time` property.

#### **measurement** — Object measurement

real-valued  $N$ -element vector

Object measurement, specified as a real-valued  $N$ -element vector.  $N$  is determined by the coordinate system used to report detections and other parameters that you specify in the `MeasurementParameters` property for the `objectDetection` object.

This argument sets the `Measurement` property.

### Output Arguments

#### **detection** — Detection report

`objectDetection` object

Detection report for a single object, returned as an `objectDetection` object. An `objectDetection` object contains these properties:

Property	Definition
Time	Measurement time
Measurement	Object measurements
MeasurementNoise	Measurement noise covariance matrix
SensorIndex	Unique ID of the sensor
ObjectClassID	Object classification
ObjectAttributes	Additional information passed to tracker
MeasurementParameters	Parameters used by initialization functions of nonlinear Kalman tracking filters

## Properties

### Time — Detection time

nonnegative real scalar

Detection time, specified as a nonnegative real scalar. You cannot set this property as a name-value pair. Use the `time` input argument instead.

Example: 5.0

Data Types: double

### Measurement — Object measurement

real-valued  $N$ -element vector

Object measurement, specified as a real-valued  $N$ -element vector. You cannot set this property as a name-value pair. Use the `measurement` input argument instead.

Example: [1.0; -3.4]

Data Types: double | single

### MeasurementNoise — Measurement noise covariance

scalar | real positive semi-definite symmetric  $N$ -by- $N$  matrix

Measurement noise covariance, specified as a scalar or a real positive semi-definite symmetric  $N$ -by- $N$  matrix.  $N$  is the number of elements in the measurement vector. For the scalar case, the matrix is a square diagonal  $N$ -by- $N$  matrix having the same data interpretation as the measurement.

Example: [5.0, 1.0; 1.0, 10.0]

Data Types: double | single

### SensorIndex — Sensor identifier

1 | positive integer

Sensor identifier, specified as a positive integer. The sensor identifier lets you distinguish between different sensors and must be unique to the sensor.

Example: 5

Data Types: double

### ObjectClassID — Object class identifier

0 (default) | positive integer

Object class identifier, specified as a positive integer. Object class identifiers distinguish between different kinds of objects. The value 0 denotes an unknown object type. If the class identifier is nonzero, `radarTracker` immediately creates a confirmed track from the detection.

Example: 1

Data Types: `double`

### MeasurementParameters — Measurement function parameters

{ } (default) | structure array | cell containing structure array | cell array

Measurement function parameters, specified as a structure array, a cell containing a structure array, or a cell array. The property contains all the arguments used by the measurement function specified by the `MeasurementFcn` property of a nonlinear tracking filter such as `trackingEKF` or `trackingUKF`.

The table shows sample fields for the `MeasurementParameters` structures.

Field	Description	Example
Frame	Frame used to report measurements, specified as one of these values: <ul style="list-style-type: none"> <li>'rectangular' — Detections are reported in rectangular coordinates.</li> <li>'spherical' — Detections are reported in spherical coordinates.</li> </ul>	'spherical'
OriginPosition	Position offset of the origin of the frame relative to the parent frame, specified as an [x y z] real-valued vector.	[0 0 0]
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, specified as a [vx vy vz] real-valued vector.	[0 0 0]
Orientation	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	[1 0 0; 0 1 0; 0 0 1]
HasAzimuth	Logical scalar indicating if azimuth is included in the measurement.	1
HasElevation	Logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if <code>HasElevation</code> is false, the reported measurements assume 0 degrees of elevation.	1

Field	Description	Example
HasRange	Logical scalar indicating if range is included in the measurement.	1
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if HasVelocity is false, the measurements are reported as [x y z]. If HasVelocity is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if Orientation performs a frame rotation from the parent coordinate frame to the child coordinate frame. When IsParentToChild is false, then Orientation performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

### ObjectAttributes – Object attributes

{ } (default) | cell array

Object attributes passed through the tracker, specified as a cell array. These attributes are added to the output of the `radarTracker` but not used by the tracker.

Example: `{[10,20,50,100], 'radar1'}`

## Examples

### Create Detection from Position Measurement

Create a detection from a position measurement. The detection is made at a timestamp of one second from a position measurement of [100;250;10] in Cartesian coordinates.

```
detection = objectDetection(1,[100;250;10])
```

```
detection =  
    objectDetection with properties:
```

```
        Time: 1  
        Measurement: [3x1 double]  
        MeasurementNoise: [3x3 double]  
        SensorIndex: 1  
        ObjectClassID: 0  
        MeasurementParameters: { }
```

```
ObjectAttributes: {}
```

### Create Detection With Measurement Noise

Create an `objectDetection` from a time and position measurement. The detection is made at a time of one second for an object position measurement of `[100;250;10]`. Add measurement noise and set other properties using Name-Value pairs.

```
detection = objectDetection(1,[100;250;10],'MeasurementNoise',10, ...  
    'SensorIndex',1,'ObjectAttributes',{'Example object',5})
```

```
detection =  
    objectDetection with properties:  
  
        Time: 1  
        Measurement: [3x1 double]  
        MeasurementNoise: [3x3 double]  
        SensorIndex: 1  
        ObjectClassID: 0  
        MeasurementParameters: {}  
        ObjectAttributes: {'Example object' [5]}
```

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

## See Also

### Objects

`radarTracker` | `radarDataGenerator` | `trackingKF` | `trackingEKF` | `trackingUKF`

**Introduced in R2021a**



# trackingKF

Linear Kalman filter for object tracking

## Description

A `trackingKF` object is a discrete-time linear Kalman filter used to track states, such as positions and velocities of target platforms.

A Kalman filter is a recursive algorithm for estimating the evolving state of a process when measurements are made on the process. The filter assumes the state-space model, including the state model and the measurement model, is linear. When the process noise and measurement noise are Gaussian and the motion model is linear, the Kalman filter is optimal. For a brief description of the linear Kalman filter algorithm, see "Linear Kalman Filters".

You can use a `trackingKF` object in these ways:

- Set the `MotionModel` property to one of predefined state transition models. See the `MotionModel` property for details on these models.
  - "1D Constant Velocity"
  - "1D Constant Acceleration"
  - "2D Constant Velocity"
  - "2D Constant Acceleration"
  - "3D Constant Velocity"
  - "3D Constant Acceleration"
- Explicitly set the motion model. Set the `MotionModel` property to "Custom", and then use the `StateTransitionModel` and `MeasurementModel` properties to specify the state transition matrix and measurement matrix, respectively. Optionally, you can specify control inputs by specifying the `ControlModel` property.

## Creation

### Syntax

```
filter = trackingKF
filter = trackingKF("MotionModel",model)
filter = trackingKF(A,H)
filter = trackingKF(A,H,B)
filter = trackingKF( ___,Name,Value)
```

### Description

`filter = trackingKF` creates a discrete-time linear Kalman filter object for estimating the state of a 2-D, constant-velocity, moving object. The function sets the `MotionModel` property of the filter to "2D Constant Velocity".

`filter = trackingKF("MotionModel",model)` sets the `MotionModel` property to a predefined motion model, `model`. In this case, the filter initializes the state as a double-precision zero vector based on the dimension of the motion model. The filter also configures the `MeasurementModel` property so that the measurement model returns position measurements.

`filter = trackingKF(A,H)` specifies the `StateTransitionModel` and the `MeasurementModel` properties to `A` and `H`, respectively. The function sets the `MotionModel` property to "Custom".

`filter = trackingKF(A,H,B)` sets the `ControlModel` property to the specified `B`. The function sets the `MotionModel` property to "Custom".

`filter = trackingKF( ____,Name,Value)` configures the properties of the Kalman filter by using one or more name-value arguments and any of the previous syntaxes. Any unspecified properties take default values. Enclose each property name in quotes.

## Properties

### State — Kalman filter state

0 (default) | real-valued scalar | real-valued  $M$ -element vector

Kalman filter state, specified as a real-valued  $M$ -element vector, where  $M$  is the size of the state vector. For information on the typical size of the state vector for each motion model, see the `MotionModel` property. If you specify the initial state as a scalar, the filter extends the state to an  $M$ -by-1 vector.

To use the filter with single-precision, floating-point variables, specify the `MotionModel` property as a predefined model and specify `State` as a single-precision vector variable. For example:

```
filter = trackingKF("MotionModel","2D Constant Velocity","State",single([1; 2; 3; 4]))
```

Example: [200; 0.2; -40; -0.01]

Data Types: single | double

### StateCovariance — State estimation error covariance

1 (default) | positive scalar | positive-definite real-valued  $M$ -by- $M$  matrix

State estimation error covariance, specified as a positive scalar or a positive-definite real-valued  $M$ -by- $M$  matrix, where  $M$  is the size of the state vector. If you specify a scalar, the property value is the product of the specified scalar and an  $M$ -by- $M$  identity matrix. The matrix represents the uncertainty in the state, and each diagonal element of the matrix represents the variance of the corresponding state component. The off-diagonal elements represent cross-covariance between different state components.

Example: [20 0.1; 0.1 1]

Data Types: double

### MotionModel — Kalman filter motion model

"Custom" | "1D Constant Velocity" | "2D Constant Velocity" | "3D Constant Velocity" | "1D Constant Acceleration" | "2D Constant Acceleration" | "3D Constant Acceleration"

Kalman filter motion model, specified as "Custom" or one of these predefined models:

- "1D Constant Velocity"
- "1D Constant Acceleration"
- "2D Constant Velocity"
- "2D Constant Acceleration"
- "3D Constant Velocity"
- "3D Constant Acceleration"

If you specify the property as one of the predefined motion models, the filter uses this state-space model:

$$x(k + 1) = A(k)x(k) + G(k)w(k)$$

$$z(k) = H(k)x(k) + v(k)$$

where  $k$  is the discrete time step,  $x$  is the state,  $A$  is the state transition matrix,  $w$  is the process noise,  $G$  is the process noise gain matrix,  $H$  is the measurement matrix,  $v$  is the measurement noise, and  $z$  is the measurement. Note that the size of the gain matrix  $G$  is  $M$ -by- $M/2$ , and the size of the process noise  $w$  is  $M/2$ , where  $M$  is the size of the state  $x$ .

Motion Model	State Vector $x$	State Transition Matrix ( $A$ )	Gain Matrix ( $G$ )
"1D Constant Velocity"	$[x; vx]$	$[1 \ dt; \ 0 \ 1]$	$[dt^2/2; \ dt]$
"2D Constant Velocity"	$[x; vx; y; vy]$	Block diagonal matrix with the $[1 \ dt; \ 0 \ 1]$ block repeated for the $x$ and $y$ spatial dimensions	Kronecker product of $\text{kron}(\text{eye}(2), [dt^2/2; \ dt])$
"3D Constant Velocity"	$[x; vx; y; vy; z; vz]$	Block diagonal matrix with the $[1 \ dt; \ 0 \ 1]$ block repeated for the $x$ , $y$ , and $z$ spatial dimensions.	Kronecker product of $\text{kron}(\text{eye}(3), [dt^2/2; \ dt])$
"1D Constant Acceleration"	$[x; vx; ax]$	$[1 \ dt \ dt^2/2; \ 0 \ 1 \ dt; \ 0 \ 0 \ 1]$	$[dt^2/2; \ dt; \ 1]$
"2D Constant Acceleration"	$[x; vx; ax; y; vy; ay]$	Block diagonal matrix with $[1 \ dt \ dt^2/2; \ 0 \ 1 \ dt; \ 0 \ 0 \ 1]$ blocks repeated for the $x$ and $y$ spatial dimensions	Kronecker product of $\text{kron}(\text{eye}(2), [dt^2/2; \ dt; \ 1])$

Motion Model	State Vector $x$	State Transition Matrix ( $A$ )	Gain Matrix ( $G$ )
"3D Constant Acceleration"	$[x; vx, ax; y; vy; ay; z; vz; az]$	Block diagonal matrix with the $[1 \ dt; 0.5*dt^2; 0 \ 1 \ dt; 0 \ 0 \ 1]$ block repeated for the $x$ , $y$ , and $z$ spatial dimensions	Kronecker product of $\text{kron}(\text{eye}(3), [dt^2/2; dt; 1])$

In the table,  $dt$  is the time step specified in the `predict` object function. If you want process noise and measurement noise values different from the default values for the motion model, specify them in the `ProcessNoise` and `MeasurementNoise` properties, respectively.

If you specify `MotionModel` as "Custom", you must specify a state transition model matrix  $A$  and a measurement model matrix  $H$  as input arguments to the Kalman filter. You can optionally specify a control model matrix,  $B$ , as well. When you specify a custom motion model, the filter uses this state-space model:

$$x(k + 1) = A(k)x(k) + B(k)u(k) + w(k)$$

$$z(k) = H(k)x(k) + v(k)$$

where  $u$  is the control input. In this case, the size of the process noise  $w$  is  $M$ , where  $M$  is the size of the state  $x$ . You can specify the covariance of  $w$  using the `ProcessNoise` property, and specify the covariance of  $v$  using the `MeasurementNoise` property.

Data Types: `char` | `string`

#### **StateTransitionModel** — State transition model between time steps

$[1 \ 1 \ 0 \ 0; 0 \ 1 \ 0 \ 0; 0 \ 0 \ 1 \ 1; 0 \ 0 \ 0 \ 1]$  (default) | real-valued  $M$ -by- $M$  matrix

State transition model between time steps, specified as a real-valued  $M$ -by- $M$  matrix.  $M$  is the size of the state vector. In the absence of controls and noise, the state transition model predicts the state at a time step to the next time step.

Example:  $[1 \ 1; 0 \ 1]$

#### **Dependencies**

To enable this property, set the `MotionModel` property to "Custom".

Data Types: `single` | `double`

#### **ControlModel** — Control model

$M$ -by- $L$  real-valued matrix

Control model, specified as an  $M$ -by- $L$  matrix.  $M$  is the dimension of the state vector, and  $L$  is the number of controls or forces. The control model adds the effect of controls on the evolution of the state.

---

**Note** To use a control model, you must specify this property when constructing the filter object. You cannot change the size of the control model matrix after creating the filter.

---

Example: [.01 0.2]

Data Types: single | double

### ProcessNoise — Covariance of process noise

1 (default) | nonnegative scalar | positive-semidefinite  $D$ -by- $D$  matrix | positive-semidefinite  $M$ -by- $M$  matrix

Covariance of process noise, specified as a nonnegative scalar, a positive-semidefinite  $D$ -by- $D$  matrix, or a positive-semidefinite  $M$ -by- $M$  matrix. Process noise represents the uncertainty of state propagation, and the filter assumes the process noise to be zero-mean Gaussian white noise.

- When the `MotionModel` property is specified as one of the predefined motion models, specify the `ProcessNoise` property as a positive-semidefinite  $D$ -by- $D$  matrix, where  $D$  is the number of dimensions of the target motion. For example,  $D = 2$  for the "2D Constant Velocity" or the "2D Constant Acceleration" motion model.

In this case, if you specify the `ProcessNoise` property as a nonnegative scalar, then the scalar extends to the diagonal elements of a diagonal covariance matrix, of size  $D$ -by- $D$ .

- When the `MotionModel` property is specified as "Custom", specify the `ProcessNoise` property as a positive-semidefinite  $M$ -by- $M$  matrix, where  $M$  is the size of the filter state. For example,  $M = 6$  if you customize a 3-D motion model in which the state is  $(x, v_x, y, v_y, z, v_z)$ .

In this case, if you specify the `ProcessNoise` property as a nonnegative scalar, then the scalar extends to the diagonal elements of a diagonal covariance matrix, of size  $M$ -by- $M$ .

Data Types: single | double

### MeasurementModel — Measurement model from state vector

[1 0 0 0; 0 0 1 0] (default) | real-valued  $N$ -by- $M$  matrix

Measurement model from the state vector, specified as a real-valued  $N$ -by- $M$  matrix, where  $N$  is the size of the measurement vector and  $M$  is the size of the state vector. The measurement model is a linear matrix that determines measurements from the filter state.

---

**Note** You cannot change the size of the measurement model matrix after creating the filter.

---

Example: [1 0.5 0.01; 1.0 1 0]

Data Types: single | double

### MeasurementNoise — Measurement noise covariance

1 (default) | positive scalar | positive-definite real-valued  $N$ -by- $N$  matrix

Covariance of the measurement noise, specified as a positive scalar or a positive-definite, real-valued  $N$ -by- $N$  matrix, where  $N$  is the size of the measurement vector. If you specify this property as a scalar, the property value is the product of the specified scalar and the  $N$ -by- $N$  identity matrix. Measurement noise represents the uncertainty of the measurement and the filter assumes measurement noise to be zero-mean Gaussian white noise.

Example: 0.2

Data Types: single | double

**EnableSmoothing — Enable state smoothing**`false` (default) | `true`

Enable state smoothing, specified as `false` or `true`. Setting this property to `true` requires the Sensor Fusion and Tracking Toolbox license. When specified as `true`, you can:

- Use the `smooth` function, provided in Sensor Fusion and Tracking Toolbox, to smooth state estimates of the previous steps. Internally, the filter stores the results from previous steps to allow backward smoothing.
- Specify the maximum number of smoothing steps using the `MaxNumSmoothingSteps` property of the tracking filter.

**MaxNumSmoothingSteps — Maximum number of smoothing steps**`5` (default) | positive integer

Maximum number of backward smoothing steps, specified as a positive integer.

**Dependencies**

To enable this property, set the `EnableSmoothing` property to `true`.

**MaxNumOOSMSteps — Maximum number of out-of-sequence measurement steps**`0` (default) | nonnegative integer

Maximum number of out-of-sequence measurement (OOSM) steps, specified as a nonnegative integer.

- Setting this property to `0` disables the OOSM retrodiction capability of the filter object.
- Setting this property to a positive integer enables the OOSM retrodiction capability of the filter object. This option requires a Sensor Fusion and Tracking Toolbox license. Also, you cannot set the `MotionModel` property to "Custom". With OOSM enabled, the filter object saves the past state and state covariance history. You can use the `OOSM` and the `retrodict` and `retroCorrect` (or `retroCorrectJPDA` for multiple OOSMs) object functions to reduce the uncertainty of the estimated state.

Increasing the value of this property increases the amount of memory that must be allocated for the state history, but enables you to process OOSMs that arrive after longer delays. Note that the effect of the uncertainty reduction using an OOSM decreases as the delay becomes longer.

