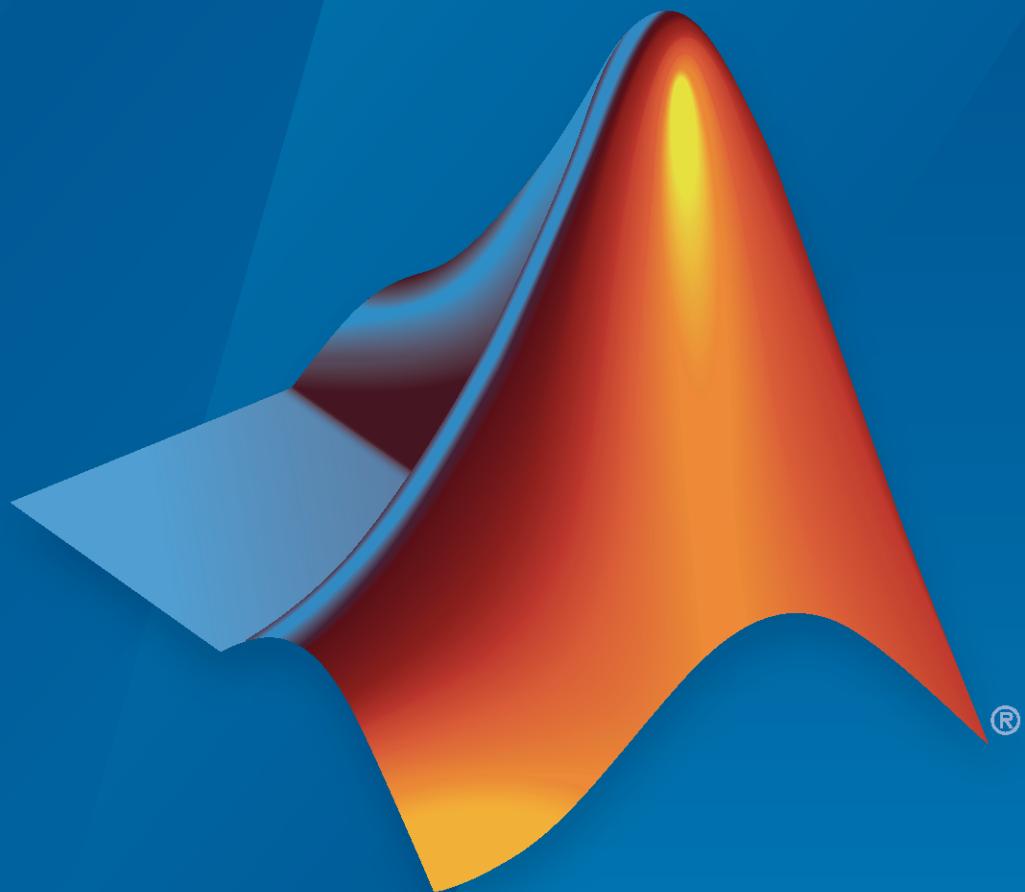


# Sensor Fusion and Tracking Toolbox™

## Reference



# MATLAB® & SIMULINK®

R2020b

 MathWorks®

# 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

*Sensor Fusion and Tracking Toolbox™ Reference Guide*

© COPYRIGHT 2018 – 2020 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

September 2018	Online only	New for Version 1.0 (Release 2018b)
March 2019	Online only	Revised for Version 1.1 (Release 2019a)
September 2019	Online only	Revised for Version 1.2 (Release 2019b)
March 2020	Online only	Revised for Version 1.3 (Release 2020a)
September 2020	Online only	Revised for Version 2.0 (Release 2020b)

<b>1</b>	<b>Functions</b>
<b>2</b>	<b>Classes</b>
<b>3</b>	<b>System Objects</b>
<b>4</b>	<b>Blocks</b>
<b>5</b>	<b>Apps</b>



# Functions

---

## allanvar

Allan variance

### Syntax

```
[avar,tau] = allanvar(Omega)
[avar,tau] = allanvar(Omega,m)
[avar,tau] = allanvar(Omega,ptStr)
[avar,tau] = allanvar(___,fs)
```

### Description

Allan variance is used to measure the frequency stability of oscillation for a sequence of data in the time domain. It can also be used to determine the intrinsic noise in a system as a function of the averaging time. The averaging time series  $\tau$  can be specified as  $\tau = m/fs$ . Here  $fs$  is the sampling frequency of data, and  $m$  is a list of ascending averaging factors (such as 1, 2, 4, 8, ...).

`[avar,tau] = allanvar(Omega)` returns the Allan variance `avar` as a function of averaging time `tau`. The default averaging time `tau` is an octave sequence given as  $(1, 2, \dots, 2^{\text{floor}\{\log_2[(N-1)/2]\}})$ , where  $N$  is the number of samples in `Omega`. If `Omega` is specified as a matrix, `allanvar` operates over the columns of `omega`.

`[avar,tau] = allanvar(Omega,m)` returns the Allan variance `avar` for specific values of `tau` defined by `m`. Since the default frequency `fs` is assumed to be 1, the output `tau` is exactly same with `m`.

`[avar,tau] = allanvar(Omega,ptStr)` sets averaging factor `m` to the specified point specification, `ptStr`. Since the default frequency `fs` is 1, the output `tau` is exactly equal to the specified `m`. `ptStr` can be specified as '`octave`' or '`decade`'.

`[avar,tau] = allanvar(___,fs)` also allows you to provide the sampling frequency `fs` of the input data `omega` in Hz. This input parameter can be used with any of the previous syntaxes.

### Examples

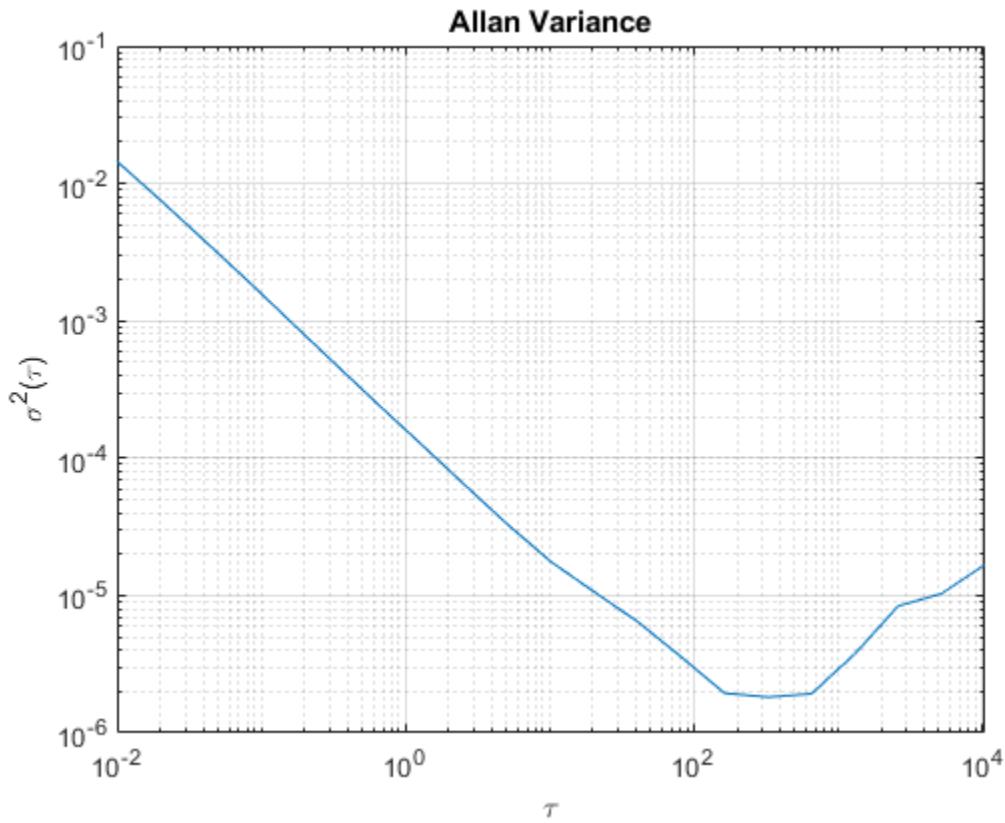
#### Determine Allan Variance of Single Axis Gyroscope

Load gyroscope data from a MAT file, including the sample rate of the data in Hz. Calculate the Allan variance.

```
load('LoggedSingleAxisGyroscope','omega','Fs')
[avar,tau] = allanvar(omega,'octave',Fs);
```

Plot the Allan variance on a loglog plot.

```
loglog(tau,avar)
xlabel('\tau')
ylabel('\sigma^2(\tau)')
title('Allan Variance')
grid on
```



### Determine Allan Deviation at Specific Values of $\tau$

Generate sample gyroscope noise, including angle random walk and rate random walk.

```
numSamples = 1e6;
Fs = 100;
nStd = 1e-3;
kStd = 1e-7;
nNoise = nStd.*randn(numSamples,1);
kNoise = kStd.*cumsum(randn(numSamples,1));
omega = nNoise+kNoise;
```

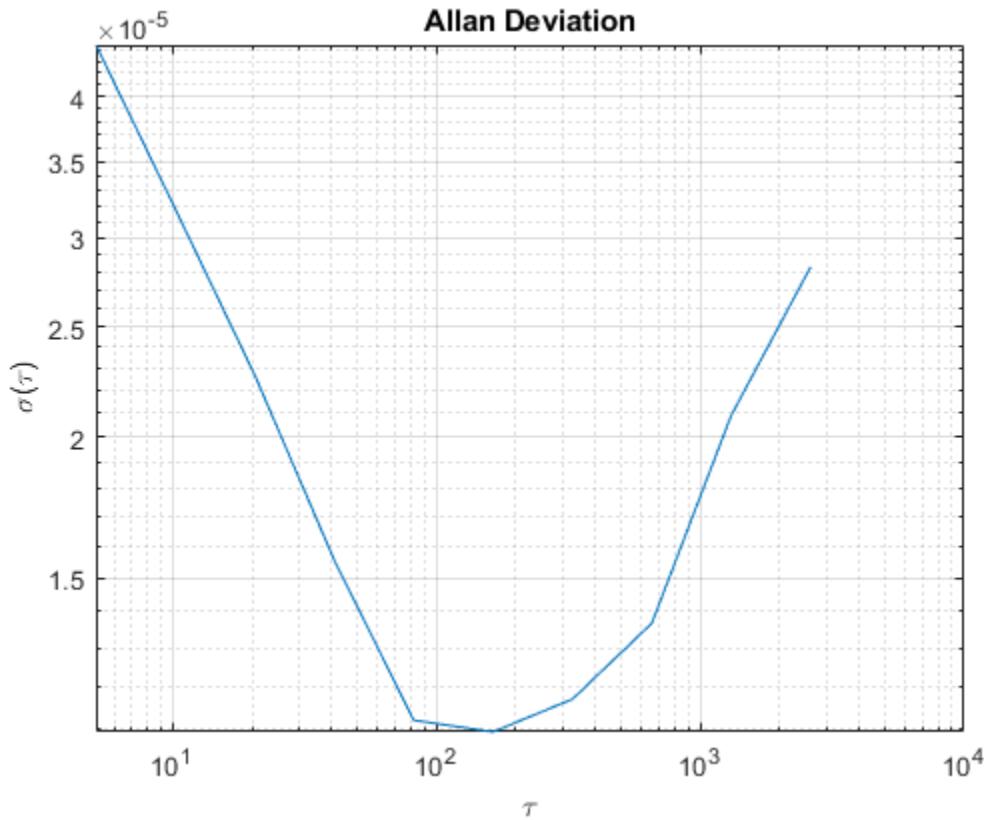
Calculate the Allan deviation at specific values of  $m = \tau$ . The Allan deviation is the square root of the Allan variance.

```
m = 2.^(9:18);
[avar,tau] = allanvar(omega,m,Fs);
adev = sqrt(avar);
```

Plot the Allan deviation on a loglog plot.

```
loglog(tau,adev)
xlabel('\tau')
ylabel('\sigma(\tau)')
```

```
title('Allan Deviation')
grid on
```



## Input Arguments

### **Omega — Input data**

N-by-1 vector | N-by-M matrix

Input data specified as an  $N$ -by-1 vector or an  $N$ -by- $M$  matrix.  $N$  is the number of samples, and  $M$  is the number of sample sets. If specified as a matrix, `allanvar` operates over the columns of `Omega`.

Data Types: `single` | `double`

### **m — Averaging factor**

scalar | vector

Averaging factor, specified as a scalar or vector with ascending integer values less than  $(N-1)/2$ , where  $N$  is the number of samples in `Omega`.

Data Types: `single` | `double`

### **ptStr — Point specification of m**

'octave' (default) | 'decade'

Point specification of `m`, specified as 'octave' or 'decade'. Based on the value of `ptStr`, `m` is specified as following:

- If `ptStr` is specified as 'octave',  $m$  is:

$$\left[ 2^0, 2^1 \dots 2^{\lfloor \log_2\left(\frac{N-1}{2}\right) \rfloor} \right]$$

- If `ptStr` is specified as 'decade',  $m$  is:

$$\left[ 10^0, 10^1 \dots 10^{\lfloor \log_{10}\left(\frac{N-1}{2}\right) \rfloor} \right]$$

$N$  is the number of samples in `Omega`.

**fs — Basic frequency of input data in Hz**

scalar

Basic frequency of the input data, `Omega`, in Hz, specified as a positive scalar.

Data Types: `single` | `double`

## Output Arguments

**avar — Allan variance of input data**

vector | matrix

Allan variance of input data at `tau`, returned as a vector or matrix.

**tau — Averaging time of Allan variance**

vector | matrix

Averaging time of Allan variance, returned as a vector, or a matrix.

## See Also

`gyroparams` | `imuSensor`

## Introduced in R2019a

## ctrect

Constant turn-rate rectangular target motion model

### Syntax

```
updatedstates = ctrect(states)
updatedstates = ctrect(states,dt)
updatedstates = ctrect(states,w,dt)
```

### Description

`updatedstates = ctrect(states)` returns the updated rectangular states from the current rectangular states based on the rectangular target motion model. The default time step is 1 second.

`updatedstates = ctrect(states,dt)` specifies the time step, `dt`, in seconds.

`updatedstates = ctrect(states,w,dt)` additionally specifies the process noise, `w`.

### Examples

#### Predict Constant Turn-Rate Rectangular State

Define a rectangular state.

```
state = [1 2 2 30 1 4.7 1.8];
```

Predict the state `dt = 1` second forward using the default syntax.

```
state = ctrect(state,0.1)
```

```
state = 1x7
```

```
1.1731    2.1002    2.0000   30.1000    1.0000    4.7000    1.8000
```

Predict the state `dt = 0.1` second forward without noise.

```
state = ctrect(state,0.1)
```

```
state = 1x7
```

```
1.3461    2.2006    2.0000   30.2000    1.0000    4.7000    1.8000
```

Predict the state `dt = 0.1` second forward with noise.

```
state = ctrect(state,0.01,0.1)
```

```
state = 1x7
```

```
1.5189    2.3014    2.0010   30.3000    1.0010    4.7000    1.8000
```

## Predict Multiple Constant Turn-Rate Rectangular States

Define a state matrix.

```
states = [1 3 4;-1 2 10;5 3 1.3;1 1.3 2.1;30 0 -30;4.7 3.4 4.5;1.8 2 3];
```

Predict the state  $dt = 1$  second ahead.

```
states = ctrect(states)
```

```
states = 7x3
```

5.7516	5.9992	5.2528
0.3625	2.0681	9.7131
5.0000	3.0000	1.3000
31.0000	1.3000	-27.9000
30.0000	0.0000	-30.0000
4.7000	3.4000	4.5000
1.8000	2.0000	3.0000

Predict the state  $dt = 0.1$  second ahead without noise.

```
states = ctrect(states,0.1)
```

```
states = 7x3
```

6.1732	6.2992	5.3660
0.6311	2.0749	9.6493
5.0000	3.0000	1.3000
34.0000	1.3000	-30.9000
30.0000	0.0000	-30.0000
4.7000	3.4000	4.5000
1.8000	2.0000	3.0000

Predict the state  $dt = 0.1$  second ahead with noise.

```
states = ctrect(states,0.1*randn(2,3),0.1)
```

```
states = 7x3
```

6.5805	6.5979	5.4759
0.9216	2.0816	9.5795
5.0054	2.9774	1.3032
37.0009	1.3004	-33.9007
30.0183	0.0086	-30.0131
4.7000	3.4000	4.5000
1.8000	2.0000	3.0000

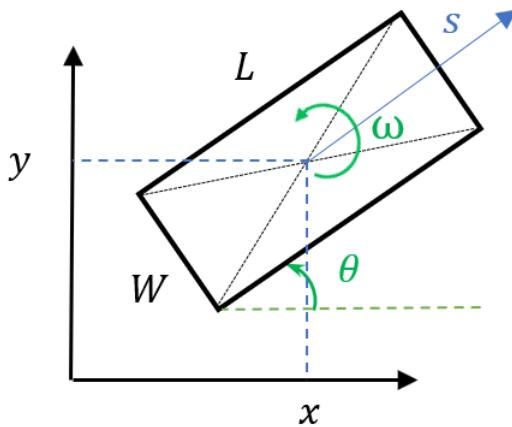
## Input Arguments

### states — Current rectangular states

1-by-7 real-valued vector | 7-by-1 real-valued vector | 7-by-N real-valued matrix

Current rectangular states, specified as a 1-by-7 real-valued vector, 7-by-1 real-valued vector, or a 7-by- $N$  real-valued matrix, where  $N$  is the number of states. The seven dimensional rectangular target state is defined as  $[x; y; s; \theta; \omega; L; W]$ :

Variable	Meaning	Unit
$x$	Position of the rectangle center in $x$ direction	m
$y$	Position of the rectangle center in $y$ direction	m
$s$	Speed in the heading direction	m/s
$\theta$	Orientation angle of the rectangle with respect to $x$ direction	degree
$\omega$	Turn-rate	degree/s
$L$	Length of the rectangle	m
$W$	Width of the rectangle	m



Example: [1;2;2;30;1;4.7;1.8]

Data Types: single | double

#### **dt — Time step**

real-valued positive scalar

Time step, specified as a real-valued positive scalar in second.