**Object Functions**

<code>predict</code>	Predict state and state estimation error covariance of linear Kalman filter
<code>correct</code>	Correct state and state estimation error covariance using tracking filter
<code>correctjpda</code>	Correct state and state estimation error covariance using tracking filter and JPDA
<code>distance</code>	Distances between current and predicted measurements of tracking filter
<code>likelihood</code>	Likelihood of measurement from tracking filter
<code>clone</code>	Create duplicate tracking filter
<code>residual</code>	Measurement residual and residual noise from tracking filter
<code>initialize</code>	Initialize state and covariance of tracking filter

**Examples**

### Create Constant-Velocity Linear Kalman Filter

Create a linear Kalman filter that uses a 2D constant velocity motion model. Assume that the measurement consists of the  $xy$ -location of the object.

Specify the initial state estimate to have zero velocity.

```
x = 5.3;
y = 3.6;
initialState = [x;0;y;0];
KF = trackingKF('MotionModel','2D Constant Velocity','State',initialState);
```

Create measured positions for the object on a constant-velocity trajectory.

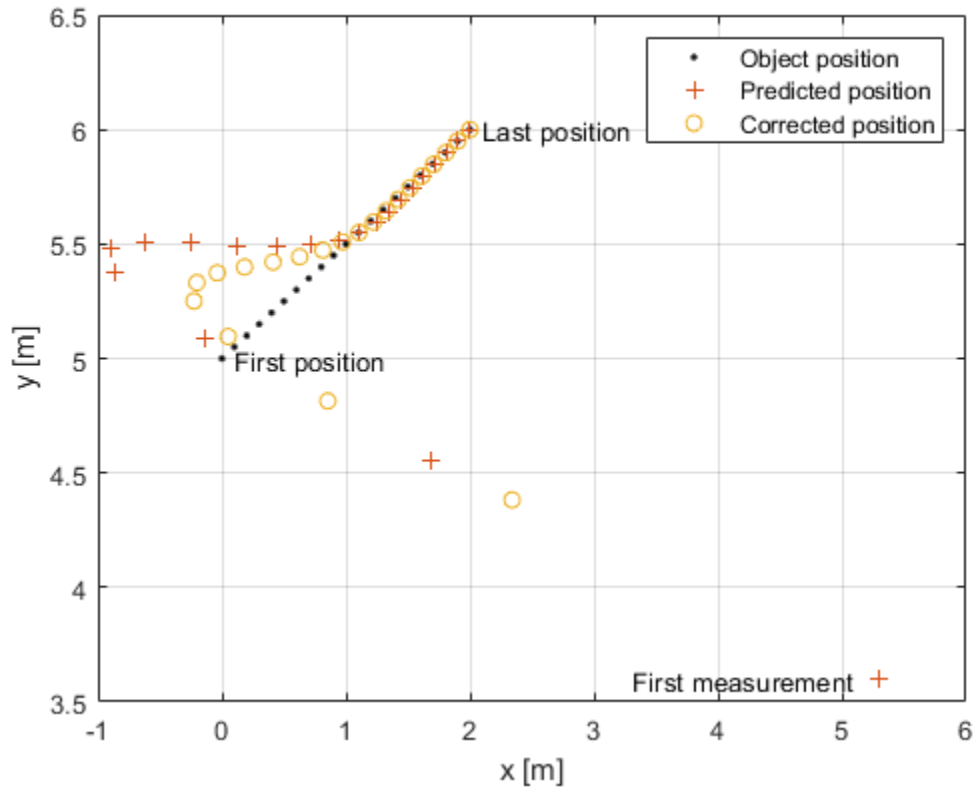
```
vx = 0.2;
vy = 0.1;
T = 0.5;
pos = [0:vx*T:2;
       5:vy*T:6]';
```

Predict and correct the state of the object.

```
for k = 1:size(pos,1)
    pstates(k,:) = predict(KF,T);
    cstates(k,:) = correct(KF,pos(k,:));
end
```

Plot the tracks.

```
plot(pos(:,1),pos(:,2),"k.",pstates(:,1),pstates(:,3),"+", ...
      cstates(:,1),cstates(:,3),"o")
xlabel("x [m]")
ylabel("y [m]")
grid
xt = [x-2, pos(1,1)+0.1, pos(end,1)+0.1];
yt = [y, pos(1,2), pos(end,2)];
text(xt,yt,["First measurement","First position","Last position"])
legend("Object position","Predicted position","Corrected position")
```



### Use Custom trackingKF with Control Inputs

Specify a simulation time of 10 seconds with a time step of 1 second.

```
rng(2021) % For repeatable results
simulationTime = 20;
dt = 1;
tspan = 0:dt:simulationTime;
steps = length(tspan);
```

Specify the motion model as a 2-D constant velocity model with a state of  $[x; vx; y; vy]$ . The measurement is  $[x; y]$ .

```
A1D = [1 dt; 0 1];
A = kron(eye(2),A1D) % State transition model
```

```
A = 4x4
```

```
1    1    0    0
0    1    0    0
0    0    1    1
0    0    0    1
```

```
H1D = [1 0];
H = kron(eye(2),H1D) % Measurement model
```



```
H = 2x4
```

```
    1    0    0    0
    0    0    1    0
```

```
sigma = 0.2;
R = sigma^2*eye(2); % Measurement noise covariance
```

Specify a control model matrix.

```
B1D = [0; 1];
B = kron(eye(2),B1D) % Control model matrix
```

```
B = 4x2
```

```
    0    0
    1    0
    0    0
    0    1
```

Assume the control inputs are sinusoidal on the velocity components,  $v_x$  and  $v_y$ .

```
gain = 5;
Ux = gain*sin(tspan(2:end));
Uy = gain*cos(tspan(2:end));
U =[Ux; Uy]; % Control inputs
```

Assuming the true initial state is  $[1 \ 1 \ 1 \ -1]$ , simulate the system to obtain true states and measurements.

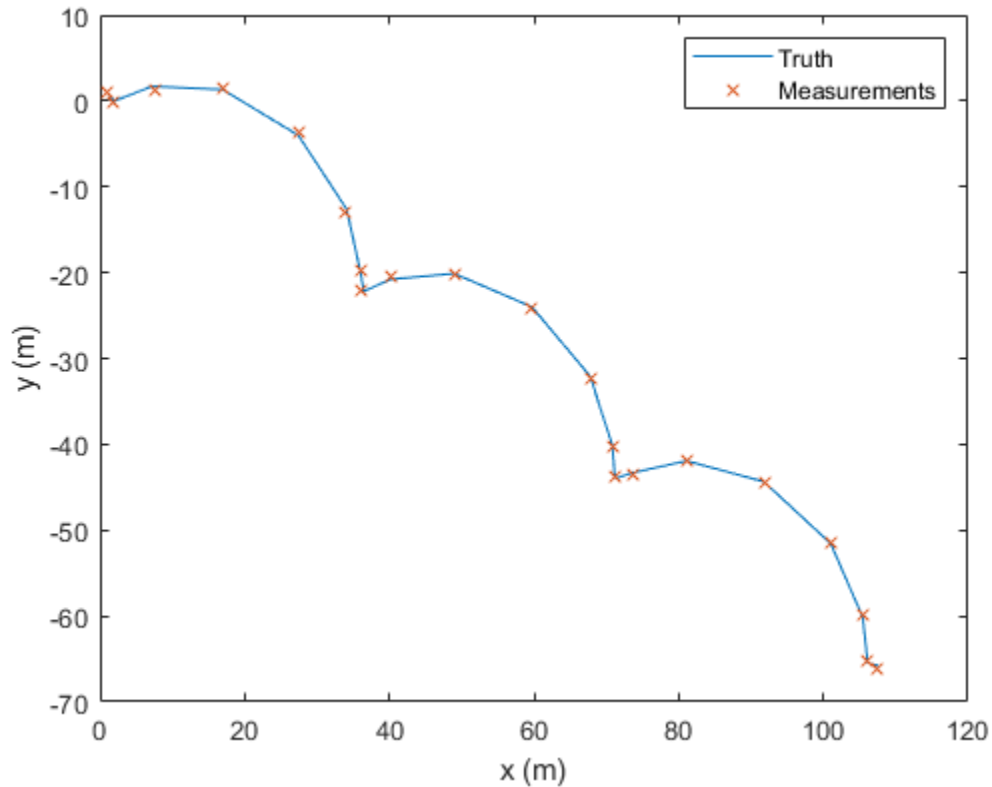
```
initialState = [1 1 1 -1]'; % [m m/s m m/s]
trueStates = NaN(4,steps);
trueStates(:,1) = initialState;

for i=2:steps
    trueStates(:,i) = A*trueStates(:,i-1) + B*U(:,i-1);
end

measurements = H*trueStates + chol(R)*randn(2,steps);
```

Visualize the true trajectory and the measurements.

```
figure
plot(trueStates(1,:),trueStates(3,:), "DisplayName", "Truth")
hold on
plot(measurements(1,:),measurements(2,:), "x", "DisplayName", "Measurements")
xlabel("x (m)")
ylabel("y (m)")
legend
```



Create a `trackingKF` filter with a custom motion model. Enable the control input by specifying the control model. Specify the initial state in the filter based on the first measurement.

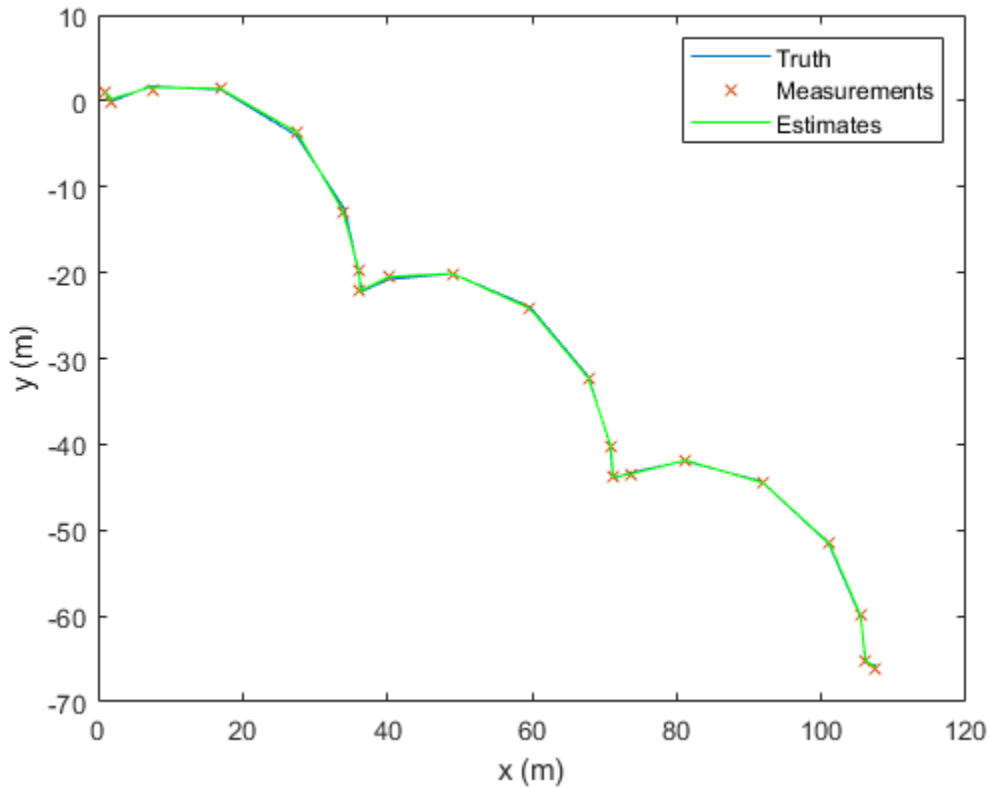
```
initialFilterState = [measurements(1,1); 0; measurements(2,1); 0];
filter = trackingKF("MotionModel", "Custom", ...
    "StateTransitionModel", A, ...
    "MeasurementModel", H, ...
    "ControlModel", B, ...
    "State", initialFilterState);
```

Estimate states by using the `predict` and `correct` object functions.

```
estimateStates = NaN(4, steps);
estimateStates(:, 1) = initialFilterState;
for i = 2:steps
    predict(filter, U(:, i-1));
    estimateStates(:, i) = correct(filter, measurements(:, i));
end
```

Visualize the state estimates.

```
plot(estimateStates(1,:), estimateStates(3,:), "g", "DisplayName", "Estimates");
```



## References

- [1] Brown, R.G. and P.Y.C. Wang. *Introduction to Random Signal Analysis and Applied Kalman Filtering*. 3rd Edition. New York: John Wiley & Sons, 1997.
- [2] Kalman, R. E. "A New Approach to Linear Filtering and Prediction Problems." *Transaction of the ASME-Journal of Basic Engineering*, Vol. 82, Series D, March 1960, pp. 35-45.
- [3] Blackman, Samuel. *Multiple-Target Tracking with Radar Applications*. Artech House. 1986.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- If you create a `trackingKF` object, and you specify the `MotionModel` property as any value other than "Custom", then you must specify the state vector explicitly at construction time using the `State` property. The choice of motion model determines the size of the state vector, but motion models do not specify the data type such as double precision or single precision. Code generation requires both the size and data type.
- In code generation, after cloning the filter, you cannot change its `EnableSmoothing` property.

- In code generation, after calling the filter, you cannot change its `MaxNum00SMSteps` property.
- The filter supports strict single-precision code generation.
- The filter supports non-dynamic memory allocation code generation.

### See Also

#### Functions

`initcvkf` | `initcakf`

#### Objects

`trackingEKF` | `trackingUKF` | `trackingABF` | `radarTracker`

#### Topics

“Linear Kalman Filters”

#### Introduced in R2021a

# trackingABF

Alpha-beta filter for object tracking

## Description

The `trackingABF` object represents an alpha-beta filter designed for object tracking for an object that follows a linear motion model and has a linear measurement model. Linear motion is defined by constant velocity or constant acceleration. Use the filter to predict the future location of an object, to reduce noise for a detected location, or to help associate multiple objects with their tracks.

## Creation

### Syntax

```
abf = trackingABF
abf = trackingABF(Name,Value)
```

### Description

`abf = trackingABF` returns an alpha-beta filter for a discrete time, 2-D constant velocity system. The motion model is named `'2D Constant Velocity'` with the state defined as  $[x; vx; y; vy]$ .

`abf = trackingABF(Name,Value)` specifies the properties of the filter using one or more `Name,Value` pair arguments. Any unspecified properties take default values.

## Properties

### MotionModel — Model of target motion

`'2D Constant Velocity'` (default) | `'1D Constant Velocity'` | `'3D Constant Velocity'` | `'1D Constant Acceleration'` | `'2D Constant Acceleration'` | `'3D Constant Acceleration'`

Model of target motion, specified as a character vector or string. Specifying 1D, 2D, or 3D specifies the dimension of the target's motion. Specifying `Constant Velocity` assumes that the target motion is a constant velocity at each simulation step. Specifying `Constant Acceleration` assumes that the target motion is a constant acceleration at each simulation step.

Data Types: `char` | `string`

### State — Filter state

real-valued  $M$ -element vector | scalar

Filter state, specified as a real-valued  $M$ -element vector. A scalar input is extended to an  $M$ -element vector. The state vector is the concatenated states from each dimension. For example, if `MotionModel` is set to `'3D Constant Acceleration'`, the state vector is in the form:  $[x; x'; x''; y; y'; y''; z; z'; z'']$  where `'` and `''` indicate first and second order derivatives, respectively.

If you want a filter with single-precision floating-point variables, specify `State` as a single-precision vector variable. For example,

```
filter = trackingABF('State',single([1;2;3;4]))
```

Example: [200;0.2;150;0.1;0;0.25]

Data Types: `single` | `double`

### **StateCovariance** — State estimation error covariance

*M*-by-*M* matrix | scalar

State error covariance, specified as an *M*-by-*M* matrix, where *M* is the size of the filter state. A scalar input is extended to an *M*-by-*M* matrix. The covariance matrix represents the uncertainty in the filter state.

Example: `eye(6)`

### **ProcessNoise** — Process noise covariance

*D*-by-*D* matrix | scalar

Process noise covariance, specified as a scalar or a *D*-by-*D* matrix, where *D* is the dimensionality of motion. For example, if `MotionModel` is '2D Constant Velocity', then *D* = 2. A scalar input is extended to a *D*-by-*D* matrix.

Example: [20 0.1; 0.1 1]

### **MeasurementNoise** — Measurement noise covariance

*D*-by-*D* matrix | scalar

Measurement noise covariance, specified as a scalar or a *D*-by-*D* matrix, where *D* is the dimensionality of motion. For example, if `MotionModel` is '2D Constant Velocity', then *D* = 2. A scalar input is extended to a *M*-by-*M* matrix.

Example: [20 0.1; 0.1 1]

### **Coefficients** — Alpha-beta filter coefficients

row vector | scalar

Alpha-beta filter coefficients, specified as a scalar or row vector. A scalar input is extended to a row vector. If you specify constant velocity in the `MotionModel` property, the coefficients are [alpha beta]. If you specify constant acceleration in the `MotionModel` property, the coefficients are [alpha beta gamma].

Example: [20 0.1]

### **EnableSmoothing** — Enable state smoothing

`false` (default) | `true`

Enable state smoothing, specified as `false` or `true`. Setting this property to `true` requires the Sensor Fusion and Tracking Toolbox license. When specified as `true`, you can:

- Use the `smooth` function, provided in Sensor Fusion and Tracking Toolbox, to smooth state estimates of the previous steps. Internally, the filter stores the results from previous steps to allow backward smoothing.
- Specify the maximum number of smoothing steps using the `MaxNumSmoothingSteps` property of the tracking filter.

**MaxNumSmoothingSteps — Maximum number of smoothing steps**

5 (default) | positive integer

Maximum number of backward smoothing steps, specified as a positive integer.

**Dependencies**

To enable this property, set the `EnableSmoothing` property to `true`.

**Object Functions**

<code>predict</code>	Predict state and state estimation error covariance of tracking filter
<code>correct</code>	Correct state and state estimation error covariance using tracking filter
<code>correctjpda</code>	Correct state and state estimation error covariance using tracking filter and JPDA
<code>distance</code>	Distances between current and predicted measurements of tracking filter
<code>likelihood</code>	Likelihood of measurement from tracking filter
<code>clone</code>	Create duplicate tracking filter

**Examples****Run trackingABF Filter**

This example shows how to create and run a `trackingABF` filter. Call the `predict` and `correct` functions to track an object and correct the state estimation based on measurements.

Create the filter. Specify the initial state.

```
state = [1;2;3;4];
abf = trackingABF('State',state);
```

Call `predict` to get the predicted state and covariance of the filter. Use a 0.5 sec time step.

```
[xPred,pPred] = predict(abf, 0.5);
```

Call `correct` with a given measurement.

```
meas = [1;1];
[xCorr,pCorr] = correct(abf, meas);
```

Continue to predict the filter state. Specify the desired time step in seconds if necessary.

```
[xPred,pPred] = predict(abf);           % Predict over 1 second
[xPred,pPred] = predict(abf,2);       % Predict over 2 seconds
```

Modify the filter coefficients and correct again with a new measurement.

```
abf.Coefficients = [0.4 0.2];
[xCorr,pCorr] = correct(abf,[8;14]);
```

**References**

- [1] Blackman, Samuel S. "Multiple-target tracking with radar applications." Dedham, MA, Artech House, Inc., 1986, 463 p. (1986).

[2] Bar-Shalom, Yaakov, X. Rong Li, and Thiagalingam Kirubarajan. *Estimation with applications to tracking and navigation: theory algorithms and software*. John Wiley & Sons, 2004.

### **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

#### **Objects**

trackingKF | trackingEKF | trackingUKF | radarTracker

**Introduced in R2021a**



# trackingEKF

Extended Kalman filter for object tracking

## Description

A `trackingEKF` object is a discrete-time extended Kalman filter used to track dynamical states, such as positions and velocities of targets and objects.

A Kalman filter is a recursive algorithm for estimating the evolving state of a process when measurements are made on the process. The extended Kalman filter can model the evolution of a state when the state follows a nonlinear motion model, when the measurements are nonlinear functions of the state, or when both conditions apply. The extended Kalman filter is based on the linearization of the nonlinear equations. This approach leads to a filter formulation similar to the linear Kalman filter, `trackingKF`.

The process and measurements can have Gaussian noise, which you can include in these ways:

- Add noise to both the process and the measurements. In this case, the sizes of the process noise and measurement noise must match the sizes of the state vector and measurement vector, respectively.
- Add noise in the state transition function, the measurement model function, or in both functions. In these cases, the corresponding noise sizes are not restricted.

See “Extended Kalman Filters” for more details.

## Creation

### Syntax

```
filter = trackingEKF
filter = trackingEKF(transitionfcn,measurementfcn,state)
filter = trackingEKF( ___,Name,Value)
```

### Description

`filter = trackingEKF` creates an extended Kalman filter object for a discrete-time system by using default values for the `StateTransitionFcn`, `MeasurementFcn`, and `State` properties. The process and measurement noises are assumed to be additive.

`filter = trackingEKF(transitionfcn,measurementfcn,state)` specifies the state transition function, `transitionfcn`, the measurement function, `measurementfcn`, and the initial state of the system, `state`.

`filter = trackingEKF( ___,Name,Value)` configures the properties of the extended Kalman filter object by using one or more `Name, Value` pair arguments and any of the previous syntaxes. Any unspecified properties have default values.

## Properties

### State — Kalman filter state

real-valued  $M$ -element vector

Kalman filter state, specified as a real-valued  $M$ -element vector, where  $M$  is the size of the filter state. The value of  $M$  is determined based on the motion model you use. For example, if you use a 2-D constant velocity model specified by `constvel`, in which the state is  $[x; vx; y; vy]$ ,  $M$  is four.

If you want a filter with single-precision floating-point variables, specify `State` as a single-precision vector variable. For example,

```
filter = trackingEKF('State',single([1;2;3;4]))
```

Example: `[200; 0.2]`

Data Types: `single` | `double`

### StateCovariance — State estimation error covariance

positive-definite real-valued  $M$ -by- $M$  matrix

State error covariance, specified as a positive-definite real-valued  $M$ -by- $M$  matrix where  $M$  is the size of the filter state. The covariance matrix represents the uncertainty in the filter state.

Example: `[20 0.1; 0.1 1]`

### StateTransitionFcn — State transition function

function handle

State transition function, specified as a function handle. This function calculates the state vector at time step  $k$  from the state vector at time step  $k - 1$ . The function can take additional input parameters, such as control inputs or time step size. The function can also include noise values. You can use one of these functions as your state transition function.

Function Name	Function Purpose
<code>constvel</code>	Constant-velocity state update model
<code>constacc</code>	Constant-acceleration state update model
<code>constturn</code>	Constant turn-rate state update model

You can also write your own state transition function. The valid syntaxes for the state transition function depend on whether the filter has additive process noise. The table shows the valid syntaxes based on the value of the `HasAdditiveProcessNoise` property.

Valid Syntaxes (HasAdditiveProcessNoise = true)	Valid Syntaxes (HasAdditiveProcessNoise = false)
<code>x(k) = statetransitionfcn(x(k-1))</code> <code>x(k) = statetransitionfcn(x(k-1),parameters)</code> <ul style="list-style-type: none"> <li>• <code>x(k)</code> is the state at time <code>k</code>.</li> <li>• <code>parameters</code> stands for all additional arguments required by the state transition function.</li> </ul>	<code>x(k) = statetransitionfcn(x(k-1),w(k-1))</code> <code>x(k) = statetransitionfcn(x(k-1),w(k-1),dt)</code> <code>x(k) = statetransitionfcn(__,parameters)</code> <ul style="list-style-type: none"> <li>• <code>x(k)</code> is the state at time <code>k</code>.</li> <li>• <code>w(k)</code> is a value for the process noise at time <code>k</code>.</li> <li>• <code>dt</code> is the time step of the <code>trackingEKF</code> filter, specified in the most recent call to the <code>predict</code> function. The <code>dt</code> argument applies when you use the filter within a tracker and call the <code>predict</code> function with the filter to predict the state of the tracker at the next time step. For the nonadditive process noise case, the tracker assumes that you explicitly specify the time step by using this syntax: <code>predict(filter,dt)</code>.</li> <li>• <code>parameters</code> stands for all additional arguments required by the state transition function.</li> </ul>

Example: `@constacc`

Data Types: `function_handle`

### StateTransitionJacobianFcn — Jacobian of state transition function

`function handle`

Jacobian of the state transition function, specified as a function handle. This function has the same input arguments as the state transition function.

The valid syntaxes for the Jacobian of the state transition function depend on whether the filter has additive process noise. The table shows the valid syntaxes based on the value of the `HasAdditiveProcessNoise` property.

Valid Syntaxes (HasAdditiveProcessNoise = true)	Valid Syntaxes (HasAdditiveProcessNoise = false)
<pre>Jx(k) = statejacobianfcn(x(k)) Jx(k) = statejacobianfcn(x(k),parameters)</pre> <ul style="list-style-type: none"> <li>• <math>x(k)</math> is the state at time <math>k</math>.</li> <li>• <math>Jx(k)</math> denotes the Jacobian of the predicted state with respect to the previous state. This Jacobian is an <math>M</math>-by-<math>M</math> matrix at time <math>k</math>. The Jacobian function can take additional input parameters, such as control inputs or time-step size.</li> <li>• <code>parameters</code> stands for all additional arguments required by the Jacobian function, such as control inputs or time-step size.</li> </ul>	<pre>[Jx(k),Jw(k)] = statejacobianfcn(x(k),w(k)) [Jx(k),Jw(k)] = statejacobianfcn(x(k),w(k),dt) [Jx(k),Jw(k)] = statejacobianfcn(__,parameters)</pre> <ul style="list-style-type: none"> <li>• <math>x(k)</math> is the state at time <math>k</math></li> <li>• <math>w(k)</math> is a sample <math>Q</math>-element vector of the process noise at time <math>k</math>. <math>Q</math> is the size of the process noise covariance. The process noise vector in the nonadditive case does not need to have the same dimensions as the state vector.</li> <li>• <math>Jx(k)</math> denotes the Jacobian of the predicted state with respect to the previous state. This Jacobian is an <math>M</math>-by-<math>M</math> matrix at time <math>k</math>. The Jacobian function can take additional input parameters, such as control inputs or time-step size.</li> <li>• <math>Jw(k)</math> denotes the <math>M</math>-by-<math>Q</math> Jacobian of the predicted state with respect to the process noise elements.</li> <li>• <code>dt</code> is the time step of the <code>trackingEKF</code> filter, <code>filter</code>, specified in the most recent call to the <code>predict</code> function. The <code>dt</code> argument applies when you use the filter within a tracker and call the <code>predict</code> function with the filter to predict the state of the tracker at the next time step. For the nonadditive process noise case, the tracker assumes that you explicitly specify the time step by using this syntax: <code>predict(filter,dt)</code>.</li> <li>• <code>parameters</code> stands for all additional arguments required by the Jacobian function, such as control inputs or time-step size.</li> </ul>

If this property is not specified, the Jacobians are computed by numeric differencing at each call of the `predict` function. This computation can increase the processing time and numeric inaccuracy.

Example: `@constaccjac`

Data Types: `function_handle`

### ProcessNoise — Process noise covariance

1 (default) | positive real scalar | positive-definite real-valued matrix

Process noise covariance, specified as a scalar or matrix.

- When `HasAdditiveProcessNoise` is `true`, specify the process noise covariance as a positive real scalar or a positive-definite real-valued  $M$ -by- $M$  matrix.  $M$  is the dimension of the state vector. When specified as a scalar, the matrix is a multiple of the  $M$ -by- $M$  identity matrix.

- When `HasAdditiveProcessNoise` is `false`, specify the process noise covariance as a  $Q$ -by- $Q$  matrix.  $Q$  is the size of the process noise vector.

You must specify `ProcessNoise` before any call to the `predict` function. In later calls to `predict`, you can optionally specify the process noise as a scalar. In this case, the process noise matrix is a multiple of the  $Q$ -by- $Q$  identity matrix.

Example: `[1.0 0.05; 0.05 2]`

### **HasAdditiveProcessNoise – Model additive process noise**

`true` (default) | `false`

Option to model process noise as additive, specified as `true` or `false`. When this property is `true`, process noise is added to the state vector. Otherwise, noise is incorporated into the state transition function.

### **MeasurementFcn – Measurement model function**

function handle

Measurement model function, specified as a function handle. The function accepts the  $M$ -element state vector as inputs and outputs the  $N$ -element measurement vector. You can use one of these functions as your measurement function.

Function Name	Function Purpose
<code>cvmeas</code>	Constant-velocity measurement model
<code>cameas</code>	Constant-acceleration measurement model
<code>ctmeas</code>	Constant turn-rate measurement model

You can also write your own measurement function.

- If `HasAdditiveMeasurementNoise` is `true`, specify the function using one of these syntaxes:

$$z(k) = \text{measurementfcn}(x(k))$$

$$z(k) = \text{measurementfcn}(x(k), \text{parameters})$$

$x(k)$  is the state at time  $k$  and  $z(k)$  is the predicted measurement at time  $k$ . The `parameters` argument stands for all additional arguments required by the measurement function.

- If `HasAdditiveMeasurementNoise` is `false`, specify the function using one of these syntaxes:

$$z(k) = \text{measurementfcn}(x(k), v(k))$$

$$z(k) = \text{measurementfcn}(x(k), v(k), \text{parameters})$$

$x(k)$  is the state at time  $k$  and  $v(k)$  is the measurement noise at time  $k$ . The `parameters` argument stands for all additional arguments required by the measurement function.

- If the `HasMeasurementWrapping` property is `true`, you must additionally return the measurement wrapping bounds, which the filter uses to wrap the measurement residuals, as the second output argument of the measurement function.

$$[z(k), \text{bounds}] = \text{measurementfcn}(\_)$$

The function must return `bounds` as an  $M$ -by-2 real-valued matrix, where  $M$  is the size of  $z(k)$ . In each row, the first and second elements specify the lower and upper bounds, respectively, for the

corresponding measurement variable. You can use `-Inf` or `Inf` to represent that the variable does not have a lower or upper bound.

For example, consider a measurement function that returns the azimuth and range of a platform as `[azimuth; range]`. If the azimuth angle wraps between `-180` and `180` degrees while the range is unbounded and nonnegative, then specify the second output argument of the function as `[-180 180; 0 Inf]`.

Example: `@cameas`

Data Types: `function_handle`

### MeasurementJacobianFcn — Jacobian of measurement function

`function handle`

Jacobian of the measurement function, specified as a function handle. The function has the same input arguments as the measurement function. The function can take additional input parameters, such as sensor position and orientation.

- If `HasAdditiveMeasurementNoise` is `true`, specify the Jacobian function using one of these syntaxes:

$$J_{mx}(k) = \text{measjacobianfcn}(x(k))$$

$$J_{mx}(k) = \text{measjacobianfcn}(x(k), \text{parameters})$$

$x(k)$  is the state at time  $k$ .  $J_{x}(k)$  denotes the  $N$ -by- $M$  Jacobian of the measurement function with respect to the state. The `parameters` argument stands for all arguments required by the measurement function.

- If `HasAdditiveMeasurementNoise` is `false`, specify the Jacobian function using one of these syntaxes:

$$[J_{mx}(k), J_{mv}(k)] = \text{measjacobianfcn}(x(k), v(k))$$

$$[J_{mx}(k), J_{mv}(k)] = \text{measjacobianfcn}(x(k), v(k), \text{parameters})$$

$x(k)$  is the state at time  $k$  and  $v(k)$  is an  $R$ -dimensional sample noise vector.  $J_{mx}(k)$  denotes the  $N$ -by- $M$  Jacobian of the measurement function with respect to the state.  $J_{mv}(k)$  denotes the Jacobian of the  $N$ -by- $R$  measurement function with respect to the measurement noise. The `parameters` argument stands for all arguments required by the measurement function.

If not specified, measurement Jacobians are computed using numerical differencing at each call to the `correct` function. This computation can increase processing time and numerical inaccuracy.

Example: `@cameasjac`

Data Types: `function_handle`

### HasMeasurementWrapping — Wrapping of measurement residuals

`0` (default) | `false` or `0` | `true` or `1`

Wrapping of measurement residuals in the filter, specified as a logical `0` (`false`) or `1` (`true`). When specified as `true`, the measurement function specified in the `MeasurementFcn` property must return two output arguments:

- The first argument is the measurement, returned as an  $M$ -element real-valued vector.

- The second argument is the wrapping bounds, returned as an  $M$ -by-2 real-valued matrix, where  $M$  is the dimension of the measurement. In each row, the first and second elements are the lower and upper bounds for the corresponding measurement variable. You can use `-Inf` or `Inf` to represent that the variable does not have a lower or upper bound.

If you enable this property, the filter wraps the measurement residuals according to the measurement bounds, which helps prevent the filter from divergence caused by incorrect measurement residual values.

These measurement functions have predefined wrapping bounds:

- `cvmeas`
- `cameas`
- `ctmeas`

In these functions, the wrapping bounds are `[-180 180]` degrees for azimuth angle measurements and `[-90 90]` degrees for elevation angle measurements. Other measurements are not bounded.

---

**Note** You can specify this property only when constructing the filter.

---

### MeasurementNoise — Measurement noise covariance

1 (default) | positive scalar | positive-definite real-valued matrix

Measurement noise covariance, specified as a positive scalar or positive-definite real-valued matrix.

- When `HasAdditiveMeasurementNoise` is `true`, specify the measurement noise covariance as a scalar or an  $N$ -by- $N$  matrix.  $N$  is the size of the measurement vector. When specified as a scalar, the matrix is a multiple of the  $N$ -by- $N$  identity matrix.
- When `HasAdditiveMeasurementNoise` is `false`, specify the measurement noise covariance as an  $R$ -by- $R$  matrix.  $R$  is the size of the measurement noise vector.

You must specify `MeasurementNoise` before any call to the `correct` function. After the first call to `correct`, you can optionally specify the measurement noise as a scalar. In this case, the measurement noise matrix is a multiple of the  $R$ -by- $R$  identity matrix.

Example: `0.2`

### HasAdditiveMeasurementNoise — Model additive measurement noise

`true` (default) | `false`

Option to enable additive measurement noise, specified as `true` or `false`. When this property is `true`, noise is added to the measurement. Otherwise, noise is incorporated into the measurement function.

### EnableSmoothing — Enable state smoothing

`false` (default) | `true`

Enable state smoothing, specified as `false` or `true`. Setting this property to `true` requires the Sensor Fusion and Tracking Toolbox license. When specified as `true`, you can:

- Use the `smooth` function, provided in Sensor Fusion and Tracking Toolbox, to smooth state estimates of the previous steps. Internally, the filter stores the results from previous steps to allow backward smoothing.

- Specify the maximum number of smoothing steps using the `MaxNumSmoothingSteps` property of the tracking filter.

### **MaxNumSmoothingSteps — Maximum number of smoothing steps**

5 (default) | positive integer

Maximum number of backward smoothing steps, specified as a positive integer.

#### **Dependencies**

To enable this property, set the `EnableSmoothing` property to `true`.

### **MaxNumOOSMSteps — Maximum number of out-of-sequence measurement steps**

0 (default) | nonnegative integer

Maximum number of out-of-sequence measurement (OOSM) steps, specified as a nonnegative integer.

- Setting this property to 0 disables the OOSM retrodiction capability of the filter object.
- Setting this property to a positive integer enables the OOSM retrodiction capability of the filter object. This option requires a Sensor Fusion and Tracking Toolbox license. With OOSM enabled, the filter object saves the past state and state covariance history. You can use the `OOSM` and the `retrodict` and `retroCorrect` (or `retroCorrectJPDA` for multiple OOSMs) object functions to reduce the uncertainty of the estimated state.

Increasing the value of this property increases the amount of memory that must be allocated for the state history, but enables you to process OOSMs that arrive after longer delays. Note that the effect of the uncertainty reduction using an OOSM decreases as the delay becomes longer.

## **Object Functions**

<code>predict</code>	Predict state and state estimation error covariance of tracking filter
<code>correct</code>	Correct state and state estimation error covariance using tracking filter
<code>correctjpda</code>	Correct state and state estimation error covariance using tracking filter and JPDA
<code>distance</code>	Distances between current and predicted measurements of tracking filter
<code>likelihood</code>	Likelihood of measurement from tracking filter
<code>clone</code>	Create duplicate tracking filter
<code>residual</code>	Measurement residual and residual noise from tracking filter
<code>initialize</code>	Initialize state and covariance of tracking filter

## **Examples**

### **Constant-Velocity Extended Kalman Filter**

Create a two-dimensional `trackingEKF` object and use name-value pairs to define the `StateTransitionJacobianFcn` and `MeasurementJacobianFcn` properties. Use the predefined constant-velocity motion and measurement models and their Jacobians.

```
EKF = trackingEKF(@constvel,@cvmeas,[0;0;0;0], ...
    'StateTransitionJacobianFcn',@constveljac, ...
    'MeasurementJacobianFcn',@cvmeasjac);
```

Run the filter. Use the `predict` and `correct` functions to propagate the state. You may call `predict` and `correct` in any order and as many times you want. Specify the measurement in Cartesian coordinates.



```

measurement = [1;1;0];
[xpred, Ppred] = predict(EKF);
[xcorr, Pcorr] = correct(EKF,measurement);
[xpred, Ppred] = predict(EKF);
[xpred, Ppred] = predict(EKF)

```

```
xpred = 4×1
```

```

1.2500
0.2500
1.2500
0.2500

```

```
Ppred = 4×4
```

```

11.7500    4.7500         0         0
 4.7500    3.7500         0         0
         0         0    11.7500    4.7500
         0         0     4.7500    3.7500

```

## More About

### Filter Parameters

This table relates the filter model parameters to the object properties.  $M$  is the size of the state vector.  $N$  is the size of the measurement vector.