Data Types: single | double

#### **w — Process noise**

real scalar | 2-by- $N$  real-valued matrix

Process noise, specified as a 2-by- $N$  real-valued matrix, where  $N$  is the number of states specified in the **states** input. If specified as a scalar, it is expanded to a 2-by- $N$  matrix with all elements equal to

the scalar. The first row of the matrix specifies the process noise in acceleration ( $\text{m/s}^2$ ). The second row specifies the process noise in yaw acceleration (degrees/ $\text{s}^2$ ).

Data Types: `single` | `double`

## Output Arguments

### **updatedstates — Updated states**

1-by-7 real-valued vector | 7-by-1 real-valued vector | 7-by- $N$  real-valued matrix

Updated states, specified as a 1-by-7 real-valued vector, a 7-by-1 real-valued vector, or a 7-by- $N$  real-valued matrix, where  $N$  is the number of states. The dimensions and setups of `updatedstates` output are exactly the same as those of the `states` input.

Data Types: `single` | `double`

## Extended Capabilities

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

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

## See Also

`ctrectcorners` | `ctrectjac` | `ctrectmeas` | `ctrectmeasjac` | `gmphd` | `initctrectgmphd` | `trackerPHD`

## Introduced in R2019b

## ctrectmeas

Constant turn-rate rectangular target measurement model

### Syntax

```
measurements = ctrectmeas(states,detections)
```

### Description

measurements = ctrectmeas(states,detections) returns the expected measurements from the current rectangular states and detections.

### Examples

#### Expected Detections Using Rectangular Measurement Model

Load detections and truth generated from a rectangular target.

```
load('rectangularTargetDetections.mat','detections','truthState');
```

Generate expected detections from the target's rectangular state and actual detections using ctrectmeas.

```
tgtState = [3;48;0;60;0;5;1.9];
zExp = ctrectmeas(tgtState,detections);
```

Set up visualization environment using theaterPlot.

```
theaterP = theaterPlot;
stateP = trackPlotter(theaterP,'DisplayName','State','MarkerFaceColor','g');
truthP = trackPlotter(theaterP,'DisplayName','Truth','MarkerFaceColor','b');
detP = detectionPlotter(theaterP,'DisplayName','Detections','MarkerFaceColor','r');
expDetP = detectionPlotter(theaterP,'DisplayName','Expected Detections','MarkerFaceColor','y');
l = legend(theaterP.Parent);
l.AutoUpdate = 'on';
hold on;
assignP = plot(theaterP.Parent,NaN,NaN,'-.','DisplayName','Association');
```

Plot actual and expected detections.

```
inDets = [detections{:}];
inMeas = horzcat(inDets.Measurement);
detP.plotDetection(inMeas');

zExpPlot = reshape(zExp,3,[]);
expDetP.plotDetection(zExpPlot');
```

Plot association lines.

```
zLines = nan(2,numel(detections)*3);
zLines(1,1:3:end) = zExpPlot(1,:);
```

```

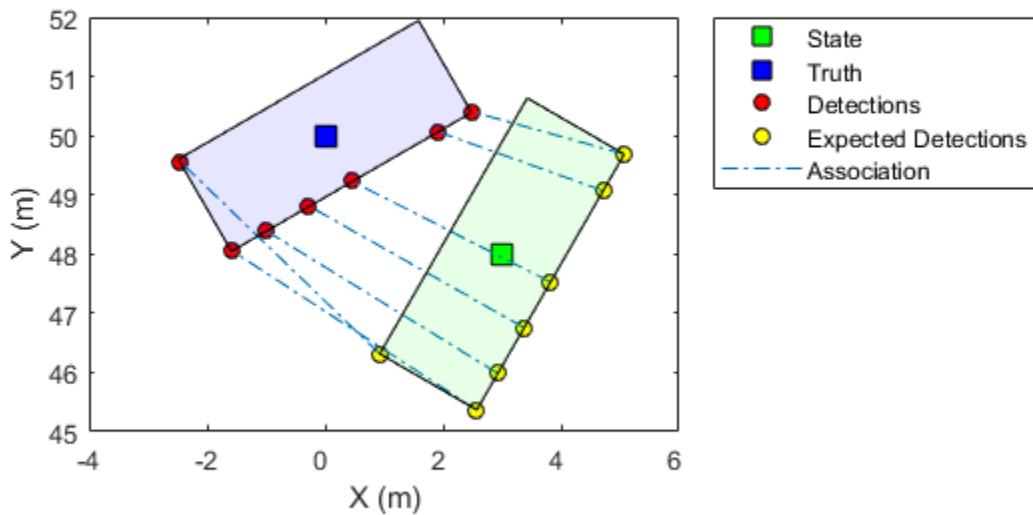
zLines(2,1:3:end) = zExpPlot(2,:);
zLines(1,2:3:end) = inMeas(1,:);
zLines(2,2:3:end) = inMeas(2,:);
assignP.XData = zLines(1,:);
assignP.YData = zLines(2,:);

Plot truth and state.

truthPos = [truthState(1:2);0];
truthDims = struct('Length',truthState(6),...
    'Width',truthState(7),...
    'Height', 0,...,
    'OriginOffset', [0 0 0]);
truthOrient = quaternion([truthState(4) 0 0],'eulerd', 'ZYX','frame');
truthP.plotTrack(truthPos',truthDims,truthOrient);

statePos = [tgtState(1:2);0];
stateDims = struct('Length',tgtState(6),...
    'Width',tgtState(7),...
    'Height',0,...,
    'OriginOffset', [0 0 0]);
stateOrient = quaternion([tgtState(4) 0 0],'eulerd', 'ZYX','frame');
stateP.plotTrack(statePos', stateDims, stateOrient);

```



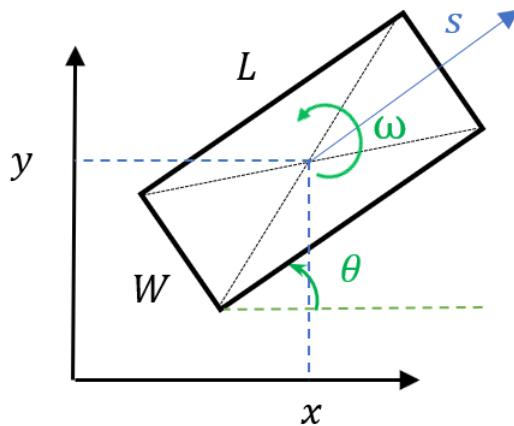
## Input Arguments

### states — Current rectangular states

7-by- $N$  real-valued matrix

Current rectangular states, specified as a 7-by- $N$  real-valued matrix, where  $N$  is the number of states. The seven-dimensional rectangular target state is defined as  $[x; y; s; \theta; \omega; L; W]$ :

Variable	Meaning	Unit
$x$	Position of the rectangle center in $x$ direction	m
$y$	Position of the rectangle center in $y$ direction	m
$s$	Speed in the heading direction	m/s
$\theta$	Orientation angle of the rectangle with respect to $x$ direction	degree
$\omega$	Turn-rate	degree/s
$L$	Length of the rectangle	m
$W$	Width of the rectangle	m



Example: [1;2;2;30;1;4.7;1.8]

Data Types: single | double

### detections — Detections of target

1-by- $M$  cell array of `objectDetection` objects

Detections of target, specified as a 1-by- $M$  cell array of `objectDetection` objects. The `MeasurementParameters` property (that specifies the transformation from the state-space to measurement-space) for each object must be the same for all the detections in the cell array.

## Output Arguments

### **measurements — Expected measurements**

*P*-by-*N*-by-*M* real-valued array

Expected measurements, returned as a *P*-by-*N*-by-*M* real-valued array. *P* is the dimension of each measurement specified in the detections input, *N* is the number of states specified in the states input, and *M* is the number of detections specified in the detections input.

## Extended Capabilities

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

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

### See Also

`ctrect` | `ctrectcorners` | `ctrectjac` | `ctrectmeasjac` | `gmphd` | `initctrectgmphd` | `trackerPHD`

**Introduced in R2019b**

## ctrectmeasjac

Jacobian of constant turn-rate rectangular target measurement model

### Syntax

```
jacobian = ctrectmeasjac(state,detections)
```

### Description

`jacobian = ctrectmeasjac(state,detections)` returns the Jacobian based on the current rectangular target state and detections.

### Examples

#### Generate Jacobian for Rectangular Target Model

Load detections generated from a rectangular target.

```
load('rectangularTargetDetections.mat','detections');
```

Calculate Jacobian based on the rectangular state of the target and detections.

```
tgtState = [3;48;0;60;0;5;1.9];
jac = ctrectmeasjac(tgtState,detections);
jac1 = jac(:,:,1)

jac1 = 3x7
```

1.0000	0	0	0.0461	0	-0.2500	0.4330
0	1.0000	0	-0.0075	0	-0.4330	-0.2500
0	0	0	0	0	0	0

### Input Arguments

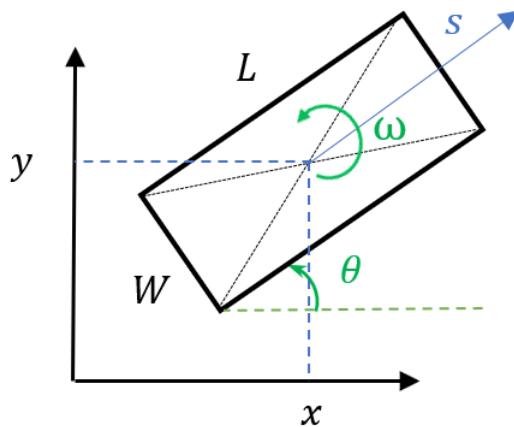
#### state — Current rectangular target state

7-by-1 real-valued vector

Current rectangular target state, specified as a 7-by-1 real-valued vector. The seven dimensional rectangular target state is defined as  $[x; y; s; \theta; \omega; L; W]$ . The meaning of these variables and their units are:

Variable	Meaning	Unit
x	Position of the rectangle center in x direction	m
y	Position of the rectangle center in y direction	m

$s$	Speed in the heading direction	m/s
$\theta$	Orientation angle of the rectangle with respect to $x$ direction	degree
$\omega$	Turn-rate	degree/s
$L$	Length of the rectangle	m
$W$	Width of the rectangle	m



Example: [1;2;2;30;1;4.7;1.8]

Data Types: single | double

#### **detections — Detections of target**

1-by- $M$  cell array of `objectDetection` objects

Detections of target, specified as a 1-by- $M$  cell array of `objectDetection` objects. The `MeasurementParameters` property (that specifies the transformation from the state-space to measurement-space) for each object must be the same for all the detections in the cell array.

## **Output Arguments**

#### **jacobian — Jacobian of measurement model**

$M$ -by-7-by- $D$  real-valued array

Jacobian of measurement model, returned as a  $M$ -by-7-by- $D$  real-valued array.  $M$  is the dimension of each measurement specified in `detections`, and  $D$  is the number of detections specified in the `detections` input.

## **Extended Capabilities**

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

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

**See Also**

`crect` | `crectcorners` | `crectjac` | `crectmeas` | `gmphd` | `initcrectgmphd` | `trackerPHD`

**Introduced in R2019b**

# ctrectjac

Jacobian of constant turn-rate rectangular target motion model

## Syntax

```
Jx = ctrectjac(state)
Jx = ctrectjac(state,dt)
[Jx,Jw] = ctrectjac(state,w,dt)
```

## Description

`Jx = ctrectjac(state)` returns the Jacobian matrix of the constant turn-rate rectangular motion model with respect to the state vector. The default time step is 1 second.

`Jx = ctrectjac(state,dt)` specifies the time step `dt` in seconds.

`[Jx,Jw] = ctrectjac(state,w,dt)` also specifies the process noise `w`.

## Examples

### Jacobian of Constant Turn-Rate Rectangular Motion Model

Define a state vector for the model.

```
state = [1;2;2;30;1;4.7;1.8];
```

Compute the Jacobian. `dt` = 1 second.

```
jac = ctrectjac(state)
```

`jac` = 7×7

1.0000	0	0.8616	-0.0177	-0.0089	0	0
0	1.0000	0.5075	0.0301	0.0150	0	0
0	0	1.0000	0	0	0	0
0	0	0	1.0000	1.0000	0	0
0	0	0	0	1.0000	0	0
0	0	0	0	0	1.0000	0
0	0	0	0	0	0	1.0000

Compute the Jacobian. `dt` = 0.1 second without noise.

```
jac = ctrectjac(state,0.1)
```

`jac` = 7×7

1.0000	0	0.0866	-0.0017	-0.0001	0	0
0	1.0000	0.0501	0.0030	0.0002	0	0
0	0	1.0000	0	0	0	0
0	0	0	1.0000	0.1000	0	0

```

0      0      0      0    1.0000      0      0
0      0      0      0      0    1.0000      0
0      0      0      0      0      0    1.0000

```

Compute the Jacobian.  $dt = 0.1$  second with noise.

```
jac = ctrectjac(state,0.01,0.1)
```

```
jac = 7x7
```

```

1.0000      0      0.0866   -0.0017   -0.0001      0      0
0      1.0000   0.0501   0.0030   0.0002      0      0
0      0      1.0000      0      0      0      0
0      0      0      1.0000   0.1000      0      0
0      0      0      0      1.0000      0      0
0      0      0      0      0      1.0000      0
0      0      0      0      0      0      1.0000

```

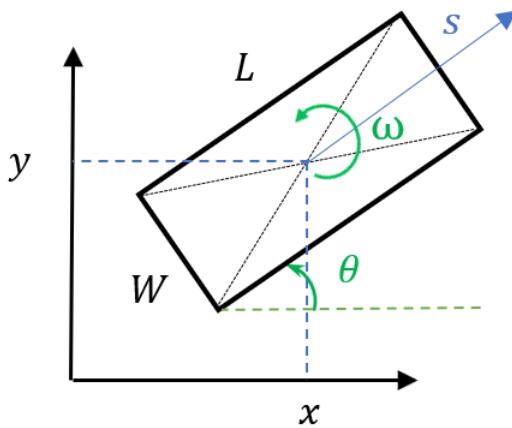
## Input Arguments

### state — Current state

1-by-7 real-valued vector

Current state, specified as a 1-by-7 real-valued vector. The state of the constant-turn rectangular target model is  $[x; y; s; \theta; \omega; L; W]$ . The meaning of these variables and their units are:

Variable	Meaning	Unit
x	Position of the rectangle center in x direction	m
y	Position of the rectangle center in y direction	m
s	Speed in the heading direction	m/s
$\theta$	Orientation angle of the rectangle with respect to x direction	degree
$\omega$	Turn-rate	degree/s
L	Length of the rectangle	m
W	Width of the rectangle	m



Example: [1;2;2;30;1;4.7;1.8]

Data Types: single | double

#### **dt — Time step**

real-valued positive scalar

Time step, specified as a real-valued positive scalar in second.

Data Types: single | double

#### **w — Process noise**

real scalar | 2-element real-valued vector

Process noise, specified as a 2-element real-valued vector. The first element specifies the process noise in linear acceleration ( $\text{m/s}^2$ ). The second element specifies the process noise in yaw acceleration ( $\text{degrees/s}^2$ ).

Data Types: single | double

## Output Arguments

#### **Jx — Jacobian matrix with respect to state**

7-by-7 matrix

Jacobian matrix with respect to state, returned as a 7-by-7 matrix.

Data Types: double

#### **Jw — Jacobian with respect to process noise**

7-by-2 matrix

Jacobian with respect to process noise, returned as a 7-by-2 matrix.

Data Types: double

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`crect` | `crectcorners` | `crectmeas` | `crectmeasjac` | `gmphd` | `initcrectgmphd` | `trackerPHD`

**Introduced in R2019b**

# jpdaEvents

Feasible joint events for trackerJPDA

## Syntax

```
FJE = jpdaEvents(validationMatrix)
```

## Description

`FJE = jpdaEvents(validationMatrix)` returns the feasible joint events, `FJE`, based on the validation matrix. A validation matrix describes the possible associations between detections and tracks, whereas a feasible joint event for multi-object tracking is one realization of the associations between detections and tracks.

## Examples

### Generate Feasible Joint Events

Define an arbitrary validation matrix for five measurements and six tracks.

```
M = [1     1     1     1     1     0     1
      1     0     1     1     0     0     0
      1     0     0     0     1     1     0
      1     1     1     1     0     0     0
      1     1     1     1     1     1     1];
```

Generate all feasible joint events and count the total number.

```
FJE = jpdaEvents(M);
nFJE = size(FJE,3);
```

Display a few of the feasible joint events.

```
disp([num2str(nFJE) ' feasible joint event matrices were generated.'])
```

574 feasible joint event matrices were generated.

```
toSee = [1:round(nFJE/5):nFJE, nFJE];
for ii = toSee
    disp("Feasible joint event matrix #" + ii + ":")
    disp(FJE(:,:,ii))
end
```

Feasible joint event matrix #1:

```
1  0  0  0  0  0  0
1  0  0  0  0  0  0
1  0  0  0  0  0  0
1  0  0  0  0  0  0
1  0  0  0  0  0  0
```

Feasible joint event matrix #116:

0	0	1	0	0	0	0
1	0	0	0	0	0	0
0	0	0	0	1	0	0
0	1	0	0	0	0	0
0	0	0	1	0	0	0

Feasible joint event matrix #231:

0	0	0	0	1	0	0
0	0	1	0	0	0	0
0	0	0	0	0	1	0
1	0	0	0	0	0	0
0	0	0	0	0	0	1

Feasible joint event matrix #346:

0	0	0	0	0	0	1
0	0	0	1	0	0	0
0	0	0	0	1	0	0
1	0	0	0	0	0	0
0	1	0	0	0	0	0

Feasible joint event matrix #461:

1	0	0	0	0	0	0
0	0	1	0	0	0	0
1	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	1

Feasible joint event matrix #574:

1	0	0	0	0	0	0
1	0	0	0	0	0	0
1	0	0	0	0	0	0
1	0	0	0	0	0	0
0	0	0	0	0	0	1

## Input Arguments

### **validationMatrix — Validation Matrix**

*m*-by-(*n*+1) matrix

Validation matrix, specified as an *m*-by-(*n*+1) matrix, where *m* is the number of detections within a cluster of a sensor scan, and *n* is the number of tracks maintained in the tracker. The validation matrix uses the first column to account for the possibility that each detection is clutter or false alarm, which is commonly referred to as "Track 0" or  $T_0$ . The validation matrix is a binary matrix listing all possible detections-to-track associations. If it is possible to assign track  $T_i$  to detection  $D_j$ , then the (*j*, *i*+1) entry of the validation matrix is 1. Otherwise, the entry is 0.

Data Types: logical

## Output Arguments

### FJE — Feasible joint events

*m*-by- $(n+1)$ -by-*p* array

Feasible joint events, specified as an  $m$ -by- $(n+1)$ -by-*p* array, where  $m$  is the number of detections within a cluster of a sensor scan,  $n$  is the number of tracks maintained in the tracker, and *p* is the total number of feasible joint events. Each page (an  $m$ -by- $(n+1)$  matrix) of FJE corresponds to one possible association between all the tracks and detections. The feasible joint event matrix on each page satisfies:

- The matrix has exactly one "1" value per row.
- Except for the first column, which maps to clutter, there can be at most one "1" per column.

For more details on feasible joint events, see “Feasible Joint Events” on page 1-23.

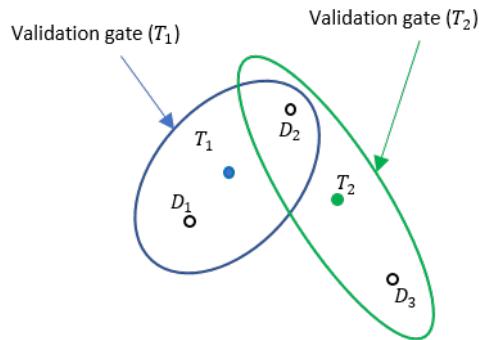
Data Types: logical

## More About

### Feasible Joint Events

In the typical workflow for a tracking system, the tracker needs to determine if a detection can be associated with any of the existing tracks. If the tracker only maintains one track, the assignment can be done by evaluating the validation gate around the predicted measurement and deciding if the measurement falls within the *validation gate*. In the measurement space, the validation gate is a spatial boundary, such as a 2-D ellipse or a 3-D ellipsoid, centered at the predicted measurement. The validation gate is defined using the probability information (state estimation and covariance, for example) of the existing track, such that the correct or ideal detections have high likelihood (97% probability, for example) of falling within this validation gate.

However, if a tracker maintains multiple tracks, the data association process becomes more complicated, because one detection can fall within the validation gates of multiple tracks. For example, in the following figure, tracks  $T_1$  and  $T_2$  are actively maintained in the tracker, and each of them has its own validation gate. Since the detection  $D_2$  is in the intersection of the validation gates of both  $T_1$  and  $T_2$ , the two tracks ( $T_1$  and  $T_2$ ) are connected and form a *cluster*. A cluster is a set of connected tracks and their associated detections.



To represent the association relationship in a cluster, the validation matrix is commonly used. Each row of the validation matrix corresponds to a detection while each column corresponds to a track. To account for the eventuality of each detection being clutter, a first column is added and usually

referred to as "Track 0" or  $T_0$ . If detection  $D_i$  is inside the validation gate of track  $D_j$ , then the  $(j, i+1)$  entry of the validation matrix is 1. Otherwise, it is zero. For the cluster shown in the figure, the validation matrix  $\Omega$  is

$$\Omega = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Note that all the elements in the first column of  $\Omega$  are 1, because any detection can be clutter or false alarm. One important step in the logic of joint probabilistic data association (JPDA) is to obtain all the feasible independent joint events in a cluster. Two assumptions for the feasible joint events are:

- A detection cannot be emitted by more than one track.
- A track cannot be detected more than once by the sensor during a single scan.

Based on these two assumptions, feasible joint events (FJEs) can be formulated. Each FJE is mapped to an FJE matrix  $\Omega_p$  from the initial validation matrix  $\Omega$ . For example, with the validation matrix  $\Omega$ , eight FJE matrices can be obtained:

$$\Omega_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \Omega_2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \Omega_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \Omega_4 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

$$\Omega_5 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \quad \Omega_6 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \Omega_7 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \Omega_8 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

As a direct consequence of the two assumptions, the  $\Omega_p$  matrices have exactly one "1" value per row. Also, except for the first column which maps to clutter, there can be at most one "1" per column. When the number of connected tracks grows in a cluster, the number of FJE increases rapidly. The `jpdEvents` function uses an efficient depth-first search algorithm to generate all the feasible joint event matrices.

## References

- [1] Zhou, Bin, and N. K. Bose. "Multitarget tracking in clutter: Fast algorithms for data association." *IEEE Transactions on aerospace and electronic systems* 29, no. 2 (1993): 352-363.
- [2] Fisher, James L., and David P. Casasent. "Fast JPDA multitarget tracking algorithm." *Applied optics* 28, no. 2 (1989): 371-376.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

- This function only supports double precision code generation.

## See Also

`trackerJPDA`

**Introduced in R2019a**

# tunernoise

Noise structure of fusion filter

## Syntax

```
noiseStruct = tunernoise(filterName)
```

## Description

`noiseStruct = tunernoise(filterName)` returns the measurement noise structure for the filter with name specified by the `filterName` input.

## Examples

### Obtain Measurement Noise Structure of `insfilterAsync`

Obtain the measurement noise structure of the `insfilterAsync` object.

```
noiseStruct = tunernoise('insfilterAsync')

noiseStruct = struct with fields:
    AccelerometerNoise: 1
        GyroscopeNoise: 1
    MagnetometerNoise: 1
        GPSPositionNoise: 1
        GPSVelocityNoise: 1
```

### Tune `insfilterAsync` to Optimize Pose Estimate

Load the recorded sensor data and ground truth data.

```
load('insfilterAsyncTuneData.mat');
```

Create timetables for the sensor data and the truth data.

```
sensorData = timetable(Accelerometer, Gyroscope, ...
    Magnetometer, GPSPosition, GPSVelocity, 'SampleRate', 100);
groundTruth = timetable(Orientation, Position, ...
    'SampleRate', 100);
```

Create an `insfilterAsync` filter object that has a few noise properties.

```
filter = insfilterAsync('State', initialState, ...
    'StateCovariance', initialStateCovariance, ...
    'AccelerometerBiasNoise', 1e-7, ...
    'GyroscopeBiasNoise', 1e-7, ...
    'MagnetometerBiasNoise', 1e-7, ...
    'GeomagneticVectorNoise', 1e-7);
```

Create a tuner configuration object for the filter. Set the maximum iterations to two. Also, set the tunable parameters as the unspecified properties.

```
config = tunerconfig('insfilterAsync','MaxIterations',8);
config.TunableParameters = setdiff(config.TunableParameters, ...
    {'GeomagneticVectorNoise', 'AccelerometerBiasNoise', ...
    'GyroscopeBiasNoise', 'MagnetometerBiasNoise'});
config.TunableParameters

ans = 1×10 string
    "AccelerationNoise"    "AccelerometerNoise"    "AngularVelocityNoise"    "GPSPositionNoise"
```

Use the tuner noise function to obtain a set of initial sensor noises used in the filter.

```
measNoise = tunernoise('insfilterAsync')

measNoise = struct with fields:
    AccelerometerNoise: 1
        GyroscopeNoise: 1
    MagnetometerNoise: 1
        GPSPositionNoise: 1
        GPSVelocityNoise: 1
```

Tune the filter and obtain the tuned parameters.

```
tunedParams = tune(filter,measNoise,sensorData,groundTruth,config);
```

Iteration	Parameter	Metric
1	AccelerationNoise	2.1345
1	AccelerometerNoise	2.1264
1	AngularVelocityNoise	1.9659
1	GPSPositionNoise	1.9341
1	GPSVelocityNoise	1.8420
1	GyroscopeNoise	1.7589
1	MagnetometerNoise	1.7362
1	PositionNoise	1.7362
1	QuaternionNoise	1.7218
1	VelocityNoise	1.7218
2	AccelerationNoise	1.7190
2	AccelerometerNoise	1.7170
2	AngularVelocityNoise	1.6045
2	GPSPositionNoise	1.5948
2	GPSVelocityNoise	1.5323
2	GyroscopeNoise	1.4803
2	MagnetometerNoise	1.4703
2	PositionNoise	1.4703
2	QuaternionNoise	1.4632
2	VelocityNoise	1.4632
3	AccelerationNoise	1.4596
3	AccelerometerNoise	1.4548
3	AngularVelocityNoise	1.3923
3	GPSPositionNoise	1.3810
3	GPSVelocityNoise	1.3322
3	GyroscopeNoise	1.2998
3	MagnetometerNoise	1.2976
3	PositionNoise	1.2976

3	QuaternionNoise	1.2943
3	VelocityNoise	1.2943
4	AccelerationNoise	1.2906
4	AccelerometerNoise	1.2836
4	AngularVelocityNoise	1.2491
4	GPSPositionNoise	1.2258
4	GPSVelocityNoise	1.1880
4	GyroscopeNoise	1.1701
4	MagnetometerNoise	1.1698
4	PositionNoise	1.1698
4	QuaternionNoise	1.1688
4	VelocityNoise	1.1688
5	AccelerationNoise	1.1650
5	AccelerometerNoise	1.1569
5	AngularVelocityNoise	1.1454
5	GPSPositionNoise	1.1100
5	GPSVelocityNoise	1.0778
5	GyroscopeNoise	1.0709
5	MagnetometerNoise	1.0675
5	PositionNoise	1.0675
5	QuaternionNoise	1.0669
5	VelocityNoise	1.0669
6	AccelerationNoise	1.0634
6	AccelerometerNoise	1.0549
6	AngularVelocityNoise	1.0549
6	GPSPositionNoise	1.0180
6	GPSVelocityNoise	0.9866
6	GyroscopeNoise	0.9810
6	MagnetometerNoise	0.9775
6	PositionNoise	0.9775
6	QuaternionNoise	0.9768
6	VelocityNoise	0.9768
7	AccelerationNoise	0.9735
7	AccelerometerNoise	0.9652
7	AngularVelocityNoise	0.9652
7	GPSPositionNoise	0.9283
7	GPSVelocityNoise	0.8997
7	GyroscopeNoise	0.8947
7	MagnetometerNoise	0.8920
7	PositionNoise	0.8920
7	QuaternionNoise	0.8912
7	VelocityNoise	0.8912
8	AccelerationNoise	0.8885
8	AccelerometerNoise	0.8811
8	AngularVelocityNoise	0.8807
8	GPSPositionNoise	0.8479
8	GPSVelocityNoise	0.8238
8	GyroscopeNoise	0.8165
8	MagnetometerNoise	0.8165
8	PositionNoise	0.8165
8	QuaternionNoise	0.8159
8	VelocityNoise	0.8159

Fuse the sensor data using the tuned filter.

```
dt = seconds(diff(groundTruth.Time));
N = size(sensorData,1);
qEst = quaternion.zeros(N,1);
```

```

posEst = zeros(N,3);
% Iterate the filter for prediction and correction using sensor data.
for ii=1:N
    if ii ~= 1
        predict(filter, dt(ii-1));
    end
    if all(~isnan(Accelerometer(ii,:)))
        fuseaccel(filter, Accelerometer(ii,:), ...
            tunedParams.AccelerometerNoise);
    end
    if all(~isnan(Gyroscope(ii,:)))
        fusegyro(filter, Gyroscope(ii,:), ...
            tunedParams.GyroscopeNoise);
    end
    if all(~isnan(Magnetometer(ii,1)))
        fusemag(filter, Magnetometer(ii,:), ...
            tunedParams.MagnetometerNoise);
    end
    if all(~isnan(GPSPosition(ii,1)))
        fusegps(filter, GPSPosition(ii,:), ...
            tunedParams.GPSPositionNoise, GPSVelocity(ii,:), ...
            tunedParams.GPSVelocityNoise);
    end
    [posEst(ii,:), qEst(ii,:)] = pose(filter);
end

```

Compute the RMS errors.

```

orientationError = rad2deg(dist(qEst, Orientation));
rmsorientationError = sqrt(mean(orientationError.^2))

rmsorientationError = 2.7801

positionError = sqrt(sum((posEst - Position).^2, 2));
rmspositionError = sqrt(mean(positionError.^2))

rmspositionError = 0.5966

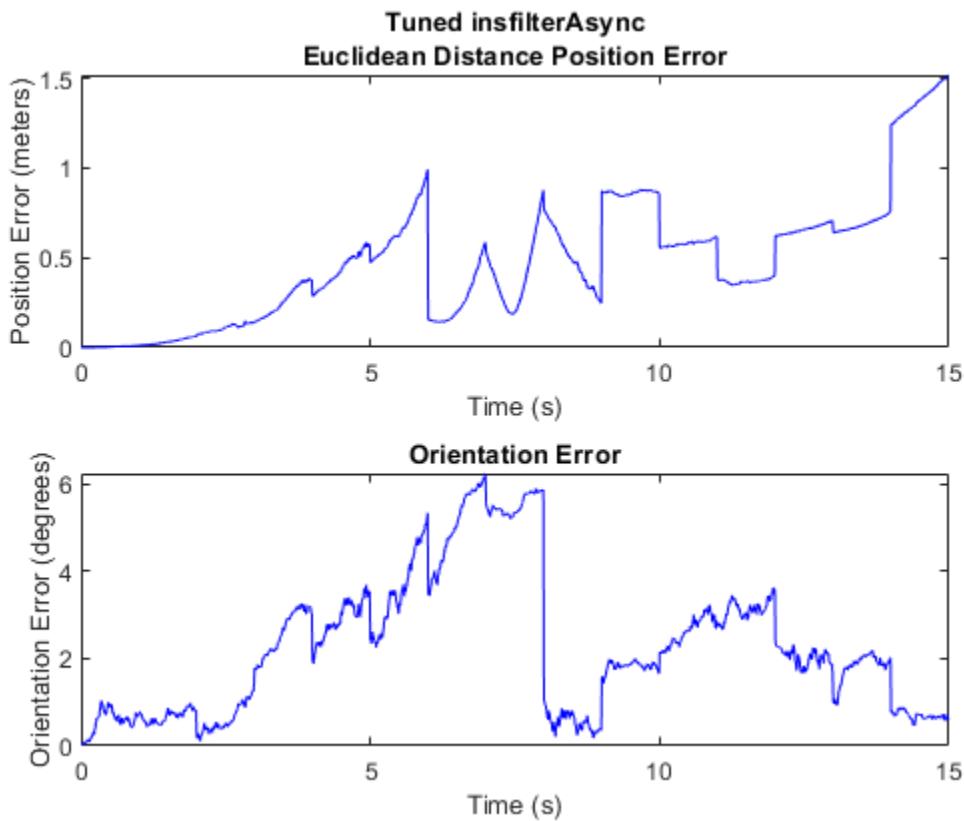
```

Visualize the results.

```

figure();
t = (0:N-1). / groundTruth.Properties.SampleRate;
subplot(2,1,1)
plot(t, positionError, 'b');
title("Tuned insfilterAsync" + newline + "Euclidean Distance Position Error")
xlabel('Time (s)');
ylabel('Position Error (meters)')
subplot(2,1,2)
plot(t, orientationError, 'b');
title("Orientation Error")
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');

```



## Input Arguments

**filterName — Name of fusion filter**  
**'insfilterAsync'**

Name of fusion filter, specified as 'insfilterAsync'.

## Output Arguments

**noiseStruct — Structure of measurement noise**  
structure

Structure of measurement noise, returned as a structure. For the `insfilterAsync` object, the structure contains these fields.

Field	Description	Default
AccelerometerNoise	Variance of accelerometer noise, specified as a scalar in $(\text{m}^2/\text{s})^2$	1
GyroscopeNoise	Variance of gyroscope noise, specified as a scalar in $(\text{rad}/\text{s})^2$	1

Field	Description	Default
MagnetometerNoise	Variance of magnetometer noise, specified as a scalar in $(\mu\text{T})^2$ .	1
GPSPositionNoise	Variance of GPS position noise, specified as a scalar in $\text{m}^2$	1
GPSVelocityNoise	Standard deviation of GPS velocity noise, specified as a scalar in $(\text{m/s})^2$	1

To use this structure with the `tune` function, change the values of the noise to proper values as initial guesses for tuning the noise.

## See Also

**Introduced in R2020b**

# insfilter

Create inertial navigation filter

## Syntax

```
filter = insfilter  
filter = insfilter('ReferenceFrame',RF)
```

## Description

`filter = insfilter` returns an `insfilterMARG` inertial navigation filter object that estimates pose based on accelerometer, gyroscope, GPS, and magnetometer measurements. See `insfilterMARG` for more details.

`filter = insfilter('ReferenceFrame',RF)` returns an `insfilterMARG` inertial navigation filter object that estimates pose relative to a reference frame specified by RF. Specify RF as 'NED' (North-East-Down) or 'ENU' (East-North-Up). The default value is 'NED'. See `insfilterMARG` for more details.