Filter Parameter	Description	Filter Property	Size
$f$	State transition function that specifies the equations of motion of the object. This function determines the state at time $k+1$ as a function of the state and the controls at time $k$ . The state transition function depends on the time-increment of the filter.	StateTransitionFcn	Function returns $M$ -element vector
$h$	Measurement function that specifies how the measurements are functions of the state and measurement noise.	MeasurementFcn	Function returns $N$ -element vector
$x_k$	Estimate of the object state.	State	$M$ -element vector

Filter Parameter	Description	Filter Property	Size
$P_k$	State error covariance matrix representing the uncertainty in the values of the state.	StateCovariance	$M$ -by- $M$ matrix
$Q_k$	Estimate of the process noise covariance matrix at step $k$ . Process noise is a measure of the uncertainty in the dynamic model. It is assumed to be zero-mean white Gaussian noise.	ProcessNoise	$M$ -by- $M$ matrix when HasAdditiveProcessNoise is true. $Q$ -by- $Q$ matrix when HasAdditiveProcessNoise is false
$R_k$	Estimate of the measurement noise covariance at step $k$ . Measurement noise reflects the uncertainty of the measurement. It is assumed to be zero-mean white Gaussian noise.	MeasurementNoise	$N$ -by- $N$ matrix when HasAdditiveMeasurementNoise is true. $R$ -by- $R$ when HasAdditiveMeasurementNoise is false.
$F$	Function determining Jacobian of propagated state with respect to previous state.	StateTransitionJacobianFcn	$M$ -by- $M$ matrix
$H$	Function determining Jacobians of measurement with respect to the state and measurement noise.	MeasurementJacobianFcn	$N$ -by- $M$ for state vector Jacobian and $N$ -by- $R$ for measurement vector Jacobian

## Algorithms

The extended Kalman filter estimates the state of a process governed by this nonlinear stochastic equation:

$$x_{k+1} = f(x_k, u_k, w_k, t)$$

$x_k$  is the state at step  $k$ .  $f()$  is the state transition function. Random noise perturbations,  $w_k$ , can affect the object motion. The filter also supports a simplified form,

$$x_{k+1} = f(x_k, u_k, t) + w_k$$

To use the simplified form, set HasAdditiveProcessNoise to true.

In the extended Kalman filter, the measurements are also general functions of the state:

$$z_k = h(x_k, v_k, t)$$

$h(x_k, v_k, t)$  is the measurement function that determines the measurements as functions of the state. Typical measurements are position and velocity or some function of position and velocity. The measurements can also include noise, represented by  $v_k$ . Again, the filter offers a simpler formulation.

$$z_k = h(x_k, t) + v_k$$

To use the simplified form, set `HasAdditiveMeasurementNoise` to `true`.

These equations represent the actual motion and the actual measurements of the object. However, the noise contribution at each step is unknown and cannot be modeled deterministically. Only the statistical properties of the noise are known.

## References

- [1] Brown, R.G. and P.Y.C. Wang. *Introduction to Random Signal Analysis and Applied Kalman Filtering*. 3rd Edition. New York: John Wiley & Sons, 1997.
- [2] Kalman, R. E. "A New Approach to Linear Filtering and Prediction Problems." *Transactions of the ASME-Journal of Basic Engineering*. Vol. 82, Series D, March 1960, pp. 35-45.
- [3] Blackman, Samuel and R. Popoli. *Design and Analysis of Modern Tracking Systems*. Artech House. 1999.
- [4] Blackman, Samuel. *Multiple-Target Tracking with Radar Applications*. Artech House. 1986.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- In code generation, after cloning the filter, you cannot change its `EnableSmoothing` property.
- In code generation, after calling the filter, you cannot change its `MaxNum00SMSSteps` property.
- The filter supports strict single-precision code generation when the specified state transition function and measurement function both support single-precision code generation.
- The filter supports non-dynamic memory allocation code generation.

## See Also

### Functions

`constacc` | `constaccjac` | `cameas` | `cameasjac` | `constturn` | `constturnjac` | `ctmeas` | `ctmeasjac` | `constvel` | `constveljac` | `cvmeas` | `cvmeasjac` | `initcaekf` | `initcvekf` | `initctekf`

### Objects

`trackingKF` | `trackingUKF` | `trackingABF` | `radarTracker`

### Topics

"Extended Kalman Filters"

**Introduced in R2021a**

# trackingUKF

Unscented Kalman filter for object tracking

## Description

The `trackingUKF` object is a discrete-time unscented Kalman filter used to track the positions and velocities of targets and objects.

An unscented Kalman filter is a recursive algorithm for estimating the evolving state of a process when measurements are made on the process. The unscented Kalman filter can model the evolution of a state that obeys a nonlinear motion model. The measurements can also be nonlinear functions of the state, and the process and measurements can have noise.

Use an unscented Kalman filter when one of both of these conditions apply:

- The current state is a nonlinear function of the previous state.
- The measurements are nonlinear functions of the state.

The unscented Kalman filter estimates the uncertainty about the state, and its propagation through the nonlinear state and measurement equations, by using a fixed number of sigma points. Sigma points are chosen by using the unscented transformation, as parameterized by the Alpha, Beta, and Kappa properties.

## Creation

### Syntax

```
filter = trackingUKF
filter = trackingUKF(transitionfcn,measurementfcn,state)
filter = trackingUKF( ____,Name,Value)
```

### Description

`filter = trackingUKF` creates an unscented Kalman filter object for a discrete-time system by using default values for the `StateTransitionFcn`, `MeasurementFcn`, and `State` properties. The process and measurement noises are assumed to be additive.

`filter = trackingUKF(transitionfcn,measurementfcn,state)` specifies the state transition function, `transitionfcn`, the measurement function, `measurementfcn`, and the initial state of the system, `state`.

`filter = trackingUKF( ____,Name,Value)` configures the properties of the unscented Kalman filter object using one or more `Name, Value` pair arguments and any of the previous syntaxes. Any unspecified properties have default values.

## Properties

### State — Kalman filter state

real-valued  $M$ -element vector

Kalman filter state, specified as a real-valued  $M$ -element vector, where  $M$  is the size of the filter state.

If you want a filter with single-precision floating-point variables, specify `State` as a single-precision vector variable. For example,

```
filter = trackingUKF('State',single([1;2;3;4]))
```

Example: [200; 0.2]

Data Types: `single` | `double`

### StateCovariance — State estimation error covariance

positive-definite real-valued  $M$ -by- $M$  matrix

State error covariance, specified as a positive-definite real-valued  $M$ -by- $M$  matrix where  $M$  is the size of the filter state. The covariance matrix represents the uncertainty in the filter state.

Example: [20 0.1; 0.1 1]

### StateTransitionFcn — State transition function

function handle

State transition function, specified as a function handle. This function calculates the state vector at time step  $k$  from the state vector at time step  $k - 1$ . The function can take additional input parameters, such as control inputs or time step size. The function can also include noise values. You can use one of these functions as your state transition function.

Function Name	Function Purpose
<code>constvel</code>	Constant-velocity state update model
<code>constacc</code>	Constant-acceleration state update model
<code>constturn</code>	Constant turn-rate state update model

You can also write your own state transition function. The valid syntaxes for the state transition function depend on whether the filter has additive process noise. The table shows the valid syntaxes based on the value of the `HasAdditiveProcessNoise` property.

Valid Syntaxes (HasAdditiveProcessNoise = true)	Valid Syntaxes (HasAdditiveProcessNoise = false)
<code>x(k) = statetransitionfcn(x(k-1))</code> <code>x(k) = statetransitionfcn(x(k-1),parameters)</code> <ul style="list-style-type: none"> <li>• <code>x(k)</code> is the state at time <code>k</code>.</li> <li>• <code>parameters</code> stands for all additional arguments required by the state transition function.</li> </ul>	<code>x(k) = statetransitionfcn(x(k-1),w(k-1))</code> <code>x(k) = statetransitionfcn(x(k-1),w(k-1),dt)</code> <code>x(k) = statetransitionfcn(__,parameters)</code> <ul style="list-style-type: none"> <li>• <code>x(k)</code> is the state at time <code>k</code>.</li> <li>• <code>w(k)</code> is a value for the process noise at time <code>k</code>.</li> <li>• <code>dt</code> is the time step of the <code>trackingUKF</code> filter, specified in the most recent call to the <code>predict</code> function. The <code>dt</code> argument applies when you use the filter within a tracker and call the <code>predict</code> function with the filter to predict the state of the tracker at the next time step. For the nonadditive process noise case, the tracker assumes that you explicitly specify the time step by using this syntax: <code>predict(filter,dt)</code>.</li> <li>• <code>parameters</code> stands for all additional arguments required by the state transition function.</li> </ul>

Example: @constacc

Data Types: function\_handle

### ProcessNoise — Process noise covariance

1 (default) | positive real scalar | positive-definite real-valued matrix

Process noise covariance, specified as a scalar or matrix.

- When `HasAdditiveProcessNoise` is `true`, specify the process noise covariance as a positive real scalar or a positive-definite real-valued  $M$ -by- $M$  matrix.  $M$  is the dimension of the state vector. When specified as a scalar, the matrix is a multiple of the  $M$ -by- $M$  identity matrix.
- When `HasAdditiveProcessNoise` is `false`, specify the process noise covariance as a  $Q$ -by- $Q$  matrix.  $Q$  is the size of the process noise vector.

You must specify `ProcessNoise` before any call to the `predict` function. In later calls to `predict`, you can optionally specify the process noise as a scalar. In this case, the process noise matrix is a multiple of the  $Q$ -by- $Q$  identity matrix.

Example: [1.0 0.05; 0.05 2]

### HasAdditiveProcessNoise — Model additive process noise

true (default) | false

Option to model process noise as additive, specified as `true` or `false`. When this property is `true`, process noise is added to the state vector. Otherwise, noise is incorporated into the state transition function.

### MeasurementFcn — Measurement model function

function handle

Measurement model function, specified as a function handle. The function accepts the  $M$ -element state vector as inputs and outputs the  $N$ -element measurement vector. You can use one of these functions as your measurement function.

Function Name	Function Purpose
cvmeas	Constant-velocity measurement model
cameas	Constant-acceleration measurement model
ctmeas	Constant turn-rate measurement model

You can also write your own measurement function.

- If `HasAdditiveMeasurementNoise` is `true`, specify the function using one of these syntaxes:

```
z(k) = measurementfcn(x(k))
```

```
z(k) = measurementfcn(x(k),parameters)
```

$x(k)$  is the state at time  $k$  and  $z(k)$  is the predicted measurement at time  $k$ . The `parameters` argument stands for all additional arguments required by the measurement function.

- If `HasAdditiveMeasurementNoise` is `false`, specify the function using one of these syntaxes:

```
z(k) = measurementfcn(x(k),v(k))
```

```
z(k) = measurementfcn(x(k),v(k),parameters)
```

$x(k)$  is the state at time  $k$  and  $v(k)$  is the measurement noise at time  $k$ . The `parameters` argument stands for all additional arguments required by the measurement function.

- If the `HasMeasurementWrapping` property is `true`, you must additionally return the measurement wrapping bounds, which the filter uses to wrap the measurement residuals, as the second output argument of the measurement function.

```
[z(k),bounds] = measurementfcn(__)
```

The function must return `bounds` as an  $M$ -by-2 real-valued matrix, where  $M$  is the size of  $z(k)$ . In each row, the first and second elements specify the lower and upper bounds, respectively, for the corresponding measurement variable. You can use `-Inf` or `Inf` to represent that the variable does not have a lower or upper bound.

For example, consider a measurement function that returns the azimuth and range of a platform as `[azimuth; range]`. If the azimuth angle wraps between  $-180$  and  $180$  degrees while the range is unbounded and nonnegative, then specify the second output argument of the function as `[-180 180; 0 Inf]`.

Example: `@cameas`

Data Types: `function_handle`

### HasMeasurementWrapping — Wrapping of measurement residuals

0 (default) | `false` or 0 | `true` or 1

Wrapping of measurement residuals in the filter, specified as a logical 0 (`false`) or 1 (`true`). When specified as `true`, the measurement function specified in the `MeasurementFcn` property must return two output arguments:



- The first argument is the measurement, returned as an  $M$ -element real-valued vector.
- The second argument is the wrapping bounds, returned as an  $M$ -by-2 real-valued matrix, where  $M$  is the dimension of the measurement. In each row, the first and second elements are the lower and upper bounds for the corresponding measurement variable. You can use `-Inf` or `Inf` to represent that the variable does not have a lower or upper bound.

If you enable this property, the filter wraps the measurement residuals according to the measurement bounds, which helps prevent the filter from divergence caused by incorrect measurement residual values.

These measurement functions have predefined wrapping bounds:

- `cvmeas`
- `cameas`
- `ctmeas`

In these functions, the wrapping bounds are `[-180 180]` degrees for azimuth angle measurements and `[-90 90]` degrees for elevation angle measurements. Other measurements are not bounded.

---

**Note** You can specify this property only when constructing the filter.

---

### MeasurementNoise — Measurement noise covariance

1 (default) | positive scalar | positive-definite real-valued matrix

Measurement noise covariance, specified as a positive scalar or positive-definite real-valued matrix.

- When `HasAdditiveMeasurementNoise` is `true`, specify the measurement noise covariance as a scalar or an  $N$ -by- $N$  matrix.  $N$  is the size of the measurement vector. When specified as a scalar, the matrix is a multiple of the  $N$ -by- $N$  identity matrix.
- When `HasAdditiveMeasurementNoise` is `false`, specify the measurement noise covariance as an  $R$ -by- $R$  matrix.  $R$  is the size of the measurement noise vector.

You must specify `MeasurementNoise` before any call to the `correct` function. After the first call to `correct`, you can optionally specify the measurement noise as a scalar. In this case, the measurement noise matrix is a multiple of the  $R$ -by- $R$  identity matrix.

Example: `0.2`

### HasAdditiveMeasurementNoise — Model additive measurement noise

`true` (default) | `false`

Option to enable additive measurement noise, specified as `true` or `false`. When this property is `true`, noise is added to the measurement. Otherwise, noise is incorporated into the measurement function.

### Alpha — Sigma point spread around state

`1.0e-3` (default) | positive scalar greater than 0 and less than or equal to 1

Sigma point spread around state, specified as a positive scalar greater than 0 and less than or equal to 1.

### Beta — Distribution of sigma points

2 (default) | nonnegative scalar

Distribution of sigma points, specified as a nonnegative scalar. This parameter incorporates knowledge of the noise distribution of states for generating sigma points. For Gaussian distributions, setting `Beta` to 2 is optimal.

### **Kappa — Secondary scaling factor for generating sigma points**

0 (default) | scalar from 0 to 3

Secondary scaling factor for generation of sigma points, specified as a scalar from 0 to 3. This parameter helps specify the generation of sigma points.

### **EnableSmoothing — Enable state smoothing**

false (default) | true

Enable state smoothing, specified as `false` or `true`. Setting this property to `true` requires the Sensor Fusion and Tracking Toolbox license. When specified as `true`, you can:

- Use the `smooth` function, provided in Sensor Fusion and Tracking Toolbox, to smooth state estimates of the previous steps. Internally, the filter stores the results from previous steps to allow backward smoothing.
- Specify the maximum number of smoothing steps using the `MaxNumSmoothingSteps` property of the tracking filter.

### **MaxNumSmoothingSteps — Maximum number of smoothing steps**

5 (default) | positive integer

Maximum number of backward smoothing steps, specified as a positive integer.

### **Dependencies**

To enable this property, set the `EnableSmoothing` property to `true`.

## **Object Functions**

<code>predict</code>	Predict state and state estimation error covariance of tracking filter
<code>correct</code>	Correct state and state estimation error covariance using tracking filter
<code>correctjpda</code>	Correct state and state estimation error covariance using tracking filter and JPDA
<code>distance</code>	Distances between current and predicted measurements of tracking filter
<code>likelihood</code>	Likelihood of measurement from tracking filter
<code>clone</code>	Create duplicate tracking filter
<code>residual</code>	Measurement residual and residual noise from tracking filter
<code>initialize</code>	Initialize state and covariance of tracking filter

## **Examples**

### **Constant-Velocity Unscented Kalman Filter**

Create a `trackingUKF` object using the predefined constant-velocity motion model, `constvel`, and the associated measurement model, `cvmeas`. These models assume that the state vector has the form `[x;vx;y;vy]` and that the position measurement is in Cartesian coordinates, `[x;y;z]`. Set the sigma point spread property to `1e-2`.

```
filter = trackingUKF(@constvel,@cvmeas,[0;0;0;0], 'Alpha',1e-2);
```

Run the filter. Use the `predict` and `correct` functions to propagate the state. You can call `predict` and `correct` in any order and as many times as you want.

```
meas = [1;1;0];
[xpred, Ppred] = predict(filter);
[xcorr, Pcorr] = correct(filter,meas);
[xpred, Ppred] = predict(filter);
[xpred, Ppred] = predict(filter)
```

xpred = 4×1

```
1.2500
0.2500
1.2500
0.2500
```

Ppred = 4×4

```
11.7500    4.7500   -0.0000    0.0000
 4.7500    3.7500    0.0000   -0.0000
-0.0000    0.0000   11.7500    4.7500
 0.0000   -0.0000    4.7500    3.7500
```

## More About

### Filter Parameters

This table relates the filter model parameters to the object properties.  $M$  is the size of the state vector.  $N$  is the size of the measurement vector.