## Examples

### Create Default INS Filter

The default INS filter is the `insfilterMARG` object. Call `insfilter` with no input arguments to create the default INS filter.

```
filter = insfilter  
  
filter =  
  insfilterMARG with properties:  
  
    IMUSampleRate: 100          Hz  
    ReferenceLocation: [0 0 0]      [deg deg m]  
      State: [22x1 double]  
      StateCovariance: [22x22 double]  
  
    Multiplicative Process Noise Variances  
      GyroscopeNoise: [1e-09 1e-09 1e-09]      (rad/s)2  
      AccelerometerNoise: [0.0001 0.0001 0.0001]      (m/s2)2  
      GyroscopeBiasNoise: [1e-10 1e-10 1e-10]      (rad/s)2  
      AccelerometerBiasNoise: [0.0001 0.0001 0.0001]      (m/s2)2  
  
    Additive Process Noise Variances  
      GeomagneticVectorNoise: [1e-06 1e-06 1e-06]      uT2  
      MagnetometerBiasNoise: [0.1 0.1 0.1]      uT2
```

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`ahrsfilter` | `imufilter` | `insfilterAsync` | `insfilterErrorState` | `insfilterMARG` | `insfilterNonholonomic`

### Topics

“Estimate Position and Orientation of a Ground Vehicle”

### Introduced in R2018b

## ecompass

Orientation from magnetometer and accelerometer readings

### Syntax

```
orientation = ecompass(accelerometerReading,magnetometerReading)
orientation = ecompass(accelerometerReading,magnetometerReading,
orientationFormat)
orientation = ecompass(accelerometerReading,magnetometerReading,
orientationFormat,'ReferenceFrame',RF)
```

### Description

`orientation = ecompass(accelerometerReading,magnetometerReading)` returns a quaternion that can rotate quantities from a parent (NED) frame to a child (sensor) frame.

`orientation = ecompass(accelerometerReading,magnetometerReading,
orientationFormat)` specifies the orientation format as quaternion or rotation matrix.

`orientation = ecompass(accelerometerReading,magnetometerReading,
orientationFormat,'ReferenceFrame',RF)` also allows you to specify the reference frame RF of the orientation output. Specify RF as 'NED' (North-East-Down) or 'ENU' (East-North-Up). The default value is 'NED'.

### Examples

#### Determine Declination of Boston

Use the known magnetic field strength and proper acceleration of a device pointed true north in Boston to determine the magnetic declination of Boston.

Define the known acceleration and magnetic field strength in Boston.

```
magneticFieldStrength = [19.535 -5.109 47.930];
properAcceleration = [0 0 9.8];
```

Pass the magnetic field strength and acceleration to the `ecompass` function. The `ecompass` function returns a quaternion rotation operator. Convert the quaternion to Euler angles in degrees.

```
q = ecompass(properAcceleration,magneticFieldStrength);
e = eulerd(q,'ZYX','frame');
```

The angle, `e`, represents the angle between true north and magnetic north in Boston. By convention, magnetic declination is negative when magnetic north is west of true north. Negate the angle to determine the magnetic declination.

```
magneticDeclinationOfBoston = -e(1)
magneticDeclinationOfBoston = -14.6563
```

## Return Rotation Matrix

The `ecompass` function fuses magnetometer and accelerometer data to return a quaternion that, when used within a quaternion rotation operator, can rotate quantities from a parent (NED) frame to a child frame. The `ecompass` function can also return rotation matrices that perform equivalent rotations as the quaternion operator.

Define a rotation that can take a parent frame pointing to magnetic north to a child frame pointing to geographic north. Define the rotation as both a quaternion and a rotation matrix. Then, convert the quaternion and rotation matrix to Euler angles in degrees for comparison.

Define the magnetic field strength in microteslas in Boston, MA, when pointed true north.

```
m = [19.535 -5.109 47.930];
a = [0 0 9.8];
```

Determine the quaternion and rotation matrix that is capable of rotating a frame from magnetic north to true north. Display the results for comparison.

```
q = compass(a,m);
quaternionEulerAngles = eulerd(q,'ZYX','frame')

quaternionEulerAngles = 1x3
14.6563         0         0

r = compass(a,m,'rotmat');
theta = -asin(r(1,3));
psi = atan2(r(2,3)/cos(theta),r(3,3)/cos(theta));
rho = atan2(r(1,2)/cos(theta),r(1,1)/cos(theta));
rotmatEulerAngles = rad2deg([rho,theta,psi])

rotmatEulerAngles = 1x3
14.6563         0         0
```

## Determine Gravity Vector

Use `ecompass` to determine the gravity vector based on data from a rotating IMU.

Load the inertial measurement unit (IMU) data.

```
load 'rpy_9axis.mat' sensorData Fs
```

Determine the orientation of the sensor body relative to the local NED frame over time.

```
orientation = compass(sensorData.Acceleration,sensorData.MagneticField);
```

To estimate the gravity vector, first rotate the accelerometer readings from the sensor body frame to the NED frame using the `orientation` quaternion vector.

```
gravityVectors = rotatepoint(orientation,sensorData.Acceleration);
```

Determine the gravity vector as an average of the recovered gravity vectors over time.

```
gravityVectorEstimate = mean(gravityVectors,1)
```

```
gravityVectorEstimate = 1x3
```

```
0.0000 -0.0000 10.2102
```

## Track Spinning Platform

Fuse modeled accelerometer and gyroscope data to track a spinning platform using both idealized and realistic data.

### Generate Ground-Truth Trajectory

Describe the ground-truth orientation of the platform over time. Use the `kinematicTrajectory` System object™ to create a trajectory for a platform that has no translation and spins about its z-axis.

```
duration = 12;
fs = 100;
numSamples = fs*duration;

accelerationBody = zeros(numSamples,3);

angularVelocityBody = zeros(numSamples,3);
zAxisAngularVelocity = [linspace(0,4*pi,4*fs),4*pi*ones(1,4*fs),linspace(4*pi,0,4*fs)]';
angularVelocityBody(:,3) = zAxisAngularVelocity;

trajectory = kinematicTrajectory('SampleRate',fs);
[~,orientationNED,~,accelerationNED,angularVelocityNED] = trajectory(accelerationBody,angularVel
```

## Model Receiving IMU Data

Use an `imuSensor` System object to mimic data received from an IMU that contains an ideal magnetometer and an ideal accelerometer.

```
IMU = imuSensor('accel-mag','SampleRate',fs);
[accelerometerData,magnetometerData] = IMU(accelerationNED, ...
                                              angularVelocityNED, ...
                                              orientationNED);
```

## Fuse IMU Data to Estimate Orientation

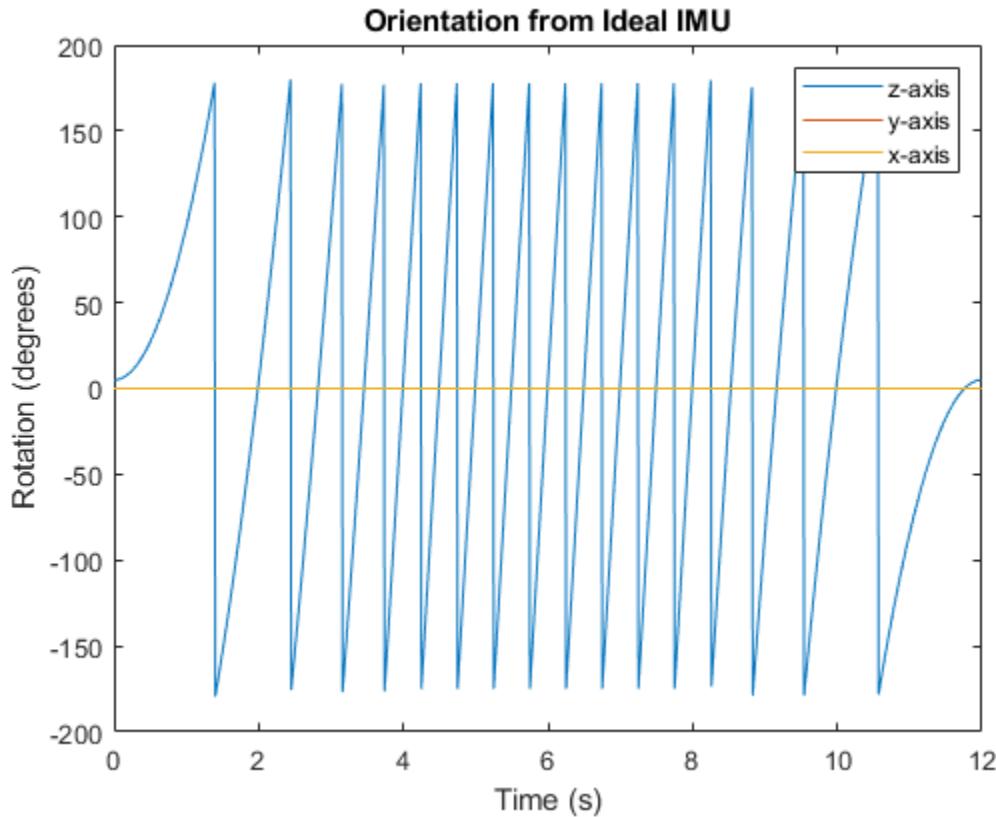
Pass the accelerometer data and magnetometer data to the `ecompass` function to estimate orientation over time. Convert the orientation to Euler angles in degrees and plot the result.

```
orientation = ecompass(accelerometerData,magnetometerData);
orientationEuler = eulerd(orientation,'ZYX','frame');

timeVector = (0:numSamples-1).'/fs;

figure(1)
```

```
plot(timeVector,orientationEuler)
legend('z-axis','y-axis','x-axis')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation from Ideal IMU')
```



### Repeat Experiment with Realistic IMU Sensor Model

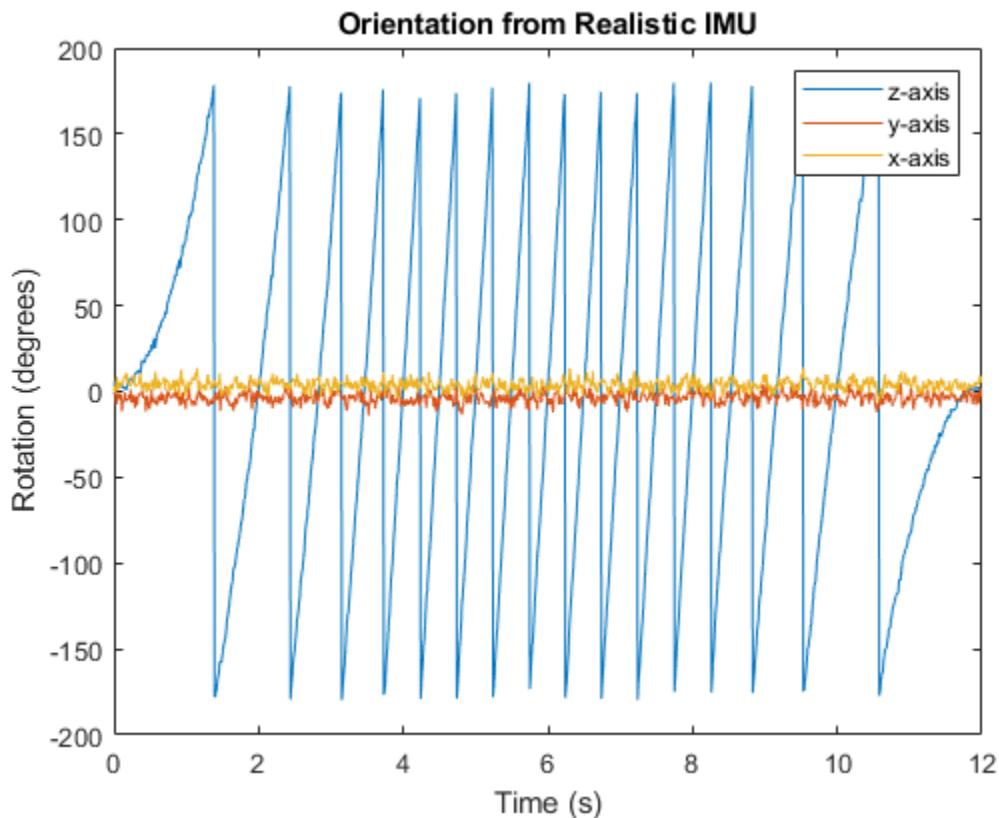
Modify parameters of the IMU System object to approximate realistic IMU sensor data. Reset the IMU and then call it with the same ground-truth acceleration, angular velocity, and orientation. Use ecompass to fuse the IMU data and plot the results.

```
IMU.Accelerometer = accelparams( ...
    'MeasurementRange',20, ...
    'Resolution',0.0006, ...
    'ConstantBias',0.5, ...
    'AxesMisalignment',2, ...
    'NoiseDensity',0.004, ...
    'BiasInstability',0.5);
IMU.Magnetometer = magparams( ...
    'MeasurementRange',200, ...
    'Resolution',0.01);
reset(IMU)

[accelerometerData,magnetometerData] = IMU(accelerationNED,angularVelocityNED,orientationNED);

orientation = ecompass(accelerometerData,magnetometerData);
orientationEuler = eulerd(orientation,'ZYX','frame');
```

```
figure(2)
plot(timeVector,orientationEuler)
legend('z-axis','y-axis','x-axis')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation from Realistic IMU')
```



## Input Arguments

**accelerometerReading — Accelerometer readings in sensor body coordinate system (m/s<sup>2</sup>)**  
N-by-3 matrix

Accelerometer readings in sensor body coordinate system in m/s<sup>2</sup>, specified as an N-by-3 matrix. The columns of the matrix correspond to the x-, y-, and z-axes of the sensor body. The rows in the matrix, N, correspond to individual samples. The accelerometer readings are normalized before use in the function.

Data Types: single | double

**magnetometerReading — Magnetometer readings in sensor body coordinate system (μT)**  
N-by-3 matrix

Magnetometer readings in sensor body coordinate system in μT, specified as an N-by-3 matrix. The columns of the matrix correspond to the x-, y-, and z-axes of the sensor body. The rows in the matrix, N, correspond to individual samples. The magnetometer readings are normalized before use in the function.

Data Types: single | double

#### **orientationFormat — Format used to describe orientation**

'quaternion' (default) | 'rotmat'

Format used to describe orientation, specified as 'quaternion' or 'rotmat'.

Data Types: char | string

## **Output Arguments**

#### **orientation — Orientation that rotates quantities from global coordinate system to sensor body coordinate system**

N-by-1 vector of quaternions (default) | 3-by-3-by-N array

Orientation that can rotate quantities from a global coordinate system to a body coordinate system, returned as a vector of quaternions or an array. The size and type of the **orientation** depends on the format used to describe orientation:

- 'quaternion' -- N-by-1 vector of quaternions with the same underlying data type as the input
- 'rotmat' -- 3-by-3-by-N array the same data type as the input

Data Types: quaternion | single | double

## **Algorithms**

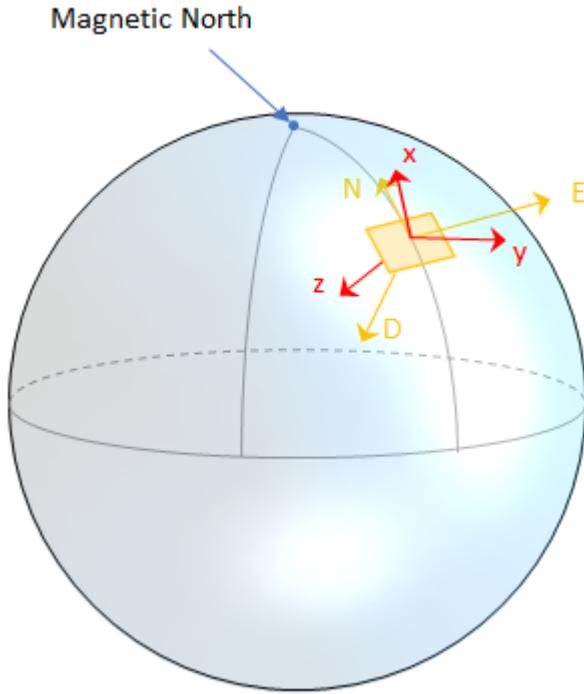
The **ecompass** function returns a quaternion or rotation matrix that can rotate quantities from a parent (NED for example) frame to a child (sensor) frame. For both output orientation formats, the rotation operator is determined by computing the rotation matrix.

The rotation matrix is first calculated with an intermediary:

$$R = \begin{bmatrix} (a \times m) \times a & a \times m & a \end{bmatrix}$$

and then normalized column-wise. *a* and *m* are the **accelerometerReading** input and the **magnetometerReading** input, respectively.

To understand the rotation matrix calculation, consider an arbitrary point on the Earth and its corresponding local NED frame. Assume a sensor body frame, [x,y,z], with the same origin.



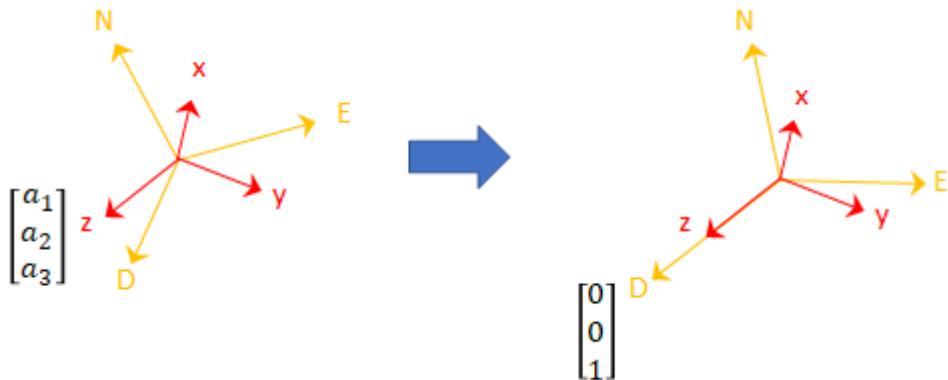
Recall that orientation of a sensor body is defined as the rotation operator (rotation matrix or quaternion) required to rotate a quantity from a parent (NED) frame to a child (sensor body) frame:

$$\begin{bmatrix} R \\ \left[ p_{\text{parent}} \right] \end{bmatrix} = \left[ p_{\text{child}} \right]$$

where

- $R$  is a 3-by-3 rotation matrix, which can be interpreted as the orientation of the child frame.
- $p_{\text{parent}}$  is a 3-by-1 vector in the parent frame.
- $p_{\text{child}}$  is a 3-by-1 vector in the child frame.