Model Parameter	Description	Filter Property	Size
$f$	State transition function that specifies the equations of motion of the object. This function determines the state at time $k+1$ as a function of the state and the controls at time $k$ . The state transition function depends on the time-increment of the filter.	StateTransitionFcn	Function returns $M$ -element vector
$h$	Measurement function that specifies how the measurements are functions of the state and measurement noise.	MeasurementFcn	Function returns $N$ -element vector
$x_k$	Estimate of the object state.	State	$M$

Model Parameter	Description	Filter Property	Size
$P_k$	State error covariance matrix representing the uncertainty in the values of the state	StateCovariance	$M$ -by- $M$
$Q_k$	Estimate of the process noise covariance matrix at step $k$ . Process noise is measure of the uncertainty in your dynamic model and is assumed to be zero-mean white Gaussian noise	ProcessNoise	$M$ -by- $M$ when HasAdditiveProcessNoise is true. $Q$ -by- $Q$ when HasAdditiveProcessNoise is false.
$R_k$	Estimate of the measurement noise covariance at step $k$ . Measurement noise reflects the uncertainty of the measurement and is assumed to be zero-mean white Gaussian noise.	MeasurementNoise	$N$ -by- $N$ when HasAdditiveMeasurementNoise is true. $R$ -by- $R$ when HasAdditiveMeasurementNoise is false.
$\alpha$	Determines spread of sigma points.	Alpha	scalar
$\beta$	<i>A priori</i> knowledge of sigma point distribution.	Beta	scalar
$\kappa$	Secondary scaling parameter.	Kappa	scalar

## Algorithms

The unscented Kalman filter estimates the state of a process governed by a nonlinear stochastic equation

$$x_{k+1} = f(x_k, u_k, w_k, t)$$

where  $x_k$  is the state at step  $k$ .  $f()$  is the state transition function,  $u_k$  are the controls on the process. The motion may be affected by random noise perturbations,  $w_k$ . The filter also supports a simplified form,

$$x_{k+1} = f(x_k, u_k, t) + w_k$$

To use the simplified form, set `HasAdditiveProcessNoise` to `true`.

In the unscented Kalman filter, the measurements are also general functions of the state,

$$z_k = h(x_k, v_k, t)$$

where  $h(x_k, v_k, t)$  is the measurement function that determines the measurements as functions of the state. Typical measurements are position and velocity or some function of these. The measurements can include noise as well, represented by  $v_k$ . Again the class offers a simpler formulation

$$z_k = h(x_k, t) + v_k$$

To use the simplified form, set `HasAdditiveMeasurementNoise` to `true`.

These equations represent the actual motion of the object and the actual measurements. However, the noise contribution at each step is unknown and cannot be modeled exactly. Only statistical properties of the noise are known.

## References

- [1] Brown, R.G. and P.Y.C. Wang. *Introduction to Random Signal Analysis and Applied Kalman Filtering*. 3rd Edition. New York: John Wiley & Sons, 1997.
- [2] Kalman, R. E. "A New Approach to Linear Filtering and Prediction Problems." *Transactions of the ASME-Journal of Basic Engineering*. Vol. 82, Series D, March 1960, pp. 35-45.
- [3] Wan, Eric A. and R. van der Merwe. "The Unscented Kalman Filter for Nonlinear Estimation". *Adaptive Systems for Signal Processing, Communications, and Control*. AS-SPCC, IEEE, 2000, pp.153-158.
- [4] Wan, Merle. "The Unscented Kalman Filter." In *Kalman Filtering and Neural Networks*. Edited by Simon Haykin. John Wiley & Sons, Inc., 2001.
- [5] Sarkka S. "Recursive Bayesian Inference on Stochastic Differential Equations." Doctoral Dissertation. Helsinki University of Technology, Finland. 2006.
- [6] Blackman, Samuel. *Multiple-Target Tracking with Radar Applications*. Artech House, 1986.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code uses an algorithm that is different from the algorithm that the `trackingUKF` object uses. You might see some numerical differences in the results obtained using the two methods.
- The filter supports strict single-precision code generation when the specified state transition function and measurement function both support single-precision code generation.
- The filter supports non-dynamic memory allocation code generation.

## See Also

### Functions

`constacc` | `constaccjac` | `cameas` | `cameasjac` | `constturn` | `constturnjac` | `ctmeas` | `ctmeasjac` | `constvel` | `constveljac` | `cvmeas` | `cvmeasjac` | `initcaukf` | `initcvukf` | `initctukf`

**Objects**

trackingKF | trackingEKF | trackingABF | radarTracker

**Introduced in R2021a**

# clone

Create duplicate tracking filter

## Syntax

```
filterClone = clone(filter)
```

## Description

`filterClone = clone(filter)` creates a copy of a tracking filter that has the same property values as the original filter.

## Input Arguments

### **filter** — Filter for object tracking

`trackingKF` object | `trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingABF` — Alpha-beta filter

## Output Arguments

### **filterClone** — Cloned filter

`tracking` filter object

Cloned filter, returned as a tracking filter object of the same type as `filter`. The cloned filter has the same properties as the original filter.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## See Also

`correct` | `correctjpd` | `distance` | `initialize` | `likelihood` | `predict` | `residual`

**Introduced in R2021a**

## correct

Correct state and state estimation error covariance using tracking filter

### Syntax

```
[xcorr,Pcorr] = correct(filter,zmeas)
[xcorr,Pcorr] = correct(filter,zmeas,measparams)
[xcorr,Pcorr] = correct(filter,zmeas,zcov)
[xcorr,Pcorr,zcorr] = correct(filter,zmeas)
[xcorr,Pcorr,zcorr] = correct(filter,zmeas,zcov)

correct(filter, ___)
xcorr = correct(filter, ___)
```

### Description

`[xcorr,Pcorr] = correct(filter,zmeas)` returns the corrected state, `xcorr`, and the corrected state estimation error covariance, `Pcorr`, for the next time step of the input tracking filter based on the current measurement, `zmeas`. The corrected values overwrite the internal state and state estimation error covariance of `filter`.

`[xcorr,Pcorr] = correct(filter,zmeas,measparams)` specifies additional parameters used by the measurement function that is defined in the `MeasurementFcn` property of `filter`. You can return any of the outputs from preceding syntaxes.

If `filter` is a `trackingKF` or `trackingABF` object, then you cannot use this syntax.

`[xcorr,Pcorr] = correct(filter,zmeas,zcov)` specifies additional measurement covariance, `zcov`, used in the `MeasurementNoise` property of `filter`.

You can use this syntax only when `filter` is a `trackingKF` object.

`[xcorr,Pcorr,zcorr] = correct(filter,zmeas)` also returns the correction of measurements, `zcorr`.

You can use this syntax only when `filter` is a `trackingABF` object.

`[xcorr,Pcorr,zcorr] = correct(filter,zmeas,zcov)` returns the correction of measurements, `zcorr`, and also specifies additional measurement covariance, `zcov`, used in the `MeasurementNoise` property of `filter`.

You can use this syntax only when `filter` is a `trackingABF` object.

`correct(filter, ___)` updates `filter` with the corrected state and state estimation error covariance without returning the corrected values. Specify the tracking filter and any of the input argument combinations from preceding syntaxes.

`xcorr = correct(filter, ___)` updates `filter` with the corrected state and state estimation error covariance but returns only the corrected state, `xcorr`.



## Examples

### Constant-Velocity Extended Kalman Filter

Create a two-dimensional `trackingEKF` object and use name-value pairs to define the `StateTransitionJacobianFcn` and `MeasurementJacobianFcn` properties. Use the predefined constant-velocity motion and measurement models and their Jacobians.

```
EKF = trackingEKF(@constvel,@cvmeas,[0;0;0;0], ...
    'StateTransitionJacobianFcn',@constveljac, ...
    'MeasurementJacobianFcn',@cvmeasjac);
```

Run the filter. Use the `predict` and `correct` functions to propagate the state. You may call `predict` and `correct` in any order and as many times you want. Specify the measurement in Cartesian coordinates.

```
measurement = [1;1;0];
[xpred, Ppred] = predict(EKF);
[xcorr, Pcorr] = correct(EKF,measurement);
[xpred, Ppred] = predict(EKF);
[xpred, Ppred] = predict(EKF)
```

```
xpred = 4×1
```

```
    1.2500
    0.2500
    1.2500
    0.2500
```

```
Ppred = 4×4
```

```
    11.7500    4.7500         0         0
     4.7500    3.7500         0         0
         0         0    11.7500    4.7500
         0         0     4.7500    3.7500
```

## Input Arguments

### **filter** — Filter for object tracking

`trackingKF` object | `trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingABF` — Alpha-beta filter

### **zmeas** — Measurement of filter

vector | matrix

Measurement of the tracked object, specified as a vector or matrix.

Data Types: `single` | `double`

### **measparams — Measurement parameters**

comma-separated list of arguments

Measurement function arguments, specified as a comma-separated list of arguments. These arguments are the same ones that are passed into the measurement function specified by the `MeasurementFcn` property of the tracking filter. If `filter` is a `trackingKF` or `trackingABF` object, then you cannot specify `measparams`.

Suppose you set `MeasurementFcn` to `@cameas`, and then call `correct`:

```
[xcorr,Pcorr] = correct(filter,frame,sensorpos,sensorvel)
```

The `correct` function internally calls the following:

```
meas = cameas(state,frame,sensorpos,sensorvel)
```

### **zcov — Measurement covariance**

$M$ -by- $M$  matrix

Measurement covariance, specified as an  $M$ -by- $M$  matrix, where  $M$  is the dimension of the measurement. The same measurement covariance matrix is assumed for all measurements in `zmeas`.

Data Types: `single` | `double`

## **Output Arguments**

### **xcorr — Corrected state of filter**

vector | matrix

Corrected state of the filter, specified as a vector or matrix. The `State` property of the input `filter` is overwritten with this value.

### **Pcorr — Corrected state covariance of filter**

vector | matrix

Corrected state covariance of the filter, specified as a vector or matrix. The `StateCovariance` property of the input `filter` is overwritten with this value.

### **zcorr — Corrected measurement of filter**

vector | matrix

Corrected measurement of the filter, specified as a vector or matrix. You can return `zcorr` only when `filter` is a `trackingABF` object.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`clone` | `correctjpd` | `distance` | `initialize` | `likelihood` | `predict` | `residual`

**Introduced in R2021a**

## correctjpda

Correct state and state estimation error covariance using tracking filter and JPDA

### Syntax

```
[xcorr,Pcorr] = correctjpda(filter,zmeas,jpdacoeffs)
```

```
[xcorr,Pcorr] = correctjpda(filter,zmeas,jpdacoeffs,measparams)
```

```
[xcorr,Pcorr] = correctjpda(filter,zmeas,jpdacoeffs,zcov)
```

```
[xcorr,Pcorr,zcorr] = correctjpda(filter,zmeas,jpdacoeffs)
```

```
[xcorr,Pcorr,zcorr] = correctjpda(filter,zmeas,jpdacoeffs,zcov)
```

### Description

`[xcorr,Pcorr] = correctjpda(filter,zmeas,jpdacoeffs)` returns the corrected state, `xcorr`, and the corrected state estimation error covariance, `Pcorr`, for the next time step of the input tracking filter. The corrected values are based on a set of measurements, `zmeas`, and their joint probabilistic data association coefficients, `jpdacoeffs`. These values overwrite the internal state and state estimation error covariance of `filter`.

`[xcorr,Pcorr] = correctjpda(filter,zmeas,jpdacoeffs,measparams)` specifies additional parameters used by the measurement function that is defined in the `MeasurementFcn` property of the tracking filter object.

If `filter` is a `trackingKF` or `trackingABF` object, then you cannot use this syntax.

`[xcorr,Pcorr] = correctjpda(filter,zmeas,jpdacoeffs,zcov)` specifies additional measurement covariance, `zcov`, used in the `MeasurementNoise` property of `filter`.

You can use this syntax only when `filter` is a `trackingKF` object.

`[xcorr,Pcorr,zcorr] = correctjpda(filter,zmeas,jpdacoeffs)` also returns the correction of measurements, `zcorr`.

You can use this syntax only when `filter` is a `trackingABF` object.

`[xcorr,Pcorr,zcorr] = correctjpda(filter,zmeas,jpdacoeffs,zcov)` returns the correction of measurements, `zcorr`, and also specifies additional measurement covariance, `zcov`, used in the `MeasurementNoise` property of `filter`.

You can use this syntax only when `filter` is a `trackingABF` object.

### Input Arguments

#### **filter** — Filter for object tracking

`trackingKF` object | `trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingABF` — Alpha-beta filter

### **zmeas — Measurements**

*M*-by-*N* matrix

Measurements, specified as an *M*-by-*N* matrix, where *M* is the dimension of a single measurement, and *N* is the number of measurements.

Data Types: `single` | `double`

### **jpdacoeffs — Joint probabilistic data association coefficients**

(*N*+1)-element vector

Joint probabilistic data association coefficients, specified as an (*N*+1)-element vector. The *i*th (*i* = 1, ..., *N*) element of `jpdacoeffs` is the joint probability that the *i*th measurement in `zmeas` is associated with the filter. The last element of `jpdacoeffs` corresponds to the probability that no measurement is associated with the filter. The sum of all elements of `jpdacoeffs` must equal 1.

Data Types: `single` | `double`

### **zcov — Measurement covariance**

*M*-by-*M* matrix

Measurement covariance, specified as an *M*-by-*M* matrix, where *M* is the dimension of the measurement. The same measurement covariance matrix is assumed for all measurements in `zmeas`.

Data Types: `single` | `double`

### **measparams — Measurement parameters**

comma-separated list of arguments

Measurement function arguments, specified as a comma-separated list of arguments. These arguments are the same ones that are passed into the measurement function specified by the `MeasurementFcn` property of the tracking filter. If `filter` is a `trackingKF` or `trackingABF` object, then you cannot specify `measparams`.

Suppose you set `MeasurementFcn` to `@cameas`, and then call `correctjpda`:

```
[xcorr,Pcorr] = correctjpda(filter,frame,sensorpos,sensorvel)
```

The `correctjpda` function internally calls the following:

```
meas = cameas(state,frame,sensorpos,sensorvel)
```

## **Output Arguments**

### **xcorr — Corrected state**

*P*-element vector

Corrected state, returned as a *P*-element vector, where *P* is the dimension of the estimated state. The corrected state represents the *a posteriori* estimate of the state vector, taking into account the current measurements and their associated probabilities.

**Pcorr — Corrected state error covariance**positive-definite  $P$ -by- $P$  matrix

Corrected state error covariance, returned as a positive-definite  $P$ -by- $P$  matrix, where  $P$  is the dimension of the state estimate. The corrected state covariance matrix represents the *a posteriori* estimate of the state covariance matrix, taking into account the current measurements and their associated probabilities.

**zcorr — Corrected measurements** $M$ -by- $N$  matrix

Corrected measurements, returned as an  $M$ -by- $N$  matrix, where  $M$  is the dimension of a single measurement, and  $N$  is the number of measurements. You can return `zcorr` only when `filter` is a `trackingABF` object.

**More About****JPDA Correction Algorithm for Discrete Extended Kalman Filter**

In the measurement update of a regular Kalman filter, the filter usually only needs to update the state and covariance based on one measurement. For instance, the equations for measurement update of a discrete extended Kalman filter can be given as

$$\begin{aligned}x_k^+ &= x_k^- + K_k(y - h(x_k^-)) \\ P_k^+ &= P_k^- - K_k S_k K_k^T\end{aligned}$$

where  $x_k^-$  and  $x_k^+$  are the a priori and a posteriori state estimates, respectively,  $K_k$  is the Kalman gain,  $y$  is the actual measurement, and  $h(x_k^-)$  is the predicted measurement.  $P_k^-$  and  $P_k^+$  are the a priori and a posteriori state error covariance matrices, respectively. The innovation matrix  $S_k$  is defined as

$$S_k = H_k P_k^- H_k^T$$

where  $H_k$  is the Jacobian matrix for the measurement function  $h$ .

In the workflow of a JPDA tracker, the filter needs to process multiple probable measurements  $y_i$  ( $i = 1, \dots, N$ ) with varied probabilities of association  $\beta_i$  ( $i = 0, 1, \dots, N$ ). Note that  $\beta_0$  is the probability that no measurements is associated with the filter. The measurement update equations for a discrete extended Kalman filter used for a JPDA tracker are

$$\begin{aligned}x_k^+ &= x_k^- + K_k \sum_{i=1}^N \beta_i (y_i - h(x_k^-)) \\ P_k^+ &= P_k^- - (1 - \beta_0) K_k S_k K_k^T + P_k\end{aligned}$$

where

$$P_k = K_k \sum_{i=1}^N \left[ \beta_i (y_i - h(x_k^-))(y_i - h(x_k^-))^T - (\delta y)(\delta y)^T \right] K_k^T$$

and

$$\delta y = \sum_{j=1}^N \beta_j (y_j - h(x_k^-))$$

Note that these equations only apply to `trackingEKF` and are not the exact equations used in other tracking filters.

## References

- [1] Fortmann, T., Y. Bar-Shalom, and M. Scheffe. "Sonar Tracking of Multiple Targets Using Joint Probabilistic Data Association." *IEEE Journal of Ocean Engineering*. Vol. 8, Number 3, 1983, pp. 173-184.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

`correctjpda` supports only double-precision code generation, not single-precision.

### See Also

`clone` | `correct` | `distance` | `initialize` | `likelihood` | `predict` | `residual`

**Introduced in R2021a**

## distance

Distances between current and predicted measurements of tracking filter

### Syntax

```
dist = distance(filter,zmeas)
dist = distance(filter,zmeas,measparams)
```

### Description

`dist = distance(filter,zmeas)` computes the normalized distances between one or more current object measurements, `zmeas`, and the corresponding predicted measurements computed by the input `filter`. Use this function to assign measurements to tracks.

This distance computation takes into account the covariance of the predicted state and the measurement noise.

`dist = distance(filter,zmeas,measparams)` specifies additional parameters that are used by the `MeasurementFcn` of the filter.

If `filter` is a `trackingKF` or `trackingABF` object, then you cannot use this syntax.

### Input Arguments

#### **filter** — Filter for object tracking

`trackingKF` object | `trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingABF` — Alpha-beta filter

#### **zmeas** — Measurements of tracked objects

matrix

Measurements of tracked objects, specified as a matrix. Each row of the matrix contains a measurement vector.

#### **measparams** — Parameters for measurement function

cell array

Parameters for measurement function, specified as a cell array. The parameters are passed to the measurement function that is defined in the `MeasurementFcn` property of the `filter`. If `filter` is a `trackingKF` or `trackingABF` object, then you cannot specify `measparams`.

Suppose you set the `MeasurementFcn` property of `filter` to `@cameas`, and then set these values:

```
measurementParams = {frame,sensorpos,sensorpos}
```



The `distance` function internally calls the following:

```
cameas (state, frame, sensorpos, sensorvel)
```

## Output Arguments

### **dist** — Distances between measurements

row vector

Distances between measurements, returned as a row vector. Each element corresponds to a distance between the predicted measurement in the input `filter` and a measurement contained in a row of `zmeas`.

## Algorithms

The `distance` function computes the normalized distance between the filter object and a set of measurements. This distance computation is a variant of the Mahalanobis distance and takes into account the residual (the difference between the object measurement and the value predicted by the filter), the residual covariance, and the measurement noise.

Consider an extended Kalman filter with state  $x$  and measurement  $z$ . The equations used to compute the residual,  $z_{\text{res}}$ , and the residual covariance,  $S$ , are

$$\begin{aligned} z_{\text{res}} &= z - h(x), \\ S &= R + HPH^T, \end{aligned}$$

where:

- $h$  is the measurement function defined in the `MeasurementFcn` property of the filter.
- $R$  is the measurement noise covariance defined in the `MeasurementNoise` property of the filter.
- $H$  is the Jacobian of the measurement function defined in the `MeasurementJacobianFcn` property of the filter.

The residual covariance calculation for other filters can vary slightly from the one shown because tracking filters have different ways of propagating the covariance to the measurement space. For example, instead of using the Jacobian of the measurement function to propagate the covariance, unscented Kalman filters sample the covariance, and then propagate the sampled points.

The equation for the Mahalanobis distance,  $d^2$ , is

$$d^2 = z_{\text{res}}^T S^{-1} z_{\text{res}},$$

The distance function computes the normalized distance,  $d_n$ , as

$$d_n = d^2 + \log(|S|),$$

where  $\log(|S|)$  is the logarithm of the determinant of residual covariance  $S$ .

The  $\log(|S|)$  term accounts for tracks that are coasted, meaning that they are predicted but have not had an update for a long time. Tracks in this state can make  $S$  very large, resulting in a smaller Mahalanobis distance relative to the updated tracks. This difference in distance values can cause the coasted tracks to incorrectly take detections from the updated tracks. The  $\log(|S|)$  term compensates for this effect by penalizing such tracks, whose predictions are highly uncertain.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`clone` | `correct` | `correctjpda` | `initialize` | `likelihood` | `predict` | `residual`

**Introduced in R2021a**

# initialize

Initialize state and covariance of tracking filter

## Syntax

```
initialize(filter, state, statecov)
initialize(filter, state, statecov, Name, Value)
```

## Description

`initialize(filter, state, statecov)` initializes the filter by setting the `State` and `StateCovariance` properties of the filter with the corresponding `state` and `statecov` inputs.

`initialize(filter, state, statecov, Name, Value)` also initializes properties of `filter` by using one or more name-value pairs. Specify the name of the filter property and the value to which you want to initialize it. You cannot change the size or type of the properties that you initialize.

## Input Arguments

### **filter** — Filter for object tracking

trackingKF object | trackingEKF object | trackingUKF object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter

### **state** — Filter state

real-valued  $M$ -element vector

Filter state, specified as a real-valued  $M$ -element vector, where  $M$  is the size of the filter state.

Example: `[200; 0.2]`

Data Types: `double`

### **statecov** — State estimation error covariance

positive-definite real-valued  $M$ -by- $M$  matrix

State estimation error covariance, specified as a positive-definite real-valued  $M$ -by- $M$  matrix.  $M$  is the size of the filter state. The covariance matrix represents the uncertainty in the filter state.

Example: `[20 0.1; 0.1 1]`

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

[clone](#) | [correct](#) | [correctjpda](#) | [distance](#) | [likelihood](#) | [predict](#) | [residual](#)

**Introduced in R2021a**

# likelihood

Likelihood of measurement from tracking filter

## Syntax

```
measlikelihood = likelihood(filter,zmeas)
measlikelihood = likelihood(filter,zmeas,measparams)
```

## Description

`measlikelihood = likelihood(filter,zmeas)` returns the likelihood of a measurement, `zmeas`, that was produced by the specified filter, `filter`.

`measlikelihood = likelihood(filter,zmeas,measparams)` specifies additional parameters that are used by the `MeasurementFcn` of the filter.

If `filter` is a `trackingKF` or `trackingABF` object, then you cannot use this syntax.

## Input Arguments

### **filter** — Filter for object tracking

`trackingKF` object | `trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingABF` — Alpha-beta filter

### **zmeas** — Current measurement of tracked object

vector | matrix

Current measurement of a tracked object, specified a vector or matrix.

### **measparams** — Parameters for measurement function

cell array

Parameters for measurement function, specified as a cell array. The parameters are passed to the measurement function that is defined in the `MeasurementFcn` of the input filter. If `filter` is a `trackingKF` or `trackingABF` object, then you cannot specify `measparams`.

## Output Arguments

### **measlikelihood** — Likelihood of measurement

scalar

Likelihood of measurement, returned as a scalar.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

### **See Also**

`clone` | `correct` | `correctjpda` | `distance` | `initialize` | `predict` | `residual`

**Introduced in R2021a**

## predict

Predict state and state estimation error covariance of tracking filter

### Syntax

```
[xpred,Ppred] = predict(filter)
[xpred,Ppred] = predict(filter,dt)
[xpred,Ppred] = predict(filter,predparams)

[xpred,Ppred,zpred] = predict(filter)
[xpred,Ppred,zpred] = predict(filter,dt)

predict(filter,___)
xpred = predict(filter,___)
```

### Description

`[xpred,Ppred] = predict(filter)` returns the predicted state, `xpred`, and the predicted state estimation error covariance, `Ppred`, for the next time step of the input tracking filter. The predicted values overwrite the internal state and state estimation error covariance of `filter`.

`[xpred,Ppred] = predict(filter,dt)` specifies the time step as a positive scalar in seconds, and returns one or more of the outputs from the preceding syntaxes.

`[xpred,Ppred] = predict(filter,predparams)` specifies additional prediction parameters used by the state transition function. The state transition function is defined in the `StateTransitionFcn` property of `filter`.

`[xpred,Ppred,zpred] = predict(filter)` also returns the predicted measurement at the next time step.

You can use this syntax only when `filter` is a `trackingABF` object.

`[xpred,Ppred,zpred] = predict(filter,dt)` returns the predicted state, state estimation error covariance, and measurement at the specified time step.

You can use this syntax only when `filter` is a `trackingABF` object.

`predict(filter,___)` updates `filter` with the predicted state and state estimation error covariance without returning the predicted values. Specify the tracking filter and any of the input argument combinations from preceding syntaxes.

`xpred = predict(filter,___)` updates `filter` with the predicted state and state estimation error covariance but returns only the predicted state, `xpred`.

### Examples

## Constant-Velocity Extended Kalman Filter

Create a two-dimensional `trackingEKF` object and use name-value pairs to define the `StateTransitionJacobianFcn` and `MeasurementJacobianFcn` properties. Use the predefined constant-velocity motion and measurement models and their Jacobians.

```
EKF = trackingEKF(@constvel,@cvmeas,[0;0;0;0], ...
    'StateTransitionJacobianFcn',@constveljac, ...
    'MeasurementJacobianFcn',@cvmeasjac);
```

Run the filter. Use the `predict` and `correct` functions to propagate the state. You may call `predict` and `correct` in any order and as many times you want. Specify the measurement in Cartesian coordinates.

```
measurement = [1;1;0];
[xpred, Ppred] = predict(EKF);
[xcorr, Pcorr] = correct(EKF,measurement);
[xpred, Ppred] = predict(EKF);
[xpred, Ppred] = predict(EKF)
```

xpred = 4×1

```
1.2500
0.2500
1.2500
0.2500
```

Ppred = 4×4

```
11.7500    4.7500         0         0
 4.7500    3.7500         0         0
         0         0    11.7500    4.7500
         0         0    4.7500    3.7500
```

## Input Arguments

### filter — Filter for object tracking

`trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingABF` — Alpha-beta filter

To use the `predict` function with a `trackingKF` linear Kalman filter, see `predict` (`trackingKF`).

### dt — Time step

positive scalar

Time step for next prediction, specified as a positive scalar in seconds.



**predparams — Prediction parameters**

comma-separated list of arguments

Prediction parameters used by the state transition function, specified as a comma-separated list of arguments. These arguments are the same arguments that are passed into the state transition function specified by the `StateTransitionFcn` property of the input `filter`.

Suppose you set the `StateTransitionFcn` property to `@constacc` and then call the `predict` function:

```
[xpred,Ppred] = predict(filter,dt)
```

The `predict` function internally calls the following:

```
state = constacc(state,dt)
```

**Output Arguments****xpred — Predicted state of filter**

vector | matrix

Predicted state of the filter, specified as a vector or matrix. The `State` property of the input `filter` is overwritten with this value.

**Ppred — Predicted state covariance of filter**

vector | matrix

Predicted state covariance of the filter, specified as a vector or matrix. The `StateCovariance` property of the input `filter` is overwritten with this value.

**zpred — Predicted measurement**

vector | matrix

Predicted measurement, specified as a vector or matrix. You can return `zpred` only when `filter` is a `trackingABF` object.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`clone` | `correct` | `correctjpda` | `distance` | `initialize` | `likelihood` | `residual`

**Introduced in R2021a**

## predict

Predict state and state estimation error covariance of linear Kalman filter

### Syntax

```
[xpred,Ppred] = predict(filter)
[xpred,Ppred] = predict(filter,dt)

[xpred,Ppred] = predict(filter)
[xpred,Ppred] = predict(filter,A)
[xpred,Ppred] = predict(filter,A,Q)

[xpred,Ppred] = predict(filter,u)
[xpred,Ppred] = predict(filter,u,A,B)
[xpred,Ppred] = predict(filter,u,A,B,Q)
```

### Description

#### Syntaxes for Predefined Motion Model

Use these syntaxes if you specify a predefined motion model in the `MotionModel` property of the `trackingKF` object.

`[xpred,Ppred] = predict(filter)` returns the predicted state `xpred` and the predicted state estimation error covariance `Ppred` after one second using the motion model specified in the `filter`. The predicted values overwrite the internal state and state estimation error covariance of the `filter`.

`[xpred,Ppred] = predict(filter,dt)` predicts the state and state estimation error covariance at the specified time step `dt`.

#### Syntaxes for Custom Motion Model Without Control Input

Use these syntaxes if you specify the `MotionModel` property as "Custom" and do not use control inputs.

`[xpred,Ppred] = predict(filter)` returns the predicted state `xpred` and the predicted state estimation error covariance `Ppred` using the state transition matrix specified in the `StateTransitionModel` property of the `filter`. The predicted values overwrite the internal state and state estimation error covariance of the `filter`.

`[xpred,Ppred] = predict(filter,A)` specifies the state transition model `A`. Use this syntax when the state transition model is time-varying.

`[xpred,Ppred] = predict(filter,A,Q)` specifies the state transition model `A` and the process noise covariance `Q`. Use this syntax when the state transition model and the process noise are time-varying.

#### Syntaxes for Custom Motion Model with Control Input

Use these syntaxes if you specify the `MotionModel` property as "Custom" and use control inputs.

`[xpred,Ppred] = predict(filter,u)` returns the predicted state `xpred` and the predicted state estimation error covariance `Ppred` using the state transition model specified in the `StateTransitionModel` property of the `filter` and a control input `u`.

`[xpred,Ppred] = predict(filter,u,A,B)` specifies the force or control input `u`, the state transition model `A`, and the control model `B`. Use this syntax when the state transition model and control model are time-varying.

`[xpred,Ppred] = predict(filter,u,A,B,Q)` specifies the force or control input `u`, the state transition model `A`, the control model `B`, and the process noise covariance `Q`. Use this syntax when the state transition model, control model, and process noise are time-varying.

## Examples

### Create Constant-Velocity Linear Kalman Filter

Create a linear Kalman filter that uses a 2D constant velocity motion model. Assume that the measurement consists of the `xy`-location of the object.

Specify the initial state estimate to have zero velocity.

```
x = 5.3;
y = 3.6;
initialState = [x;0;y;0];
KF = trackingKF('MotionModel','2D Constant Velocity','State',initialState);
```

Create measured positions for the object on a constant-velocity trajectory.

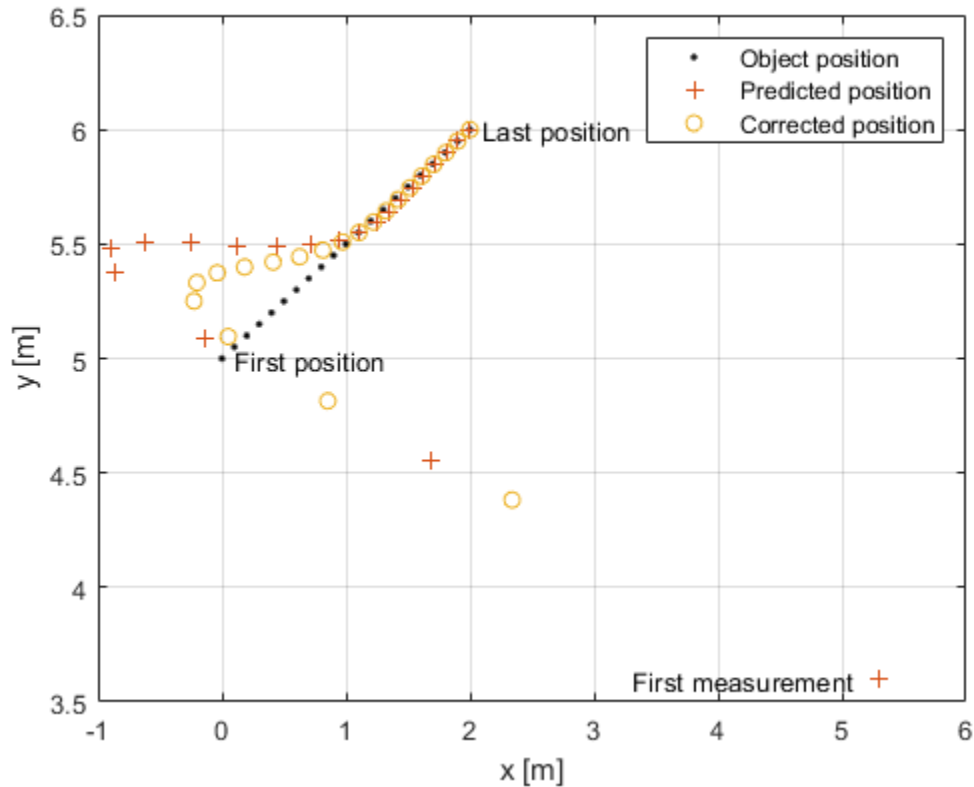
```
vx = 0.2;
vy = 0.1;
T = 0.5;
pos = [0:vx*T:2;
       5:vy*T:6]';
```

Predict and correct the state of the object.

```
for k = 1:size(pos,1)
    pstates(k,:) = predict(KF,T);
    cstates(k,:) = correct(KF,pos(k,:));
end
```

Plot the tracks.

```
plot(pos(:,1),pos(:,2),"k.",pstates(:,1),pstates(:,3),"+", ...
      cstates(:,1),cstates(:,3),"o")
xlabel("x [m]")
ylabel("y [m]")
grid
xt = [x-2, pos(1,1)+0.1, pos(end,1)+0.1];
yt = [y, pos(1,2), pos(end,2)];
text(xt,yt,["First measurement","First position","Last position"])
legend("Object position","Predicted position","Corrected position")
```



### Use Custom trackingKF with Control Inputs

Specify a simulation time of 10 seconds with a time step of 1 second.

```
rng(2021) % For repeatable results
simulationTime = 20;
dt = 1;
tspan = 0:dt:simulationTime;
steps = length(tspan);
```

Specify the motion model as a 2-D constant velocity model with a state of  $[x; vx; y; vy]$ . The measurement is  $[x; y]$ .

```
A1D = [1 dt; 0 1];
A = kron(eye(2),A1D) % State transition model
```

```
A = 4x4
```

```
1 1 0 0
0 1 0 0
0 0 1 1
0 0 0 1
```

```
H1D = [1 0];
H = kron(eye(2),H1D) % Measurement model
```

```
H = 2x4
```

```
    1    0    0    0
    0    0    1    0
```

```
sigma = 0.2;
R = sigma^2*eye(2); % Measurement noise covariance
```

Specify a control model matrix.

```
B1D = [0; 1];
B = kron(eye(2),B1D) % Control model matrix
```

```
B = 4x2
```

```
    0    0
    1    0
    0    0
    0    1
```

Assume the control inputs are sinusoidal on the velocity components, vx and vy.

```
gain = 5;
Ux = gain*sin(tspan(2:end));
Uy = gain*cos(tspan(2:end));
U =[Ux; Uy]; % Control inputs
```

Assuming the true initial state is [1 1 1 -1], simulate the system to obtain true states and measurements.

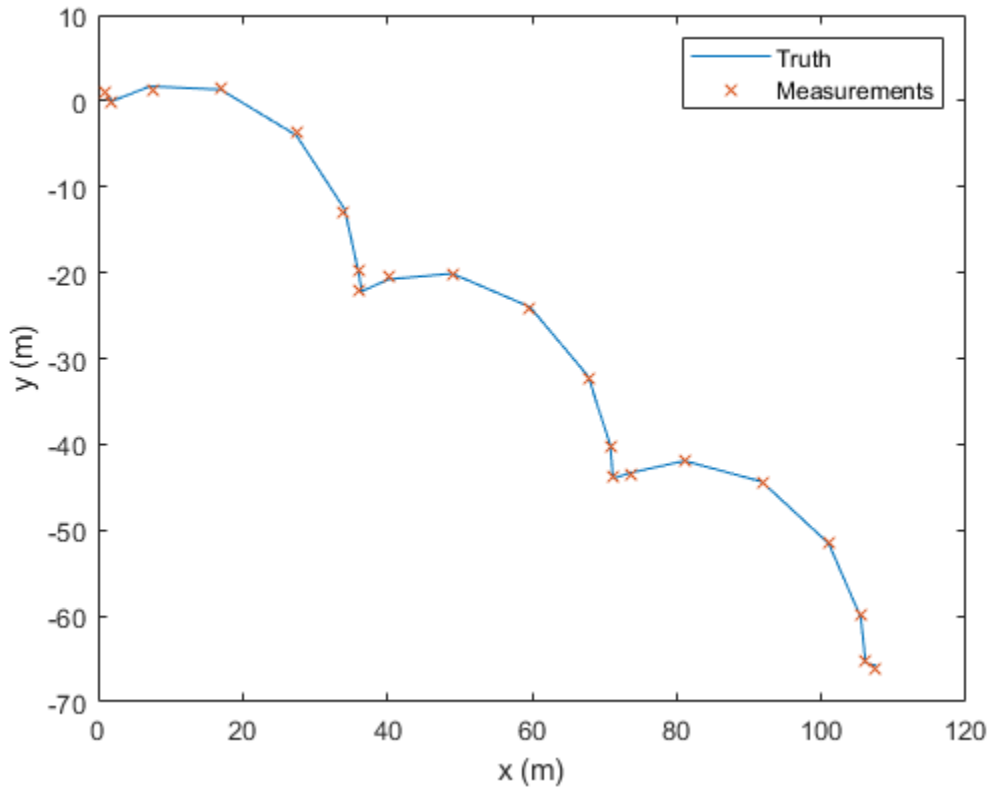
```
initialState = [1 1 1 -1]'; % [m m/s m m/s]
trueStates = NaN(4,steps);
trueStates(:,1) = initialState;

for i=2:steps
    trueStates(:,i) = A*trueStates(:,i-1) + B*U(:,i-1);
end

measurements = H*trueStates + chol(R)*randn(2,steps);
```

Visualize the true trajectory and the measurements.

```
figure
plot(trueStates(1,:),trueStates(3,:), "DisplayName", "Truth")
hold on
plot(measurements(1,:),measurements(2,:), "x", "DisplayName", "Measurements")
xlabel("x (m)")
ylabel("y (m)")
legend
```



Create a `trackingKF` filter with a custom motion model. Enable the control input by specifying the control model. Specify the initial state in the filter based on the first measurement.