For a stable sensor body, an accelerometer returns the acceleration due to gravity. If the sensor body is perfectly aligned with the NED coordinate system, all acceleration due to gravity is along the  $z$ -axis, and the accelerometer reads  $[0 \ 0 \ 1]$ . Consider the rotation matrix required to rotate a quantity from the NED coordinate system to a quantity indicated by the accelerometer.

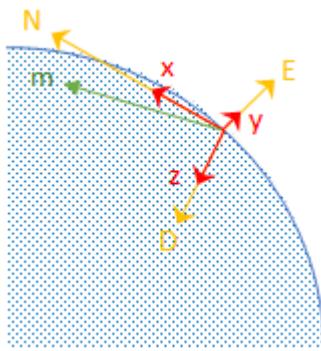


$$\begin{bmatrix} r_{11} & r_{21} & r_{31} \\ r_{12} & r_{22} & r_{32} \\ r_{13} & r_{23} & r_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The third column of the rotation matrix corresponds to the accelerometer reading:

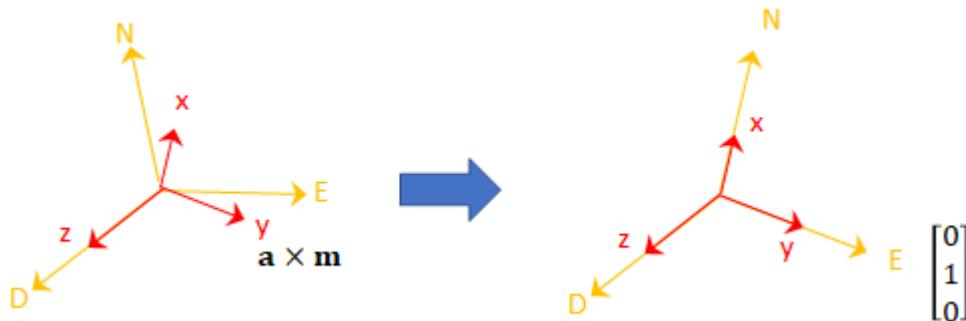
$$\begin{bmatrix} r_{31} \\ r_{32} \\ r_{33} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

A magnetometer reading points toward magnetic north and is in the *N-D* plane. Again, consider a sensor body frame aligned with the NED coordinate system.



By definition, the *E*-axis is perpendicular to the *N-D* plane, therefore  $N \times D = E$ , within some amplitude scaling. If the sensor body frame is aligned with NED, both the acceleration vector from the accelerometer and the magnetic field vector from the magnetometer lie in the *N-D* plane. Therefore  $m \times a = y$ , again with some amplitude scaling.

Consider the rotation matrix required to rotate NED to the child frame,  $[x \ y \ z]$ .



$$\begin{bmatrix} r_{11} & r_{21} & r_{31} \\ r_{12} & r_{22} & r_{32} \\ r_{13} & r_{23} & r_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix}$$

The second column of the rotation matrix corresponds to the cross product of the accelerometer reading and the magnetometer reading:

$$\begin{bmatrix} r_{21} \\ r_{22} \\ r_{23} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix}$$

By definition of a rotation matrix, column 1 is the cross product of columns 2 and 3:

$$\begin{aligned} \begin{bmatrix} r_{11} \\ r_{12} \\ r_{13} \end{bmatrix} &= \begin{bmatrix} r_{21} \\ r_{22} \\ r_{23} \end{bmatrix} \times \begin{bmatrix} r_{31} \\ r_{32} \\ r_{33} \end{bmatrix} \\ &= (a \times m) \times a \end{aligned}$$

Finally, the rotation matrix is normalized column-wise:

$$R_{ij} = \frac{R_{ij}}{\sqrt{\sum_{i=1}^3 R_{ij}^2}} , \forall j$$

---

**Note** The `ecompass` algorithm uses magnetic north, not true north, for the NED coordinate system.

---

## References

- [1] Open Source Sensor Fusion. <https://github.com/memsindustrygroup/Open-Source-Sensor-Fusion/tree/master/docs>

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`ahrsfilter` | `imufilter` | `quaternion`

### Topics

“Determine Orientation Using Inertial Sensors”

### Introduced in R2018b

# magcal

Magnetometer calibration coefficients

## Syntax

```
[A,b,expmfs] = magcal(D)
[A,b,expmfs] = magcal(D,fitkind)
```

## Description

`[A,b,expmfs] = magcal(D)` returns the coefficients needed to correct uncalibrated magnetometer data `D`.

To produce the calibrated magnetometer data `C`, use equation  $C = (D - b) * A$ . The calibrated data `C` lies on a sphere of radius `expmfs`.

`[A,b,expmfs] = magcal(D,fitkind)` constrains the matrix `A` to be the type specified by `fitkind`. Use this syntax when only the soft- or hard-iron effect needs to be corrected.

## Examples

### Correct Data Lying on Ellipsoid

Generate uncalibrated magnetometer data lying on an ellipsoid.

```
c = [-50; 20; 100]; % ellipsoid center
r = [30; 20; 50]; % semiaxis radii

[x,y,z] = ellipsoid(c(1),c(2),c(3),r(1),r(2),r(3),20);
D = [x(:),y(:),z(:)];
```

Correct the magnetometer data so that it lies on a sphere. The option for the calibration is set by default to 'auto'.

```
[A,b,expmfs] = magcal(D); % calibration coefficients
expmfs % Dipaly expected magnetic field strength in uT

expmfs = 31.0723

C = (D-b)*A; % calibrated data
```

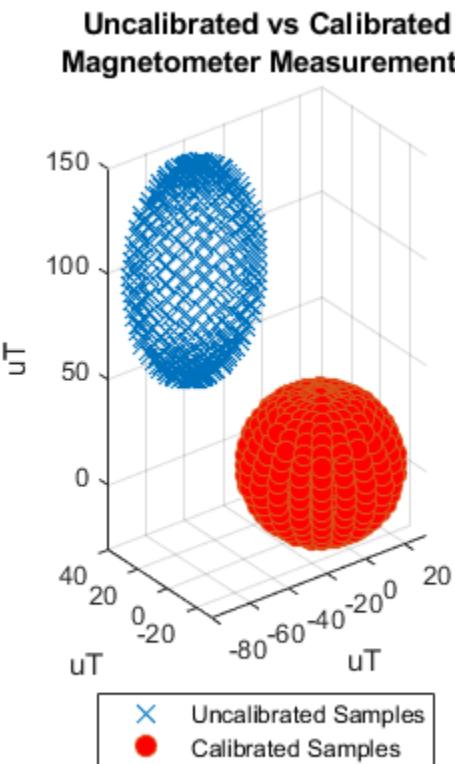
Visualize the uncalibrated and calibrated magnetometer data.

```
figure(1)
plot3(x(:),y(:),z(:), 'LineStyle', 'none', 'Marker', 'X', 'MarkerSize', 8)
hold on
grid(gca, 'on')
plot3(C(:,1),C(:,2),C(:,3), 'LineStyle', 'none', 'Marker', ...
'o', 'MarkerSize', 8, 'MarkerFaceColor', 'r')
axis equal
xlabel('uT')
```

```

ylabel('uT')
zlabel('uT')
legend('Uncalibrated Samples', 'Calibrated Samples','Location', 'southoutside')
title("Uncalibrated vs Calibrated" + newline + "Magnetometer Measurements")
hold off

```



## Input Arguments

### D — Raw magnetometer data

*N*-by-3 matrix (default)

Input matrix of raw magnetometer data, specified as a *N*-by-3 matrix. Each column of the matrix corresponds to the magnetometer measurements in the first, second and third axes, respectively. Each row of the matrix corresponds to a single three-axis measurement.

Data Types: `single` | `double`

### fitkind — Matrix output type

'auto' (default) | 'eye' | 'diag' | 'sym'

Matrix type for output A. The matrix type of A can be constrained to:

- 'eye' – identity matrix
- 'diag' – diagonal

- 'sym' - symmetric
- 'auto' - whichever of the previous options gives the best fit

## Output Arguments

### A — Correction matrix for soft-iron effect

3-by-3 matrix

Correction matrix for the soft-iron effect, returned as a 3-by-3 matrix.

### b — Correction vector for hard-iron effect

3-by-1 vector

Correction vector for the hard-iron effect, returned as a 3-by-1 array.

### expmfs — Expected magnetic field strength

scalar

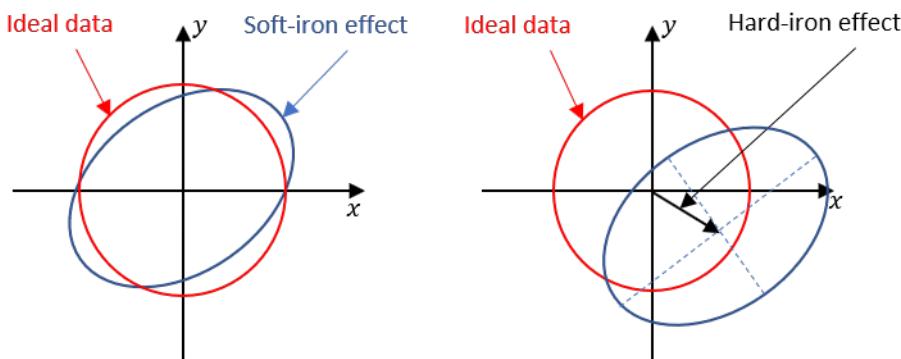
Expected magnetic field strength, returned as a scalar.

## More About

### Soft- and Hard-Iron Effects

Because a magnetometer usually rotates through a full range of 3-D rotation, the ideal measurements from a magnetometer should form a perfect sphere centered at the origin if the magnetic field is unperturbed. However, due to distorting magnetic fields from the sensor circuit board and the surrounding environment, the spherical magnetic measurements can be perturbed. In general, two effects exist.

- 1 The soft-iron effect is described as the distortion of the ellipsoid from a sphere and the tilt of the ellipsoid, as shown in the left figure. This effect is caused by disturbances that influence the magnetic field but may not generate their own magnetic field. For example, metals such as nickel and iron can cause this kind of distortion.
- 2 The hard-iron effect is described as the offset of the ellipsoid center from the origin. This effect is produced by materials that exhibit a constant, additive field to the earth's magnetic field. This constant additive offset is in addition to the soft-iron effect as shown in the figure on the right.



The underlying algorithm in `magcal` determines the best-fit ellipsoid to the raw sensor readings and attempts to "invert" the ellipsoid to produce a sphere. The goal is to generate a correction matrix A to

account for the soft-iron effect and a vector  $b$  to account for the hard-iron effect. The three output options, 'eye', 'diag' and 'sym' correspond to three parameter-solving algorithms, and the 'auto' option chooses among these three options to give the best fit.

## References

- [1] Ozyagcilar, T. "Calibrating an eCompass in the Presence of Hard and Soft-iron Interference." *Freescale Semiconductor Ltd.* 1992, pp. 1-17.

## See Also

### Classes

`magparams`

### Objects

`imuSensor`

## Introduced in R2019a

# monteCarloRun

Monte Carlo realization of tracking scenario

## Syntax

```
recordings = monteCarloRun(scenario, numRuns)
recordings = monteCarloRun(scenario, numRuns, Name, Value)
[recordings, rngs] = monteCarloRun(___)
```

## Description

`recordings = monteCarloRun(scenario, numRuns)` runs a tracking scenario multiple times and saves the running recording of every run. Each run, called a realization of the scenario, is with a different random seed.

`recordings = monteCarloRun(scenario, numRuns, Name, Value)` specifies options using one or more name-value pair arguments. Enclose each `Name` in quotes.

`[recordings, rngs] = monteCarloRun(___)` also returns the random number generator values at the beginning of each realization run.

## Examples

### Run Scenario Twice with Automatic Random Seeds

Load a prerecorded tracking scenario.

```
load ATCScenario.mat scenario
```

Execute two Monte Carlo runs and display the running time.

```
tic
recordings = monteCarloRun(scenario, 2);
disp("Time to run the scenarios: " + toc + " sec")
```

Time to run the scenarios: 563.2982 sec

Run the Monte Carlo simulations again using parallel computing.

```
tic
recordings = monteCarloRun(scenario, 2, 'UseParallel', true);
```

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 12).

```
disp("Time to run the scenarios in parallel: " + toc + " sec")
```

Time to run the scenarios in parallel: 469.5262 sec

## Input Arguments

### **scenario — Tracking scenario**

*M*-element array of `trackingScenario` objects | *M*-element cell array of `trackingScenario` objects

Tracking scenario, specified as an *M*-element array of `trackingScenario` objects or an *M*-element cell array of `trackingScenario` objects.

### **numRuns — Number of Monte Carlo runs**

positive integer

Numbers of Monte Carlo runs, specified as a positive integer.

## 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: `monteCarloRun(sc,3,'UseParallel',false)`

### **UseParallel — Enable parallel computing**

`false` (default) | `true`

Enable parallel computing, specified as `true` or `false`. Using parallel computing requires a Parallel Computing Toolbox™ license and an open parallel pool.

### **InitialSeeds — Initial random seeds**

integer in  $[0, 2^{32}-1]$  | *numRuns*-element array of integers in  $[0, 2^{32}-1]$

Initial random seeds for obtaining repeatable results, specified as an integer in  $[0, 2^{32}-1]$  or an array of integers in  $[0, 2^{32}-1]$ . If specified as an integer, an array of seed values is randomly generated using the integer as an initial seed. If unspecified, the function uses the current random number generator to randomly generate an array of initial seeds.

## Output Arguments

### **recordings — Monte Carlo recordings of tracking scenario**

*M*-by-*numRuns* array of `trackingScenarioRecording` objects

Monte Carlo recordings of a tracking scenario, returned as a *M*-by-*numRuns* array of `trackingScenarioRecording` objects.

### **rngs — Random number generator values**

*M*-by-*numRuns* array of structures

Random number generator values, returned as a *M*-by-*numRuns* array of structures. The fields of each structure are the same as the output of the `rng` function.

## See Also

`rng` | `trackingScenario` | `trackingScenarioRecording`

**Introduced in R2020a**

# partitionDetections

Partition detections based on Mahalanobis distance

## Syntax

```
partitions = partitionDetections(detections)
partitions = partitionDetections(detections,tLower,tUpper)
partitions = partitionDetections(detections,tLower,tUpper,'MaxNumPartitions',
maxNumber)
partitions = partitionDetections(detections,allThresholds)
```

## Description

Using multiple distance thresholds, the function separates detections into different detection cells based on their relative Mahalanobis distances and reports all the possible partitions. A partition of a set of detections is defined as a division of these detections into nonempty mutually exclusive detection cells. A detection cell is a group of detections whose distance to at least one other detection in the cell is less than the distance threshold. In other words, two detections belong to the same detection cell if their distance is less than the distance threshold.

`partitions = partitionDetections(detections)` returns possible `partitions` from `detections`, using distance partitioning algorithm. By default, the function considers all real value Mahalanobis distance thresholds between 0.5 and 6.25.

`partitions = partitionDetections(detections,tLower,tUpper)` allows you to specify the lower and upper bounds of the distance thresholds, `tLower` and `tUpper`.

`partitions = partitionDetections(detections,tLower,tUpper,'MaxNumPartitions',
maxNumber)` allows you to specify the maximum number of allowed partitions, `maxNumber`, in addition to the lower and upper bounds of the distance thresholds, `tLower` and `tUpper`.

`partitions = partitionDetections(detections,allThresholds)` allows you to specify the exact thresholds considered for partition.