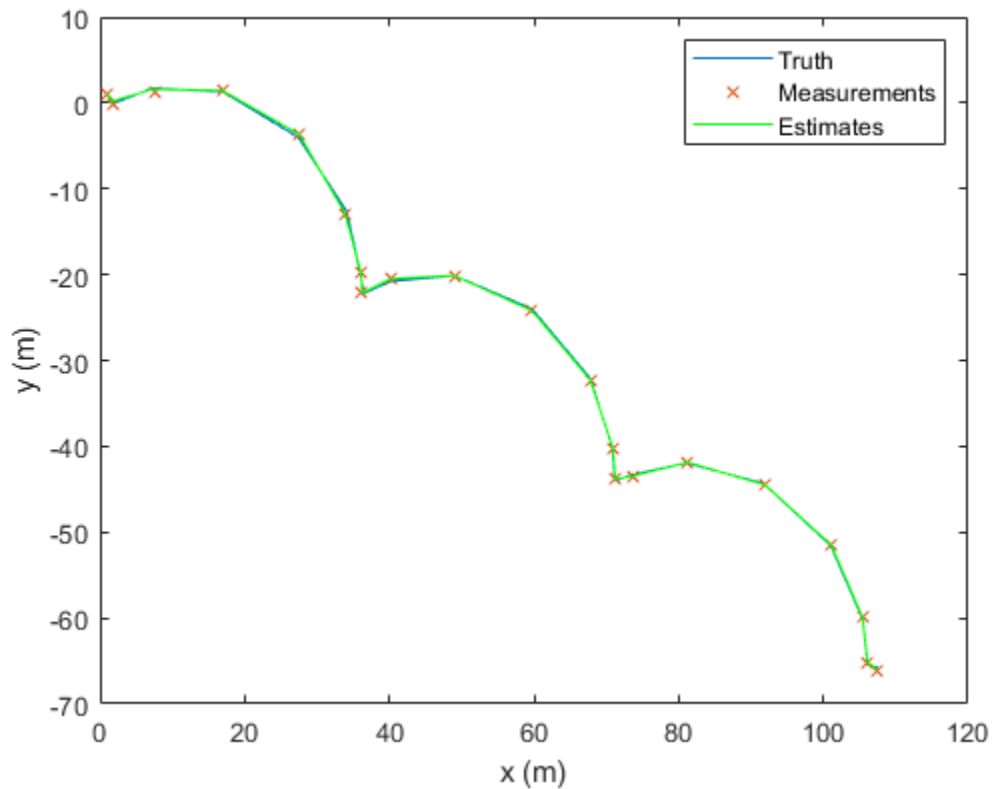
```
initialFilterState = [measurements(1,1); 0; measurements(2,1); 0];
filter = trackingKF("MotionModel", "Custom", ...
    "StateTransitionModel", A, ...
    "MeasurementModel", H, ...
    "ControlModel", B, ...
    "State", initialFilterState);
```

Estimate states by using the `predict` and `correct` object functions.

```
estimateStates = NaN(4, steps);
estimateStates(:, 1) = initialFilterState;
for i = 2:steps
    predict(filter, U(:, i-1));
    estimateStates(:, i) = correct(filter, measurements(:, i));
end
```

Visualize the state estimates.

```
plot(estimateStates(1,:), estimateStates(3,:), "g", "DisplayName", "Estimates");
```



## Input Arguments

### **filter** – Linear Kalman filter for object tracking

trackingKF object

Linear Kalman filter for object tracking, specified as a trackingKF object.

### **dt** – Time step

positive scalar

Time step, specified as a positive scalar. Units are in seconds.

### **A** – State transition model

real-valued  $M$ -by- $M$  matrix

State transition model, specified as a real-valued  $M$ -by- $M$  matrix, where  $M$  is the size of the state vector.

### **Q** – Covariance of process noise

nonnegative scalar | positive-semidefinite  $D$ -by- $D$  matrix | positive-semidefinite  $M$ -by- $M$  matrix

Covariance of process noise, specified as a nonnegative scalar, a positive-semidefinite  $D$ -by- $D$  matrix, or a positive-semidefinite  $M$ -by- $M$  matrix.

- When the `MotionModel` property of the filter is specified as one of the predefined motion models, specify `Q` as a positive-semidefinite  $D$ -by- $D$  matrix, where  $D$  is the number of dimensions of the target motion. For example,  $D = 2$  for the "2D Constant Velocity" or the "2D Constant Acceleration" motion model.

In this case, if you specify `Q` as a nonnegative scalar, then the scalar extends to the diagonal elements of a diagonal covariance matrix, whose size is  $D$ -by- $D$ .

- When the `MotionModel` property of the filter is specified as "Custom", specify `Q` as a positive-semidefinite  $M$ -by- $M$  matrix, where  $M$  is the size of the filter state. For example,  $M = 6$  if you customize a 3-D motion model in which the state is  $(x, v_x, y, v_y, z, v_z)$ .

In this case, if you specify `Q` as a nonnegative scalar, then the scalar extends to the diagonal elements of a diagonal covariance matrix, whose size is  $M$ -by- $M$ .

#### **u — Control vector**

real-valued  $L$ -element vector

Control vector, specified as a real-valued  $L$ -element vector.

#### **B — Control model**

real-valued  $M$ -by- $L$  matrix

Control model, specified as a real-valued  $M$ -by- $L$  matrix.  $M$  is the size of the state vector.  $L$  is the number of independent controls.

## **Output Arguments**

#### **xpred — Predicted state**

real-valued  $M$ -element vector

Predicted state, returned as a real-valued  $M$ -element vector. The predicted state represents the deducible estimate of the state vector, propagated from the previous state using the state transition and control models.

#### **Ppred — Predicted state error covariance matrix**

real-valued  $M$ -by- $M$  matrix

Predicted state covariance matrix, returned as a real-valued  $M$ -by- $M$  matrix.  $M$  is the size of the state vector. The predicted state covariance matrix represents the deducible estimate of the covariance matrix vector. The filter propagates the covariance matrix from the previous estimate.

## **Extended Capabilities**

#### **C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

## **See Also**

`clone` | `correct` | `correctjpd` | `distance` | `initialize` | `likelihood` | `residual`

**Introduced in R2021a**



# residual

Measurement residual and residual noise from tracking filter

## Syntax

```
[zres, rescov] = residual(filter, zmeas)
[zres, rescov] = residual(filter, zmeas, measparams)
```

## Description

`[zres, rescov] = residual(filter, zmeas)` computes the residual and residual covariance of the current given measurement, `zmeas`, with the predicted measurement in the tracking filter, `filter`. This function applies to filters that assume a Gaussian distribution for noise.

`[zres, rescov] = residual(filter, zmeas, measparams)` specifies additional parameters that are used by the `MeasurementFcn` of the filter.

If `filter` is a `trackingKF` object, then you cannot use this syntax.

## Input Arguments

### **filter** — Filter for object tracking

`trackingKF` object | `trackingEKF` object | `trackingUKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter

### **zmeas** — Current measurement of tracked object

vector | matrix

Current measurement of a tracked object, specified as a vector or matrix.

### **measparams** — Parameters for measurement function

cell array

Parameters for measurement function, specified as a cell array. The parameters are passed to the measurement function that is defined in the `MeasurementFcn` property of the input `filter`. If `filter` is a `trackingKF` object, then you cannot specify `measparams`.

## Output Arguments

### **zres** — Residual between current and predicted measurement

matrix

Residual between current and predicted measurement, returned as a matrix.

**rescov — Residual covariance**

matrix

Residual covariance, returned as a matrix.

**Algorithms**

The residual is the difference between a measurement and the value predicted by the filter. For Kalman filters, the residual calculation depends on whether the filter is linear or nonlinear.

**Linear Kalman Filters**

Given a linear Kalman filter with a current measurement of  $z$ , the residual  $z_{\text{res}}$  is defined as

$$z_{\text{res}} = z - Hx,$$

where:

- $H$  is the measurement model set by the `MeasurementModel` property of the filter.
- $x$  is the current filter state.

The covariance of the residual,  $S$ , is defined as

$$S = R + HPH^T,$$

where:

- $P$  is the state covariance matrix.
- $R$  is the measurement noise matrix set by the `MeasurementNoise` property of the filter.

**Nonlinear Kalman Filters**

Given a nonlinear Kalman filter with a current measurement of  $z$ , the residual  $z_{\text{res}}$  is defined as:

$$z_{\text{res}} = z - h(x),$$

where:

- $h$  is the measurement function set by the `MeasurementFcn` property.
- $x$  is the current filter state.

The covariance of the residual,  $S$ , is defined as:

$$S = R + R_p,$$

where:

- $R$  is the measurement noise matrix set by the `MeasurementNoise` property of the filter.
- $R_p$  is the state covariance matrix projected onto the measurement space.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using MATLAB® Coder™.

**See Also**

`clone` | `correct` | `correctjpd` | `distance` | `initialize` | `likelihood` | `predict`

**Introduced in R2021a**

# insSensor

Inertial navigation system and GNSS/GPS simulation model

## Description

The `insSensor` System object models a device that fuses measurements from an inertial navigation system (INS) and global navigation satellite system (GNSS) such as a GPS, and outputs the fused measurements.

To output fused INS and GNSS measurements:

- 1 Create the `insSensor` object and set its properties.
- 2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#)

## Creation

### Syntax

```
INS = insSensor
INS = insSensor(Name,Value)
```

### Description

`INS = insSensor` returns a System object, `INS`, that models a device that outputs measurements from an INS and GNSS.

`INS = insSensor(Name,Value)` sets properties on page 4-803 using one or more name-value pairs. Unspecified properties have default values. Enclose each property name in quotes.

## Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see [System Design in MATLAB Using System Objects](#).

### MountingLocation — Location of sensor on platform (m)

`[0 0 0]` (default) | three-element real-valued vector of form `[x y z]`

Location of the sensor on the platform, in meters, specified as a three-element real-valued vector of the form `[x y z]`. The vector defines the offset of the sensor origin from the origin of the platform.

**Tunable:** Yes

Data Types: `single` | `double`

**RollAccuracy — Accuracy of roll measurement (deg)**

`0.2` (default) | nonnegative real scalar

Accuracy of the roll measurement of the sensor body, in degrees, specified as a nonnegative real scalar.

Roll is the rotation around the x-axis of the sensor body. Roll noise is modeled as a white noise process. `RollAccuracy` sets the standard deviation of the roll measurement noise.

**Tunable:** Yes

Data Types: `single` | `double`

**PitchAccuracy — Accuracy of pitch measurement (deg)**

`0.2` (default) | nonnegative real scalar

Accuracy of the pitch measurement of the sensor body, in degrees, specified as a nonnegative real scalar.

Pitch is the rotation around the y-axis of the sensor body. Pitch noise is modeled as a white noise process. `PitchAccuracy` defines the standard deviation of the pitch measurement noise.

**Tunable:** Yes

Data Types: `single` | `double`

**YawAccuracy — Accuracy of yaw measurement (deg)**

`1` (default) | nonnegative real scalar

Accuracy of the yaw measurement of the sensor body, in degrees, specified as a nonnegative real scalar.

Yaw is the rotation around the z-axis of the sensor body. Yaw noise is modeled as a white noise process. `YawAccuracy` defines the standard deviation of the yaw measurement noise.

**Tunable:** Yes

Data Types: `single` | `double`

**PositionAccuracy — Accuracy of position measurement (m)**

`[1 1 1]` (default) | nonnegative real scalar | three-element real-valued vector

Accuracy of the position measurement of the sensor body, in meters, specified as a nonnegative real scalar or a three-element real-valued vector. The elements of the vector set the accuracy of the x-, y-, and z-position measurements, respectively. If you specify `PositionAccuracy` as a scalar value, then the object sets the accuracy of all three positions to this value.

Position noise is modeled as a white noise process. `PositionAccuracy` defines the standard deviation of the position measurement noise.

**Tunable:** Yes

Data Types: `single` | `double`

**VelocityAccuracy — Accuracy of velocity measurement (m/s)**

`0.05` (default) | nonnegative real scalar

Accuracy of the velocity measurement of the sensor body, in meters per second, specified as a nonnegative real scalar.

Velocity noise is modeled as a white noise process. `VelocityAccuracy` defines the standard deviation of the velocity measurement noise.

**Tunable:** Yes

Data Types: `single` | `double`

### **AccelerationAccuracy — Accuracy of acceleration measurement (m/s<sup>2</sup>)**

0 (default) | nonnegative real scalar

Accuracy of the acceleration measurement of the sensor body, in meters per second, specified as a nonnegative real scalar.

Acceleration noise is modeled as a white noise process. `AccelerationAccuracy` defines the standard deviation of the acceleration measurement noise.

**Tunable:** Yes

Data Types: `single` | `double`

### **AngularVelocityAccuracy — Accuracy of angular velocity measurement (deg/s)**

0 (default) | nonnegative real scalar

Accuracy of the angular velocity measurement of the sensor body, in meters per second, specified as a nonnegative real scalar.

Angular velocity is modeled as a white noise process. `AngularVelocityAccuracy` defines the standard deviation of the acceleration measurement noise.

**Tunable:** Yes

Data Types: `single` | `double`

### **TimeInput — Enable input of simulation time**

`false` or 0 (default) | `true` or 1

Enable input of simulation time, specified as a logical 0 (`false`) or 1 (`true`). Set this property to `true` to input the simulation time by using the `simTime` argument.

**Tunable:** No

Data Types: `logical`

### **HasGNSSFix — Enable GNSS fix**

`true` or 1 (default) | `false` or 0

Enable GNSS fix, specified as a logical 1 (`true`) or 0 (`false`). Set this property to `false` to simulate the loss of a GNSS receiver fix. When a GNSS receiver fix is lost, position measurements drift at a rate specified by the `PositionErrorFactor` property.

**Tunable:** Yes

### **Dependencies**

To enable this property, set `TimeInput` to `true`.

Data Types: `logical`

**PositionErrorFactor — Position error factor without GNSS fix**

`[0 0 0]` (default) | nonnegative scalar | 1-by-3 vector of scalars

Position error factor without GNSS fix, specified as a scalar or a 1-by-3 vector of scalars.

When the `HasGNSSFix` property is set to `false`, the position error grows at a quadratic rate due to constant bias in the accelerometer. The position error for a position component  $E(t)$  can be expressed as  $E(t) = 1/2\alpha t^2$ , where  $\alpha$  is the position error factor for the corresponding component and  $t$  is the time since the GNSS fix is lost. While running, the object computes  $t$  based on the `simTime` input. The computed  $E(t)$  values for the  $x$ ,  $y$ , and  $z$  components are added to the corresponding position components of the `gTruth` input.

**Tunable:** Yes

**Dependencies**

To enable this property, set `TimeInput` to `true` and `HasGNSSFix` to `false`.

Data Types: `single` | `double`

**RandomStream — Random number source**

`'Global stream'` (default) | `'mt19937ar with seed'`

Random number source, specified as one of these options:

- `'Global stream'` -- Generate random numbers using the current global random number stream.
- `'mt19937ar with seed'` -- Generate random numbers using the `mt19937ar` algorithm, with the seed specified by the `Seed` property.

Data Types: `char` | `string`

**Seed — Initial seed**

`67` (default) | nonnegative integer

Initial seed of the `mt19937ar` random number generator algorithm, specified as a nonnegative integer.

**Dependencies**

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**Usage****Syntax**

```
measurement = INS(gTruth)  
measurement = INS(gTruth, simTime)
```

## Description

`measurement = INS(gTruth)` models the data received from an INS sensor reading and GNSS sensor reading. The output measurement is based on the inertial ground-truth state of the sensor body, `gTruth`.

`measurement = INS(gTruth, simTime)` additionally specifies the time of simulation, `simTime`. To enable this syntax, set the `TimeInput` property to `true`.

## Input Arguments

### **gTruth — Inertial ground-truth state of sensor body**

structure

Inertial ground-truth state of sensor body, in local Cartesian coordinates, specified as a structure containing these fields:

Field	Description
'Position'	Position, in meters, specified as a real, finite $N$ -by-3 matrix of $[x \ y \ z]$ vectors. $N$ is the number of samples in the current frame.
'Velocity'	Velocity ( $v$ ), in meters per second, specified as a real, finite $N$ -by-3 matrix of $[v_x \ v_y \ v_z]$ vector. $N$ is the number of samples in the current frame.
'Orientation'	Orientation with respect to the local Cartesian coordinate system, specified as one of these options: <ul style="list-style-type: none"> <li><math>N</math>-element column vector of quaternion objects</li> <li>3-by-3-by-<math>N</math> array of rotation matrices</li> <li><math>N</math>-by-3 matrix of <math>[x_{roll} \ y_{pitch} \ z_{yaw}]</math> angles in degrees</li> </ul> Each quaternion or rotation matrix is a frame rotation from the local Cartesian coordinate system to the current sensor body coordinate system. $N$ is the number of samples in the current frame.
'Acceleration'	Acceleration ( $a$ ), in meters per second squared, specified as a real, finite $N$ -by-3 matrix of $[a_x \ a_y \ a_z]$ vectors. $N$ is the number of samples in the current frame.
'AngularVelocity'	Angular velocity ( $\omega$ ), in degrees per second squared, specified as a real, finite $N$ -by-3 matrix of $[\omega_x \ \omega_y \ \omega_z]$ vectors. $N$ is the number of samples in the current frame.