## Examples

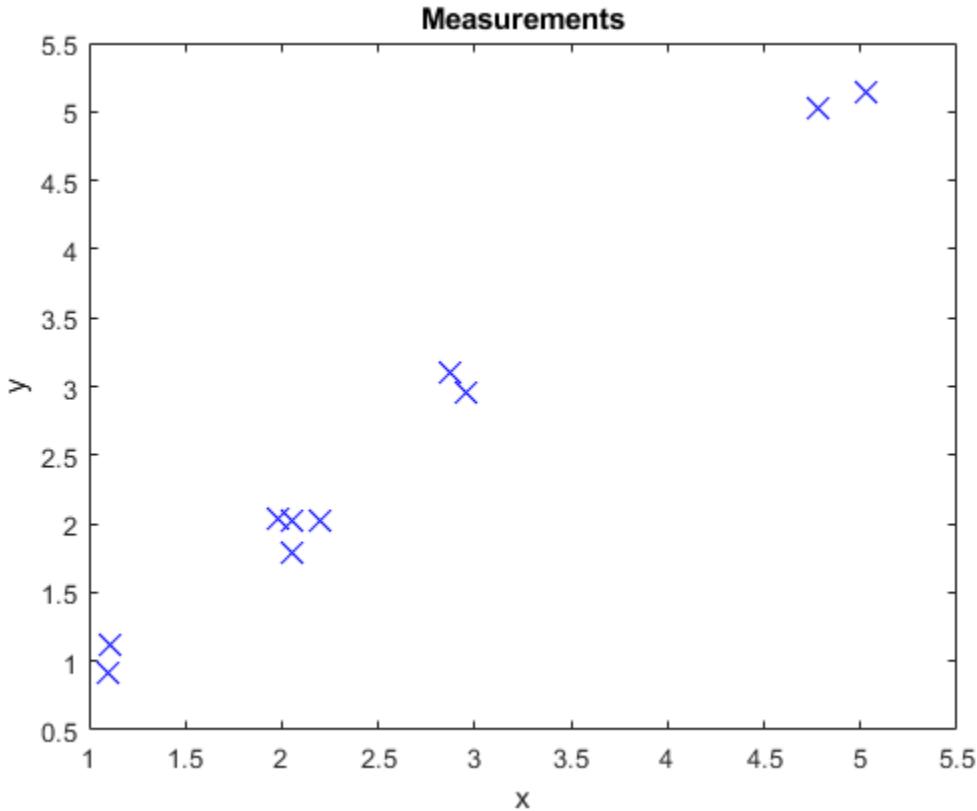
### Generate Partition from Object Detection

Generate 2-D detections using `objectDetection`.

```
rng(2018); % For reproducible results
detections = cell(10,1);
for i = 1:numel(detections)
    id = randi([1 5]);
    detections{i} = objectDetection(0,[id;id] + 0.1*randn(2,1));
    detections{i}.MeasurementNoise = 0.01*eye(2);
end
```

Extract and display generated position measurements.

```
d = [detections{:}];  
measurements = [d.Measurement];  
  
figure()  
plot(measurements(1,:),measurements(2,:),'x','MarkerSize',10,'MarkerEdgeColor','b')  
title('Measurements')  
xlabel('x')  
ylabel('y')
```

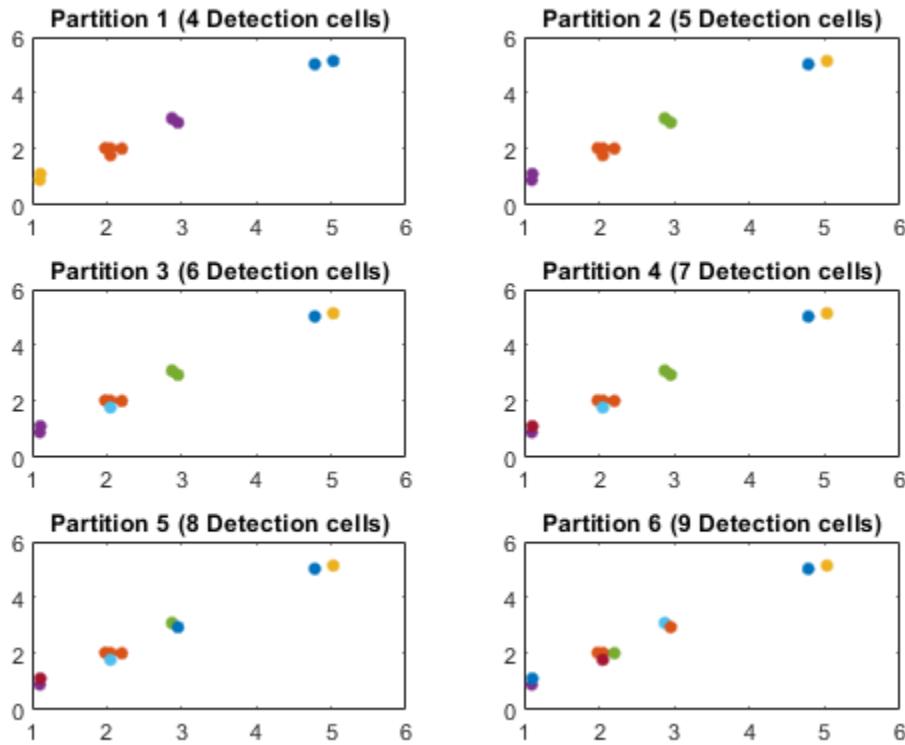


Generate partitions from the detections and count the number of partitions.

```
partitions = partitionDetections(detections);  
numPartitions = size(partitions,2);
```

Visualize the partitions. Each color represents a detection cell.

```
figure()  
for i = 1:numPartitions  
    numCells = max(partitions(:,i));  
    subplot(3,2,i);  
    for k = 1:numCells  
        ids = partitions(:,i) == k;  
        plot(measurements(1,ids),measurements(2,ids),'.','MarkerSize',15);  
        hold on;  
    end  
    title(['Partition ',num2str(i), ' (',num2str(k), ' Detection cells)']);  
end
```



## Input Arguments

### **detections — Object detections**

*N*-element cell array

Object detections, specified as an *N*-element cell array of `objectDetection` objects, where *N* is the number of detections. You can create `detections` directly, or you can obtain `detections` from the outputs of sensor objects, such as `radarSensor`, `monostaticRadarSensor`, `irSensor`, and `sonarSensor`.

Data Types: `cell`

### **tLower — Lower bound of distance thresholds**

scalar

Lower bound of distance thresholds, specified as a scalar. This argument sets the lower bound of the Mahalanobis distance thresholds considered for partition.

Example: `0.05`

Data Types: `double`

### **tUpper — Upper bound of distance thresholds**

scalar

Upper bound of distance thresholds, specified as a scalar. This argument sets the upper bound of the Mahalanobis distance thresholds considered for partition.

Example: 0.98

Data Types: double

#### **maxNumber — Maximum number of allowed partitions**

positive integer

Maximum number of allowed partitions, specified as a positive integer.

Example: 20

Data Types: double

#### **allThresholds — All thresholds for partitions**

M-element vector

All thresholds for partitions, specified as an  $M$  element vector. The function calculates partitions based on each threshold value provided in `allThresholds`. Note that multiple thresholds can result in the same partition, and the function output `partitions`, given as an  $N$ -by- $Q$  matrix with  $Q \leq M$ , only contains unique partitions.

Example: [0.1;0.2;0.35;0.4]

Data Types: double

## **Output Arguments**

#### **partitions — Partitions of detections**

$N$ -by- $Q$  matrix

Partitions of detections, specified as an  $N$ -by- $Q$  matrix.  $N$  is the number of detections, and  $Q$  is the number of partitions. Each column of the matrix represents a valid partition. In each column, the value of the  $i$ th element represents the identity number of the detection cell that the  $i$ th detection belongs to. For example, given a partition matrix  $P$ , if  $P(i,j) = k$ , then in partition  $j$ , detection  $i$  belongs to detection cell  $k$ .

## **References**

- [1] Granstrom, K., C. Lundquist, and O. Orguner. " Extended target tracking using a Gaussian-mixture PHD filter." *IEEE Transactions on Aerospace and Electronic Systems*. Vol. 48, Number 4, 2012, pp. 3268-3286.

## **Extended Capabilities**

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

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

Usage notes and limitations:

- The code generation configuration must allow recursion.

**See Also**

[objectDetection](#) | [trackerPHD](#)

**Introduced in R2019a**

# 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  $m_1$ -by-...-by- $m_N$  array of random unit quaternions, where  $m_1$ , ...,  $m_N$  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  $m_1$ -by-...-by- $m_N$  array of random unit quaternions, where  $m_1$ , ...,  $m_N$  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)
```

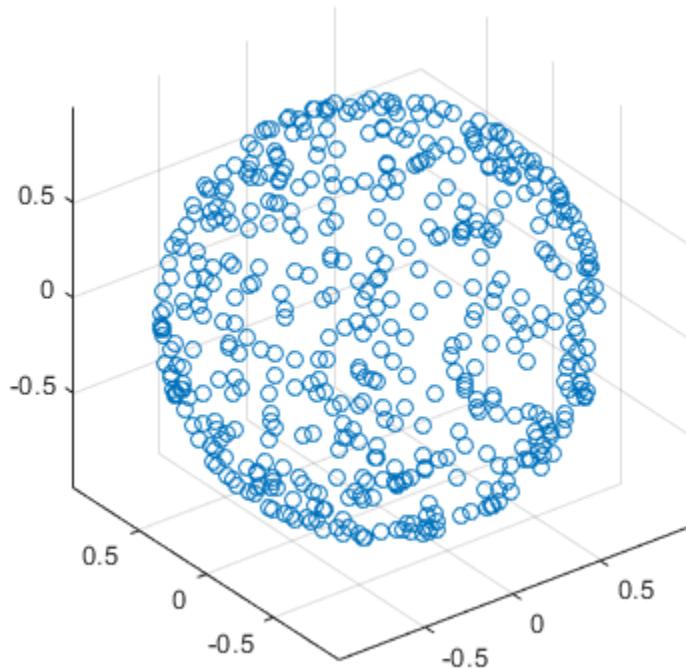
```
r=3x3 quaternion array
 0.17446 + 0.59506i - 0.73295j + 0.27976k      0.69704 - 0.060589i + 0.68679j - 0.19695
 0.21908 - 0.89875i - 0.298j + 0.23548k      -0.049744 + 0.59691i + 0.56459j + 0.56780
 0.6375 + 0.49338i - 0.24049j + 0.54068k      0.2979 - 0.53568i + 0.31819j + 0.72323
```

### Create Uniform Distribution of Random Rotations

Create a vector of 500 random quaternions. Use `rotatepoint` on page 1-138 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

 **$m_1, \dots, m_N$  — 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

 **$[m_1, \dots, m_N]$  — 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`

### Topics

"Rotations, Orientation, and Quaternions"

### Introduced in R2019a

# 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

### **Topics**

"Rotations, Orientation, and Quaternions"

**Introduced in R2020a**

# 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 = 1x3
```

```
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

[euler](#) | [eulerd](#) | [rotvec](#)

### Objects

[quaternion](#)

### Topics

"Rotations, Orientation, and Quaternions"

## Introduced in R2018b

# 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' | 'ZXY' | 'ZXZ' | 'YXZ' | 'YXY' | 'YZX' | 'XYZ' | 'XYX' | 'XZY' | 'XZX'

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.

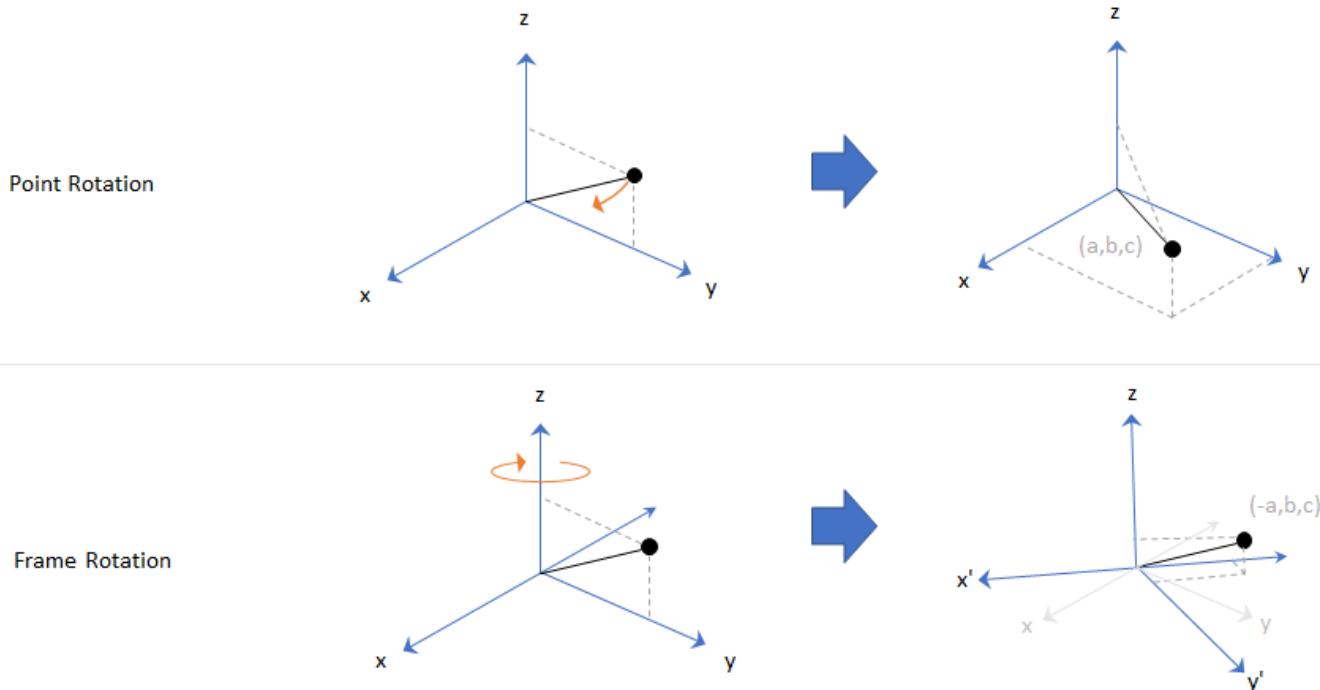
Data Types: `char` | `string`

**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](#)

#### Topics

“Rotations, Orientation, and Quaternions”

### Introduced in R2018b

# 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 does not 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];
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 = 1x3
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');

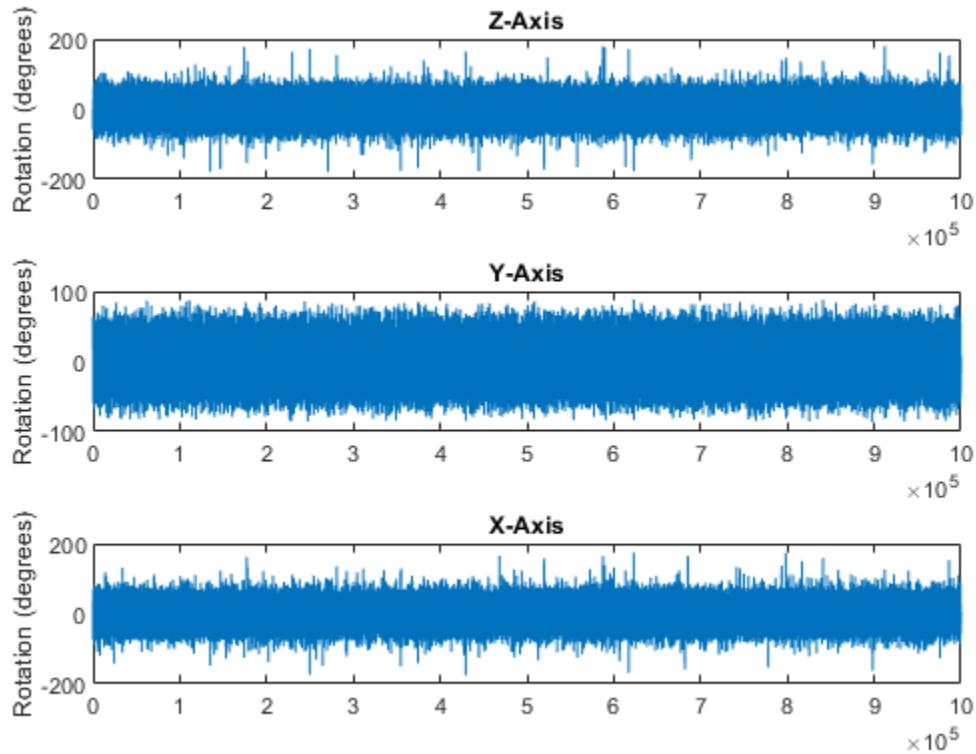
figure(1)

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

```



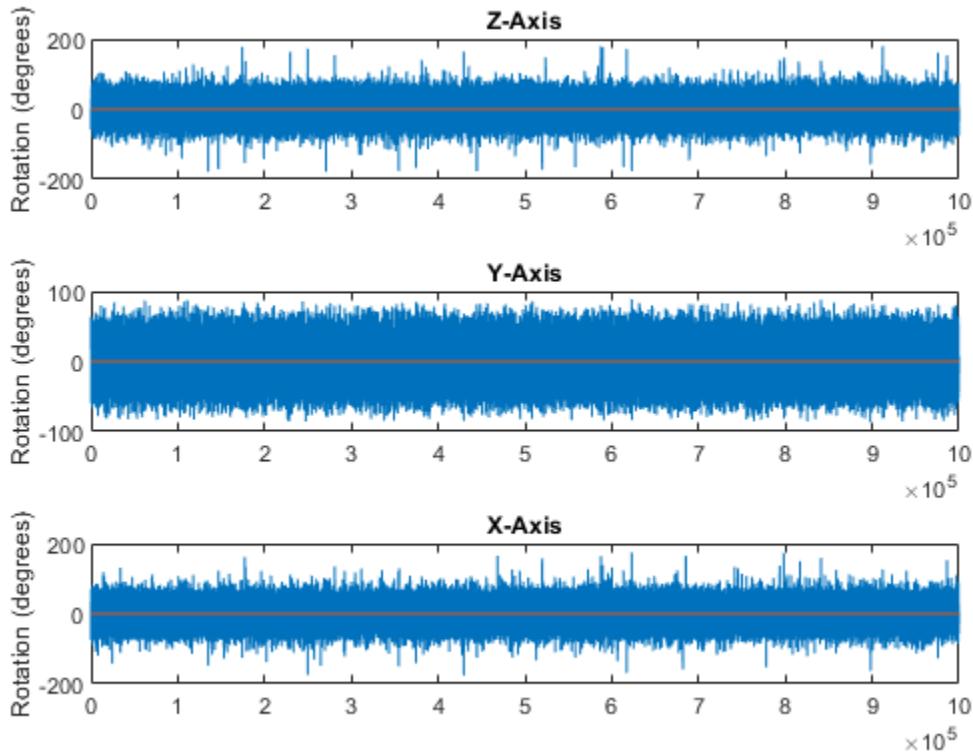
Use `meanrot` to determine the average quaternion given the vector of quaternions. Convert to Euler angles and plot the results.

```
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');
```