The field values must be of type `double` or `single`.

The `Position`, `Velocity`, and `Orientation` fields are required. The other fields are optional.

```
Example: struct('Position',[0 0 0],'Velocity',[0 0
0],'Orientation',quaternion([1 0 0 0]))
```

### **simTime — Simulation time**

nonnegative real scalar

Simulation time, in seconds, specified as a nonnegative real scalar.

Data Types: `single` | `double`

### **Output Arguments**

#### **measurement — Measurement of sensor body motion**

structure

Measurement of the sensor body motion, in local Cartesian coordinates, returned as a structure containing these fields:

<b>Field</b>	<b>Description</b>
'Position'	Position, in meters, specified as a real, finite $N$ -by-3 matrix of $[x\ y\ z]$ vectors. $N$ is the number of samples in the current frame.
'Velocity'	Velocity ( $v$ ), in meters per second, specified as a real, finite $N$ -by-3 matrix of $[v_x\ v_y\ v_z]$ vector. $N$ is the number of samples in the current frame.
'Orientation'	Orientation with respect to the local Cartesian coordinate system, specified as one of these options: <ul style="list-style-type: none"> <li><math>N</math>-element column vector of quaternion objects</li> <li>3-by-3-by-<math>N</math> array of rotation matrices</li> <li><math>N</math>-by-3 matrix of <math>[x_{\text{roll}}\ y_{\text{pitch}}\ z_{\text{yaw}}]</math> angles in degrees</li> </ul> Each quaternion or rotation matrix is a frame rotation from the local Cartesian coordinate system to the current sensor body coordinate system. $N$ is the number of samples in the current frame.
'Acceleration'	Acceleration ( $a$ ), in meters per second squared, specified as a real, finite $N$ -by-3 matrix of $[a_x\ a_y\ a_z]$ vectors. $N$ is the number of samples in the current frame.
'AngularVelocity'	Angular velocity ( $\omega$ ), in degrees per second squared, specified as a real, finite $N$ -by-3 matrix of $[\omega_x\ \omega_y\ \omega_z]$ vectors. $N$ is the number of samples in the current frame.

The returned field values are of type `double` or `single` and are of the same type as the corresponding field values in the `gTruth` input.



## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `insSensor`

`perturbations` Perturbation defined on object  
`perturb` Apply perturbations to object

### Common to All System Objects

`step` Run System object algorithm  
`clone` Create duplicate System object  
`isLocked` Determine if System object is in use  
`reset` Reset internal states of System object  
`release` Release resources and allow changes to System object property values and input characteristics

## Examples

### Generate INS Measurements from Stationary Input

Create a motion structure that defines a stationary position at the local north-east-down (NED) origin. Because the platform is stationary, you need to define only a single sample. Assume the ground-truth motion is sampled for 10 seconds with a 100 Hz sample rate. Create a default `insSensor` System object™. Preallocate variables to hold output from the `insSensor` object.

```
Fs = 100;
duration = 10;
numSamples = Fs*duration;

motion = struct( ...
    'Position', zeros(1,3), ...
    'Velocity', zeros(1,3), ...
    'Orientation', ones(1,1, 'quaternion'));

INS = insSensor;

positionMeasurements = zeros(numSamples,3);
velocityMeasurements = zeros(numSamples,3);
orientationMeasurements = zeros(numSamples,1, 'quaternion');
```

In a loop, call `INS` with the stationary motion structure to return the position, velocity, and orientation measurements in the local NED coordinate system. Log the position, velocity, and orientation measurements.

```
for i = 1:numSamples
    measurements = INS(motion);

    positionMeasurements(i,:) = measurements.Position;
    velocityMeasurements(i,:) = measurements.Velocity;
```

```
orientationMeasurements(i) = measurements.Orientation;
```

```
end
```

Convert the orientation from quaternions to Euler angles for visualization purposes. Plot the position, velocity, and orientation measurements over time.

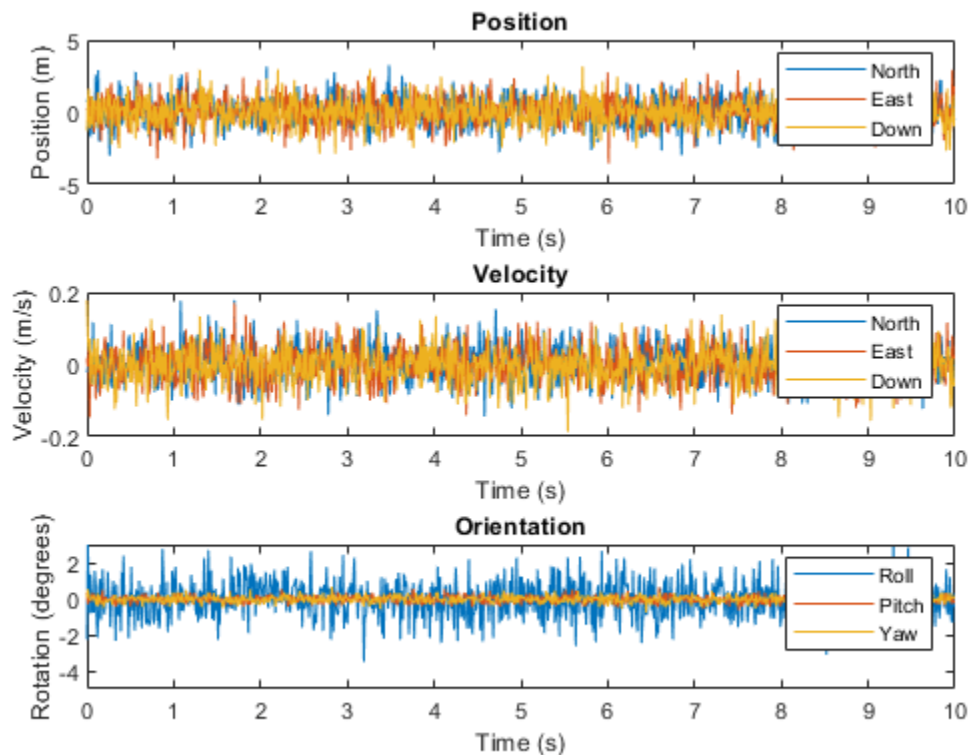
```
orientationMeasurements = eulerd(orientationMeasurements, 'ZYX', 'frame');
```

```
t = (0:(numSamples-1))/Fs;
```

```
subplot(3,1,1)
plot(t,positionMeasurements)
title('Position')
xlabel('Time (s)')
ylabel('Position (m)')
legend('North', 'East', 'Down')
```

```
subplot(3,1,2)
plot(t,velocityMeasurements)
title('Velocity')
xlabel('Time (s)')
ylabel('Velocity (m/s)')
legend('North', 'East', 'Down')
```

```
subplot(3,1,3)
plot(t,orientationMeasurements)
title('Orientation')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
legend('Roll', 'Pitch', 'Yaw')
```



### Generate INS Measurements for Radar Scenario

Generate INS measurements using the `insSensor System` object™. Use `waypointTrajectory` to generate the ground-truth path. Use `radarScenario` to organize the simulation and visualize the motion.

Specify the ground-truth trajectory as a figure-eight path in the North-East plane. Use a 50 Hz sample rate and 5 second duration.

```

Fs = 50;
duration = 5;
numSamples = Fs*duration;
t = (0:(numSamples-1)).'/Fs;

a = 2;

x = a.*sqrt(2).*cos(t) ./ (sin(t).^2 + 1);
y = sin(t) .* x;
z = zeros(numSamples,1);

waypoints = [x,y,z];

path = waypointTrajectory('Waypoints',waypoints,'TimeOfArrival',t);

```

Create an `insSensor` System object to model receiving INS data. Set the `PositionAccuracy` to 0.1.

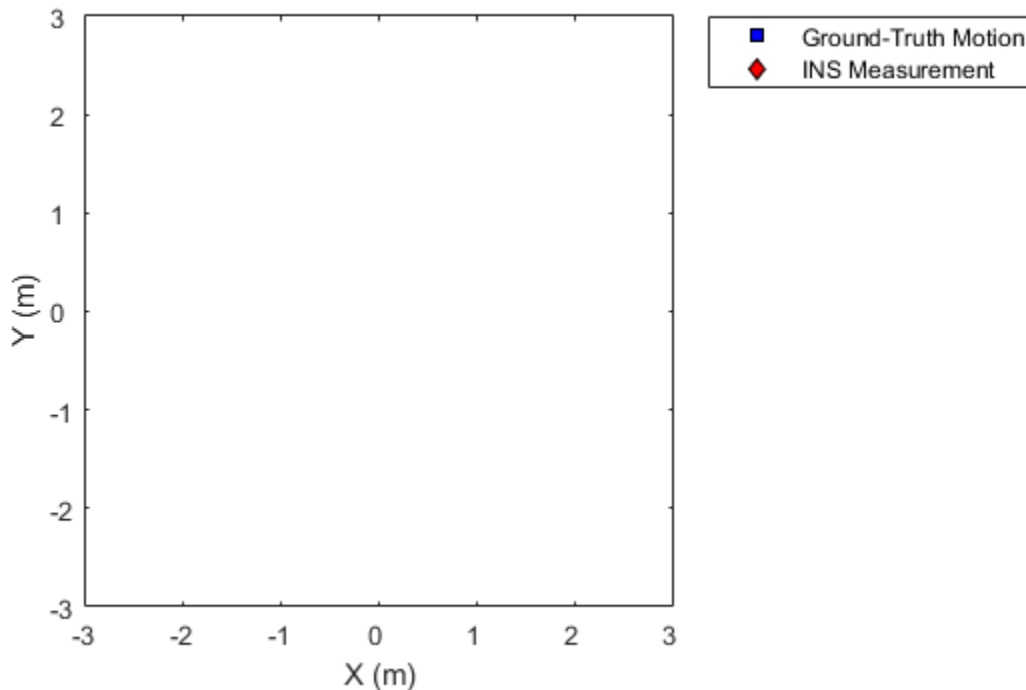
```
ins = insSensor('PositionAccuracy',0.1);
```

Create a radar scenario with a single platform whose motion is defined by `path`.

```
scenario = radarScenario('UpdateRate',Fs);
plat = platform(scenario);
plat.Trajectory = path;
```

Create a theater plot to visualize the ground-truth platform motion and the platform motion measurements modeled by `insSensor`.

```
tp = theaterPlot('XLimits',[-3, 3],'YLimits', [-3, 3]);
platPlotter = platformPlotter(tp, ...
    'DisplayName', 'Ground-Truth Motion', ...
    'Marker', 's', ...
    'MarkerFaceColor','blue');
insPlotter = detectionPlotter(tp, ...
    'DisplayName','INS Measurement', ...
    'Marker','d', ...
    'MarkerFaceColor','red');
```



In a loop, advance the scenario until it is complete. For each time step, get the current motion sample, model INS measurements for the motion, and then plot the result.

```
while advance(scenario)
    motion = platformPoses(scenario,'quaternion');
```

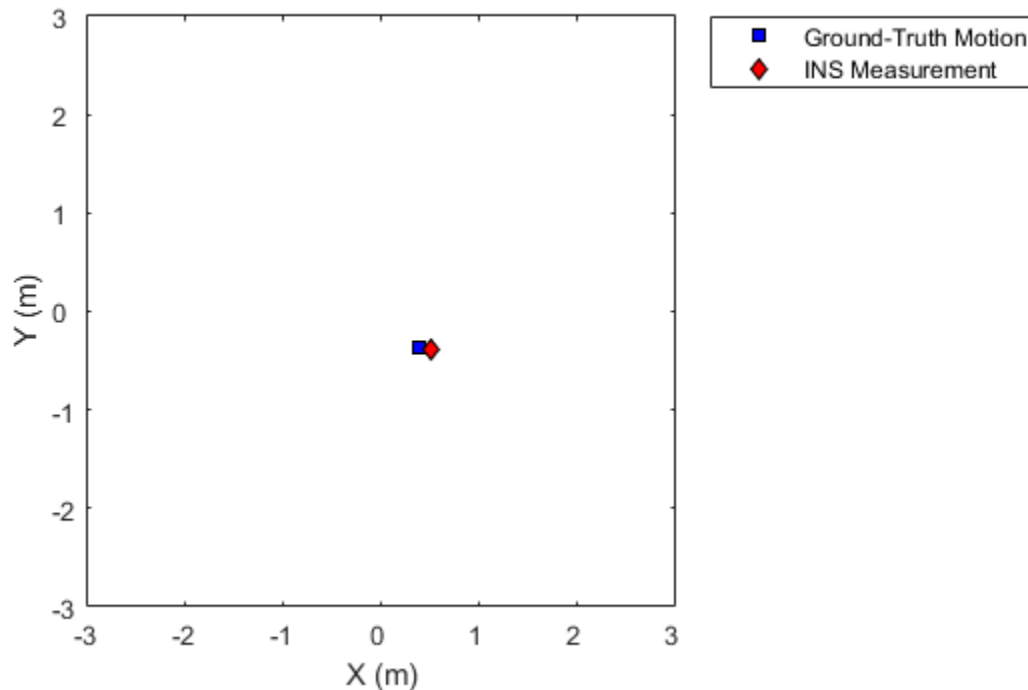
```

insMeas = ins(motion);

plotPlatform(platPlotter,motion.Position);
plotDetection(insPlotter,insMeas.Position);

pause(1/scenario.UpdateRate)
end

```



### Generate INS Measurements for a Turning Platform

Generate INS measurements using the `insSensor System` object™. Use `waypointTrajectory` to generate the ground-truth path.

Specify a ground-truth orientation that begins with the sensor body  $x$ -axis aligned with North and ends with the sensor body  $x$ -axis aligned with East. Specify waypoints for an arc trajectory and a time-of-arrival vector for the corresponding waypoints. Use a 100 Hz sample rate. Create a `waypointTrajectory System` object with the waypoint constraints, and set `SamplesPerFrame` so that the entire trajectory is output with one call.

```

eulerAngles = [0,0,0; ...
               0,0,0; ...
               90,0,0; ...
               90,0,0];

```

```

orientation = quaternion(eulerAngles, 'eulerd', 'ZYX', 'frame');

r = 20;
waypoints = [0,0,0; ...
             100,0,0; ...
             100+r,r,0; ...
             100+r,100+r,0];

toa = [0,10,10+(2*pi*r/4),20+(2*pi*r/4)];

Fs = 100;
numSamples = floor(Fs*toa(end));

path = waypointTrajectory('Waypoints', waypoints, ...
                          'TimeOfArrival', toa, ...
                          'Orientation', orientation, ...
                          'SampleRate', Fs, ...
                          'SamplesPerFrame', numSamples);

```

Create an `insSensor` System object to model receiving INS data. Set the `PositionAccuracy` to 0.1.

```
ins = insSensor('PositionAccuracy', 0.1);
```

Call the waypoint trajectory object, `path`, to generate the ground-truth motion. Call the INS simulator, `ins`, with the ground-truth motion to generate INS measurements.

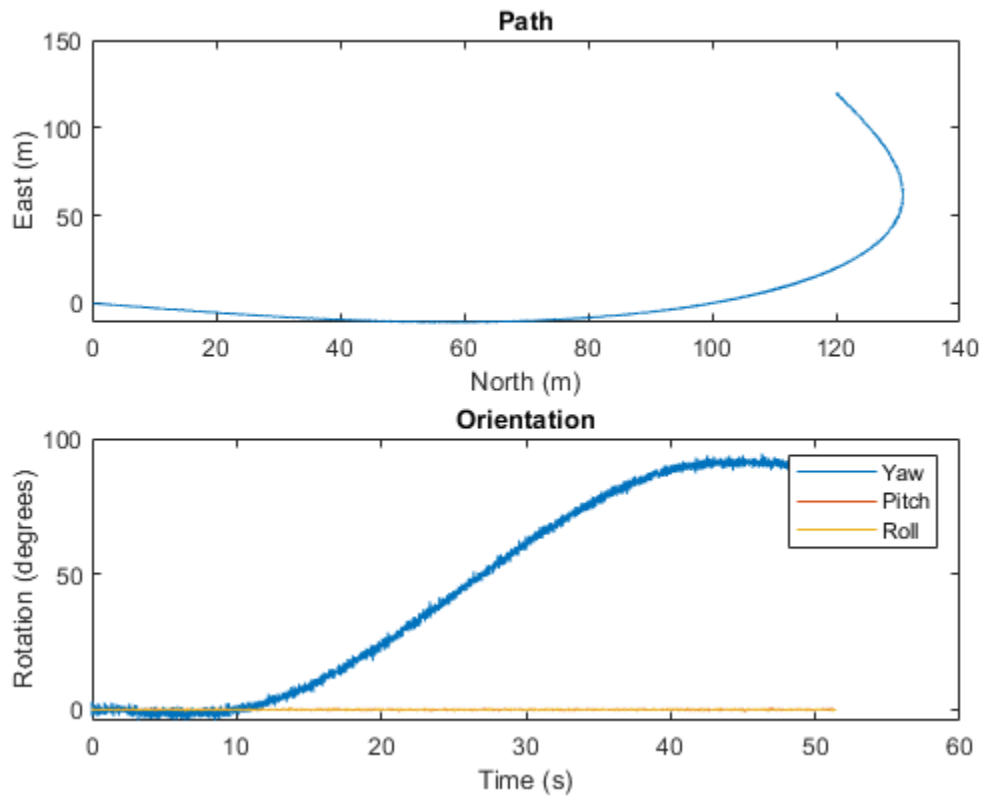
```
[motion.Position, motion.Orientation, motion.Velocity] = path();
insMeas = ins(motion);
```

Convert the orientation returned by `ins` to Euler angles in degrees for visualization purposes. Plot the full path and orientation over time.

```
orientationMeasurementEuler = eulerd(insMeas.Orientation, 'ZYX', 'frame');

subplot(2,1,1)
plot(insMeas.Position(:,1), insMeas.Position(:,2));
title('Path')
xlabel('North (m)')
ylabel('East (m)')

subplot(2,1,2)
t = (0:(numSamples-1)).'/Fs;
plot(t, orientationMeasurementEuler(:,1), ...
      t, orientationMeasurementEuler(:,2), ...
      t, orientationMeasurementEuler(:,3));
title('Orientation')
legend('Yaw', 'Pitch', 'Roll')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
```



## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The object functions, perturbations and perturb, do not support code generation.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Objects

radarScenario

### Objects

Introduced in R2021a