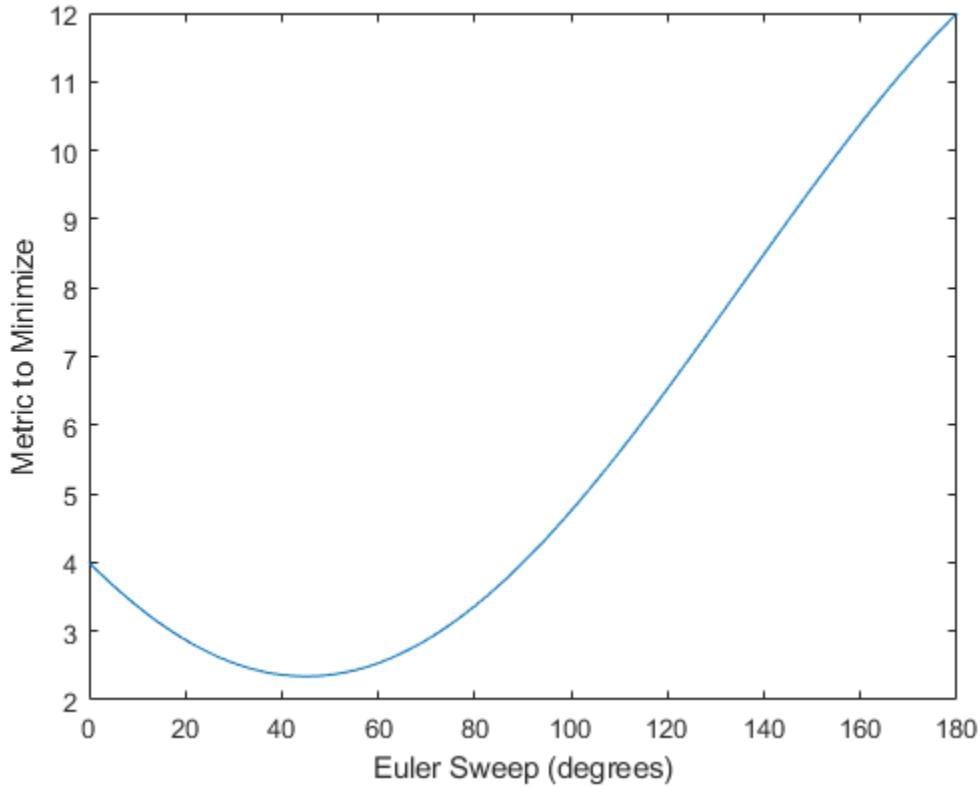
Convert `q0`, `q90`, and `qSweep` to rotation matrices. In a loop, calculate the metric to minimize for each member of the quaternion sweep. Plot the results and return the value of the Euler sweep that corresponds to the minimum of the metric.

```
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
```

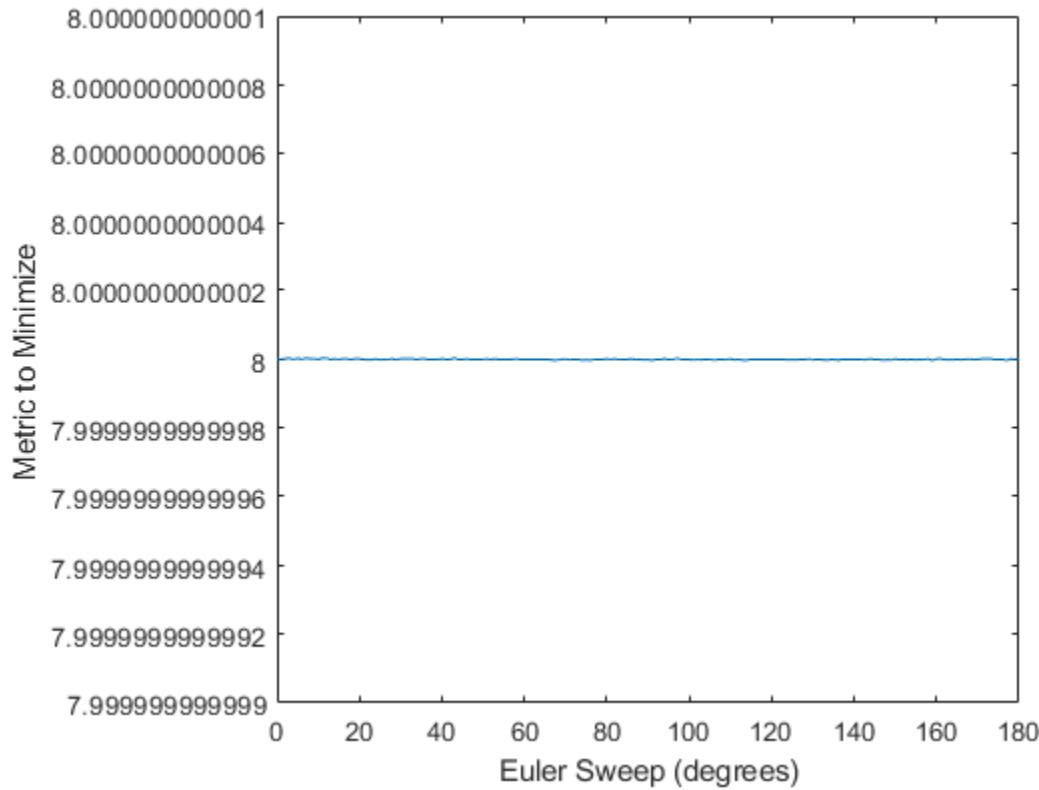
## Limitations

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 = 1x3
    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 `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`

### Topics

"Rotations, Orientation, and Quaternions"

## Introduced in R2018b

# slerp

Spherical linear interpolation

## Syntax

```
qθ = slerp(q1,q2,T)
```

## Description

$qθ = \text{slerp}(q1, q2, T)$  spherically interpolates between  $q1$  and  $q2$  by the interpolation coefficient  $T$ .

## 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))
```

$$abc = 10 \times 3$$

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.

## 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.

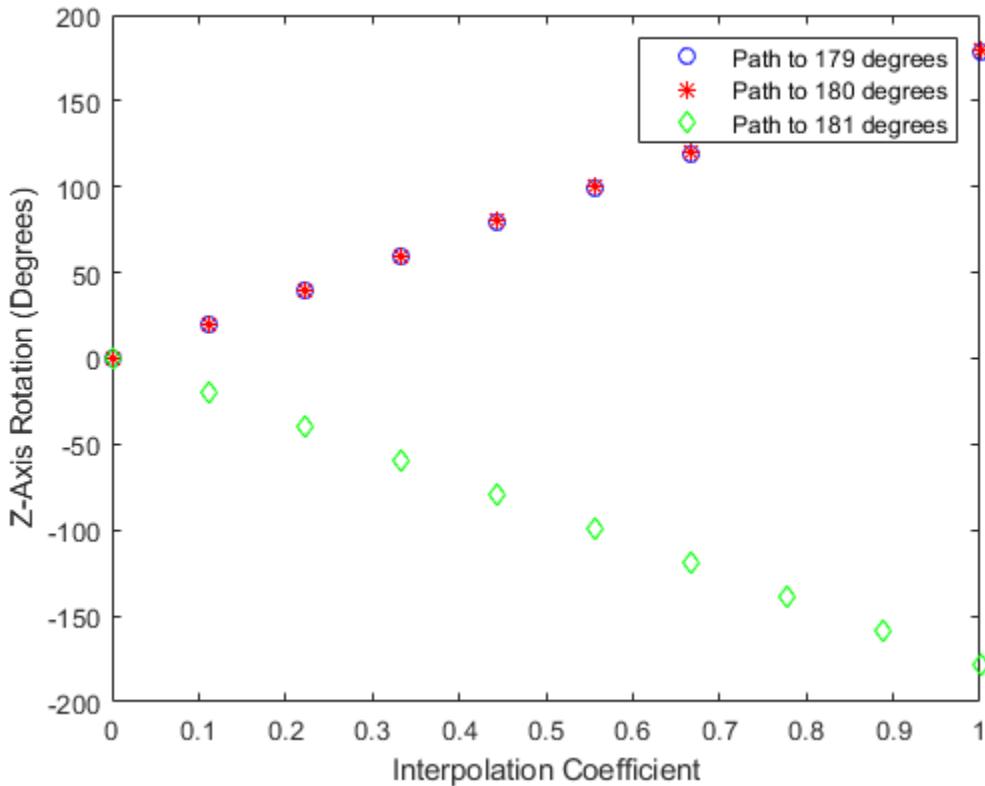
```
T = linspace(0,1,10);
```

```
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.

## 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**

#### **$q_0$ — 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](#)

### Topics

“Lowpass Filter Orientation Using Quaternion SLERP”

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

# 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](#)

### Topics

[“Rotations, Orientation, and Quaternions”](#)

### Introduced in R2018b

# 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  $N = \text{numel}(\text{quat})$ .

Data Types: single | double

## Extended Capabilities

**C/C++ Code Generation**

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

## See Also

**Functions**

[classUnderlying](#) | [parts](#)

**Objects**

[quaternion](#)

**Topics**

["Rotations, Orientation, and Quaternions"](#)

**Introduced in R2018b**

# 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`

### **Topics**

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

# 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=4×1 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=1×4 quaternion array
  0.53767 - 0.31877i - 3.5784j - 0.7254k           1.8339 + 1.3077i - 2.7694j + 0.063055k
```

### 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=2×2 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=2×2 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`**Topics**

“Rotations, Orientation, and Quaternions”

**Introduced in R2018b**

# transformMotion

Compute motion quantities between two relatively fixed frames

## Syntax

```
[posS,orientS,velS,accS,angvelS] = transformMotion(posSFromP,orientSFromP,
posP)
[____] = transformMotion(posSFromP,orientSFromP,posP,orientP)
[____] = transformMotion(posSFromP,orientSFromP,posP,orientP,velP)
[____] = transformMotion(posSFromP,orientSFromP,posP,orientP,velP,accP)
[____] = transformMotion(posSFromP,orientSFromP,posP,orientP,velP,accP,
angvelP)
```

## Description

`[posS,orientS,velS,accS,angvelS] = transformMotion(posSFromP,orientSFromP, posP)` computes motion quantities of the sensor frame relative to the navigation frame (`posS`, `orientS`, `velS`, `accS`, and `angvelS`) using the position of sensor frame relative to the platform frame, `posSFromP`, the orientation of the sensor frame relative to the platform frame, `orientSFromP`, and the position of the platform frame relative to the navigation frame, `posP`. Note that the position and orientation between the sensor frame and the platform frame are assumed to be fixed. Also, the unspecified quantities between the navigation frame and the platform frame (such as orientation, velocity, and acceleration) are assumed to be zero.

`[____] = transformMotion(posSFromP,orientSFromP,posP,orientP)` additionally specifies the orientation of the platform frame relative to the navigation frame, `orientP`. The output arguments are the same as those of the previous syntax.

`[____] = transformMotion(posSFromP,orientSFromP,posP,orientP,velP)` additionally specifies the velocity of the platform frame relative to the navigation frame, `velP`. The output arguments are the same as those of the previous syntax.

`[____] = transformMotion(posSFromP,orientSFromP,posP,orientP,velP,accP)` additionally specifies the acceleration of the platform frame relative to the navigation frame, `accP`. The output arguments are the same as those of the previous syntax.

`[____] = transformMotion(posSFromP,orientSFromP,posP,orientP,velP,accP, angvelP)` additionally specifies the angular velocity of the platform frame relative to the navigation frame, `angvelP`. The output arguments are the same as those of the previous syntax.

## Examples

### Transform State to Sensor Frame

Define the pose, velocity, and acceleration of the platform frame relative to the navigation frame.

```
posPlat = [20 -1 0];
orientPlat = quaternion(1, 0, 0, 0);
velPlat = [0 0 0];
```

```
accPlat = [0 0 0];
angvelPlat = [0 0 1];
```

Define the position and orientation offset of IMU sensor frame relative to the platform frame.

```
posPlat2IMU = [1 2 3];
orientPlat2IMU = quaternion([45 0 0], 'eulerd', 'ZYX', 'frame');
```

Calculate the motion quantities of the sensor frame relative to the navigation frame and print the results.

```
[posIMU, orientIMU, velIMU, accIMU, angvelIMU] ...
= transformMotion(posPlat2IMU, orientPlat2IMU, ...
posPlat, orientPlat, velPlat, accPlat, angvelPlat);

fprintf('IMU position is:\n');

IMU position is:

fprintf('%.2f %.2f %.2f\n', posIMU);

21.00 1.00 3.00

orientIMU

orientIMU = quaternion
    0.92388 +      0i +      0j + 0.38268k

velIMU

velIMU = 1x3

    -2      1      0

accPlat

accPlat = 1x3

    0      0      0
```

## Input Arguments

### **posSFromP — Position of sensor frame relative to platform frame**

1-by-3 vector of real scalars

Position of the sensor frame relative to the platform frame, specified as a 1-by-3 vector of real scalars.

Example: [1 2 3]

### **orientSFromP — Orientation of sensor frame relative to platform frame**

quaternion | 3-by-3 rotation matrix

Orientation of the sensor frame relative to the platform frame, specified as a quaternion or a 3-by-3 rotation matrix.

Example: `quaternion(1,0,0,0)`

**posP — Position of platform frame relative to navigation frame**

*N*-by-3 matrix of real scalars

Position of platform frame relative to navigation frame, specified as an *N*-by-3 matrix of real scalars. *N* is the number of position quantities.

Example: `[1 2 3]`

**orientP — Orientation of platform frame relative to navigation frame**

*N*-by-1 array of quaternion | 3-by-3-by-*N* array of scalars

Orientation of platform frame relative to navigation frame, specified as an *N*-by-1 array of quaternions, or a 3-by-3-by-*N* array of scalars. Each 3-by-3 matrix must be a rotation matrix. *N* is the number of orientation quantities.

Example: `quaternion(1,0,0,0)`

**velP — Velocity of platform frame relative to navigation frame**

*N*-by-3 matrix of real scalars

Velocity of platform frame relative to navigation frame, specified as an *N*-by-3 matrix of real scalars. *N* is the number of velocity quantities.

Example: `[ 4 8 6 ]`

**accP — Acceleration of platform frame relative to navigation frame**

*N*-by-3 matrix of real scalars

Acceleration of platform frame relative to navigation frame, specified as an *N*-by-3 matrix of real scalars. *N* is the number of acceleration quantities.

Example: `[ 4 8 6 ]`

**angvelP — Angular velocity of platform frame relative to navigation frame**

*N*-by-3 matrix of real scalars

Angular velocity of platform frame relative to navigation frame, specified as an *N*-by-3 matrix of real scalars. *N* is the number of angular velocity quantities.

Example: `[ 4 2 3 ]`

## Output Arguments

**posS — Position of sensor frame relative to navigation frame**

*N*-by-3 matrix of real scalars

Position of sensor frame relative to navigation frame, returned as an *N*-by-3 matrix of real scalars. *N* is the number of position quantities specified by the posP input.

**orientS — Orientation of sensor frame relative to navigation frame**

*N*-by-1 array of quaternion | 3-by-3-by-*N* array of scalars

Orientation of sensor frame relative to navigation frame, returned as an *N*-by-1 array of quaternions, or a 3-by-3-by-*N* array of scalars. *N* is the number of orientation quantities specified by the orientP input. The returned orientation quantity type is same with the orientP input.

**velS — Velocity of sensor frame relative to navigation frame**

*N*-by-3 matrix of real scalars

Velocity of sensor frame relative to navigation frame, returned as an *N*-by-3 matrix of real scalars. *N* is the number of position quantities specified by the `velP` input.

**accS — Acceleration of sensor frame relative to navigation frame**

*N*-by-3 matrix of real scalars

Acceleration of sensor frame relative to navigation frame, returned as an *N*-by-3 matrix of real scalars. *N* is the number of position quantities specified by the `accP` input.

**angvelS — Angular velocity of sensor frame relative to navigation frame**

*N*-by-3 matrix of real scalars

Angular velocity of sensor frame relative to navigation frame, returned as an *N*-by-3 matrix of real scalars. *N* is the number of position quantities specified by the `angvelP` input.

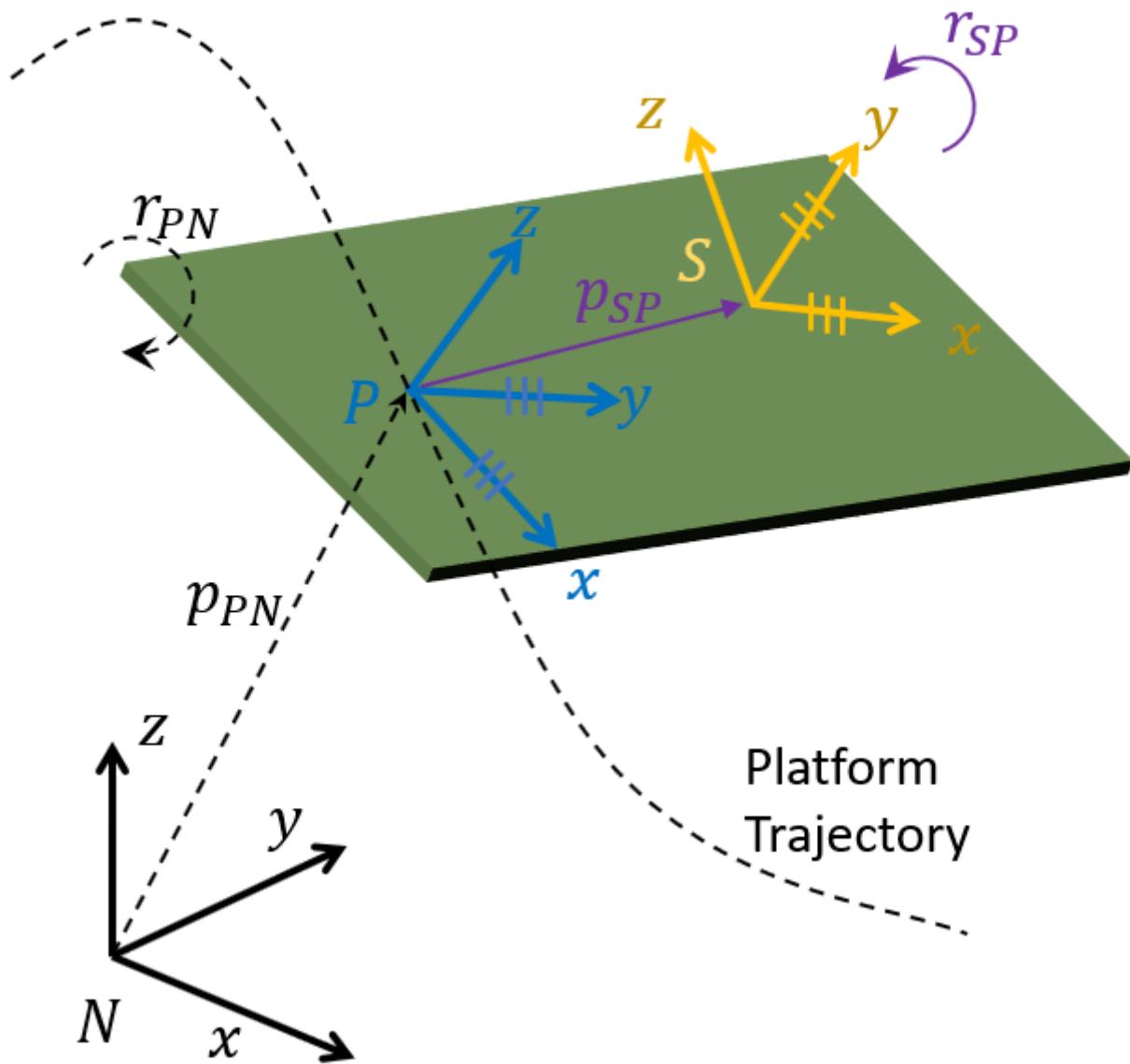
## More About

### Motion Quantities Used in `transformMotion`

The `transformMotion` function calculates the motion quantities of the sensor frame (*S*), which is fixed on a rigid platform, relative to the navigation frame (*N*) using the mounting information of the sensor on the platform and the motion information of the platform frame (*P*).

As shown in the figure, the position and orientation of the platform frame and the sensor frame are fixed on the platform. The position of the sensor frame relative to the platform frame is  $p_{SP}$ , and the orientation of the sensor frame relative to the platform frame is  $r_{SP}$ . Since the two frames are both fixed,  $p_{SP}$  and  $r_{SP}$  are constant.

To compute the motion quantities of the sensor frame relative to the navigation frame, the quantities describing the motion of the platform frame relative to the navigation frame are required. These quantities include: the platform position ( $p_{PN}$ ), orientation ( $r_{PN}$ ), velocity, acceleration, angular velocity, and angular acceleration relative to the navigation frame. You can specify these quantities through the function input arguments except the angular acceleration, which is always assumed to be zero in the function. The unspecified quantities are also assumed to be zero.



**See Also**

Introduced in R2020a

## 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=5×1 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 = 5×1
    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,'eulerrd','zyx','frame');
qVector2 = quaternion(angles2,'eulerrd','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],'eulerrd','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  $[A\text{dim}1, \dots, A\text{dim}N] = \text{size}(\text{quatA})$  and  $[B\text{dim}1, \dots, B\text{dim}N] = \text{size}(\text{quatB})$ , then for  $i = 1:N$ , either  $A\text{dim}i == B\text{dim}i$  or  $A\text{dim}i == 1$  or  $B\text{dim}i == 1$ .

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  $\text{size}(\text{quatB})$ .

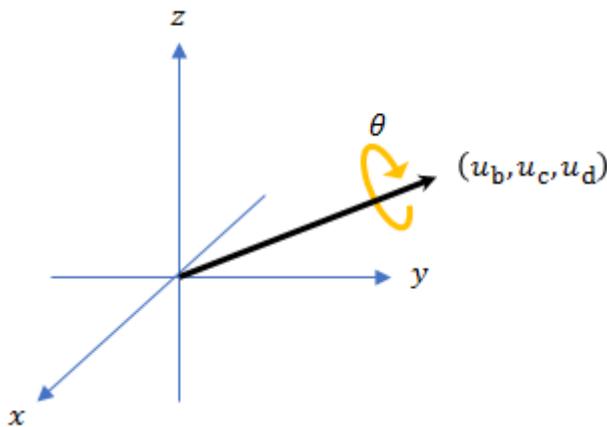
Data Types: single | double

## 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  $\theta_q$ :

$$q = \cos(\theta_q/2) + \sin(\theta_q/2)(u_b i + u_c j + u_d k).$$



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

[conj](#) | [parts](#)

#### Objects

[quaternion](#)

#### Topics

“Rotations, Orientation, and Quaternions”

### Introduced in R2018b

# 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]);
eulerAnglesRandians = euler(quat, 'ZYX', 'frame')
eulerAnglesRandians = 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' | 'ZXY' | 'ZXZ' | 'YXZ' | 'YXY' | 'YZX' | 'XYZ' | 'XYX' | 'XZY' | 'XZX'

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.

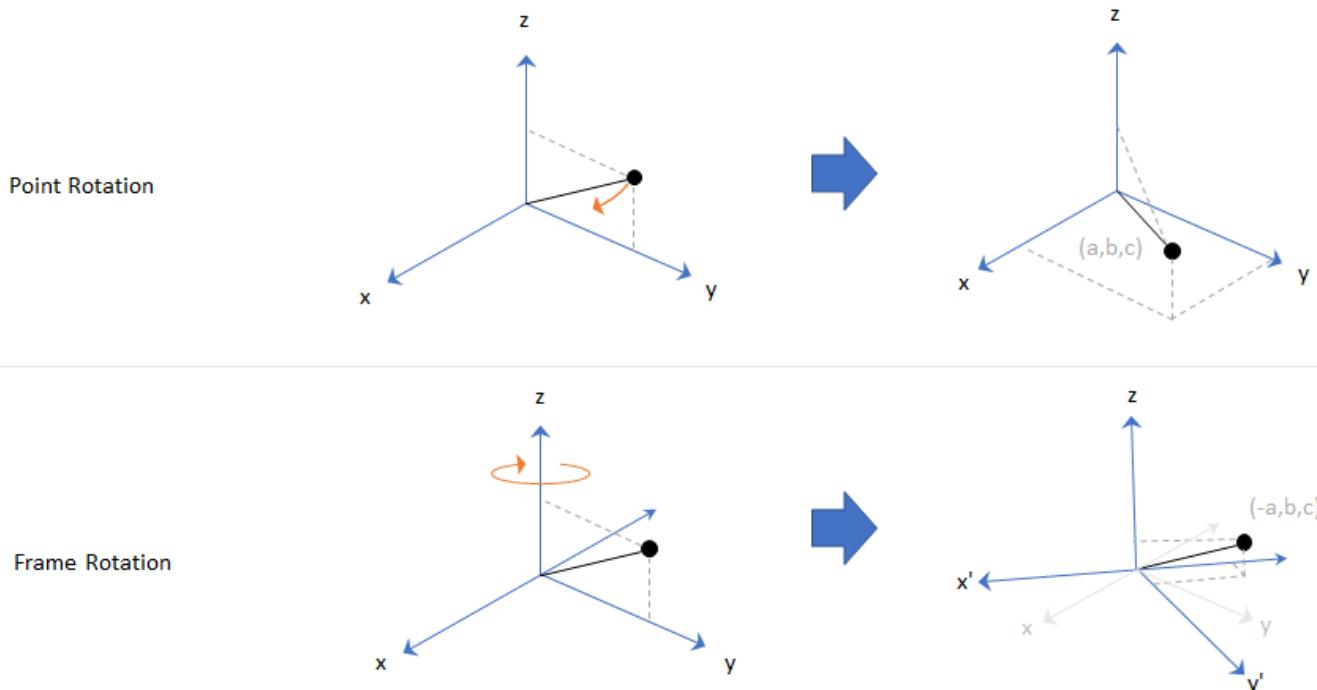
Data Types: `char` | `string`

**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](#)

### Topics

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

# 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=4×1 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=4×1 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

`log | power, .^`

### Objects

`quaternion`

### Topics

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

# **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=2×1 quaternion array
 1 + 2i + 3j + 4k
 5 + 6i + 7j + 8k
```

```
B = 2;
C = A.\B
```

```
C=2×1 quaternion array
 0.066667 - 0.13333i - 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=2×2 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=2×2 quaternion array
  16 + 2i + 3j + 13k      9 + 7i + 6j + 12k
  5 + 11i + 10j + 8k      4 + 14i + 15j + 1k

C = A.\B

C=2×2 quaternion array
  2.7 -    1.9i -    0.9j -    1.7k      1.5159 -  0.37302i -  0.15079j -  0.0238
  2.2778 +  0.46296i -  0.57407j + 0.092593k      1.2471 +  0.91379i -  0.33908j -  0.1092
```

## 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  $p$ ,  $A./p = A.\lvert p$ .

---

### 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

`conj` | `norm` | `rdivide`, `./` | `times`, `.*`

### Objects

`quaternion`

### Topics

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

## 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=3×1 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=3×1 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`

### Topics

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

## 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 = 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

`mtimes`, `*` | `times`, `.``*` | `uminus`, `-`

### Objects

`quaternion`

### Topics

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

## 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=4×1 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=4×1 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

$$q = a_q + b_qi + c_qj + d_qk,$$

the product of  $q$  and a real scalar  $\beta$  is

$$\beta q = \beta a_q + \beta b_qi + \beta c_qj + \beta d_qk$$

### 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

j	j	-k	-1	i
k	k	j	-i	-1

When reading the table, the rows are read first, for example:  $ij = k$  and  $ji = -k$ .

Given two quaternions,  $q = a_q + b_qi + c_qj + d_qk$ , and  $p = a_p + b_pi + c_pj + d_pk$ , the multiplication can be expanded as:

$$\begin{aligned} z = pq &= (a_p + b_pi + c_pj + d_pk)(a_q + b_qi + c_qj + d_qk) \\ &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\ &\quad + b_p a_q i + b_p b_q i^2 + b_p c_q i j + b_p d_q i k \\ &\quad + c_p a_q j + c_p b_q j i + c_p c_q j^2 + c_p d_q j k \\ &\quad + d_p a_q k + d_p b_q k i + d_p c_q k j + d_p d_q k^2 \end{aligned}$$

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`

### Topics

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

# 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

`conj` | `normalize` | `parts`

### Objects

`quaternion`

### Topics

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

# 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

`conj` | `norm` | `times`, `.*`

### Objects

`quaternion`

### Topics

“Rotations, Orientation, and Quaternions”

**Introduced in R2018b**

## ones

Create quaternion array with real parts set to one and imaginary parts set to zero

### Syntax

```
quatOnes = ones('quaternion')
quatOnes = ones(n,'quaternion')
quatOnes = ones(sz,'quaternion')
quatOnes = ones(sz1,...,szN,'quaternion')

quatOnes = ones(___, 'like', prototype, 'quaternion')
```

### Description

`quatOnes = 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  $Q = 1 + 0i + 0j + 0k$ .

`quatOnes = ones(n,'quaternion')` returns an n-by-n quaternion matrix with the real parts set to 1 and the imaginary parts set to 0.

`quatOnes = 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.

`quatOnes = ones(sz1,...,szN,'quaternion')` returns a sz1-by-...-by-szN array of ones where `sz1,...,szN` indicates the size of each dimension.

`quatOnes = 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.

```
quatOnes = ones('quaternion')

quatOnes = quaternion
1 + 0i + 0j + 0k
```

## Square Matrix of Quaternion Ones

Create an n-by-n matrix of quaternion ones.

```
n = 3;
quatOnes = ones(n, 'quaternion')

quatOnes=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];
quatOnesSyntax1 = ones(dims, 'quaternion')

quatOnesSyntax1 = 3x1x2 quaternion array
quatOnesSyntax1(:,:,1) =

 1 + 0i + 0j + 0k
 1 + 0i + 0j + 0k
 1 + 0i + 0j + 0k

quatOnesSyntax1(:,:,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:

```
quatOnesSyntax2 = ones(3,1,2, 'quaternion');
isequal(quatOnesSyntax1, quatOnesSyntax2)

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

### **quatOnes — 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](#)

### **Topics**

[“Rotations, Orientation, and Quaternions”](#)

## Introduced in R2018b

# 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=2×1 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 = 2×1
```

```
1  
5
```

```
qB = 2×1
```

```
2  
6
```

```
qC = 2×1
```

```
3  
7
```

```
qD = 2×1
```

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`, `d`, 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`**Topics**

“Rotations, Orientation, and Quaternions”

**Introduced in R2018b**

# 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=2x3 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

[exp](#) | [log](#)

**Objects**

quaternion

**Topics**

"Rotations, Orientation, and Quaternions"

**Introduced in R2018b**

## 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=3×3 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=1×3 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=2×2 quaternion array
 -2.4847 + 1.1659i - 0.37547j + 2.8068k      0.28786 - 0.29876i - 0.51231j - 4.2972k
 0.38986 - 3.66061i - 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

**Topics**

"Rotations, Orientation, and Quaternions"

**Introduced in R2018b**

# 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=2×1 quaternion array
 1 + 2i + 3j + 4k
 5 + 6i + 7j + 8k
```

```
B = 2;
C = A./B
```

```
C=2×1 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=2×2 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=2×2 quaternion array
 1 + 2i + 3j + 4k      2 + 3i + 4j + 5k
 3 + 4i + 5j + 6k      4 + 5i + 6j + 7k

C = A./B

C=2×2 quaternion array
 2.7 - 0.1i - 2.1j - 1.7k      2.2778 + 0.092593i - 0.46296j - 0.5740
 1.8256 - 0.081395i + 0.45349j - 0.24419k      1.4524 - 0.5i + 1.0238j - 0.2619
```

## 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  $p$ ,  $A./p = A.\lvert p$ .

---

### 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`

### Topics

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

# rotateframe

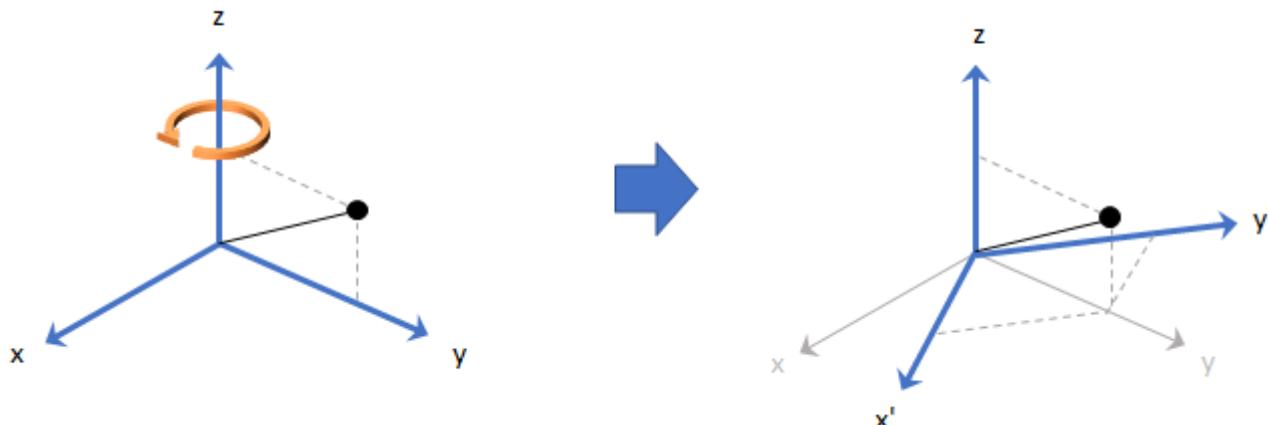
Quaternion frame rotation

## 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.

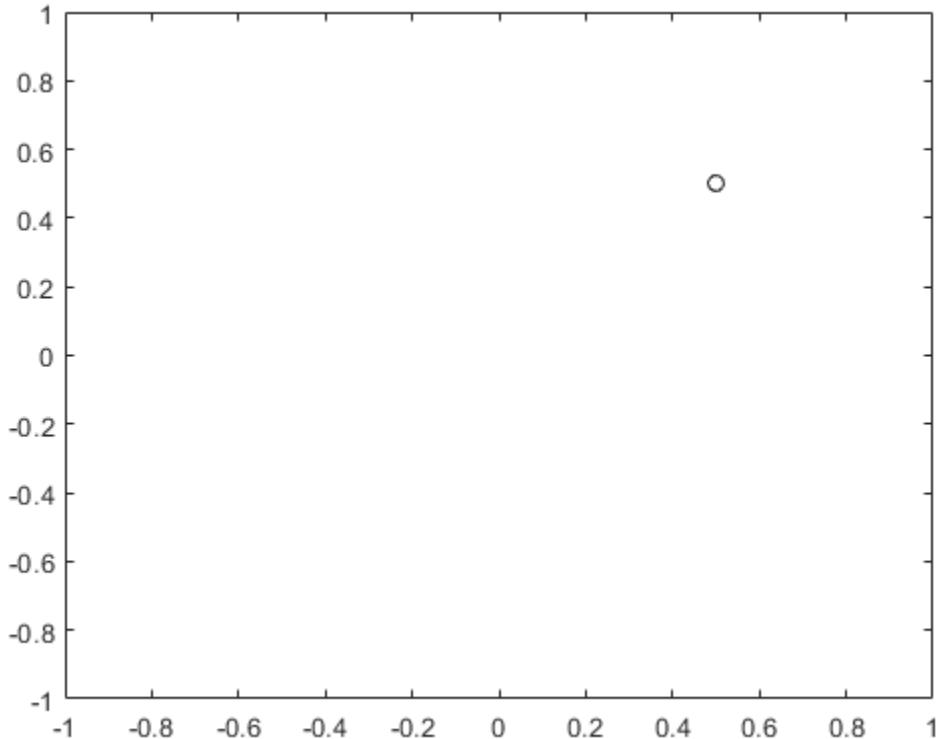


## Examples

### 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');

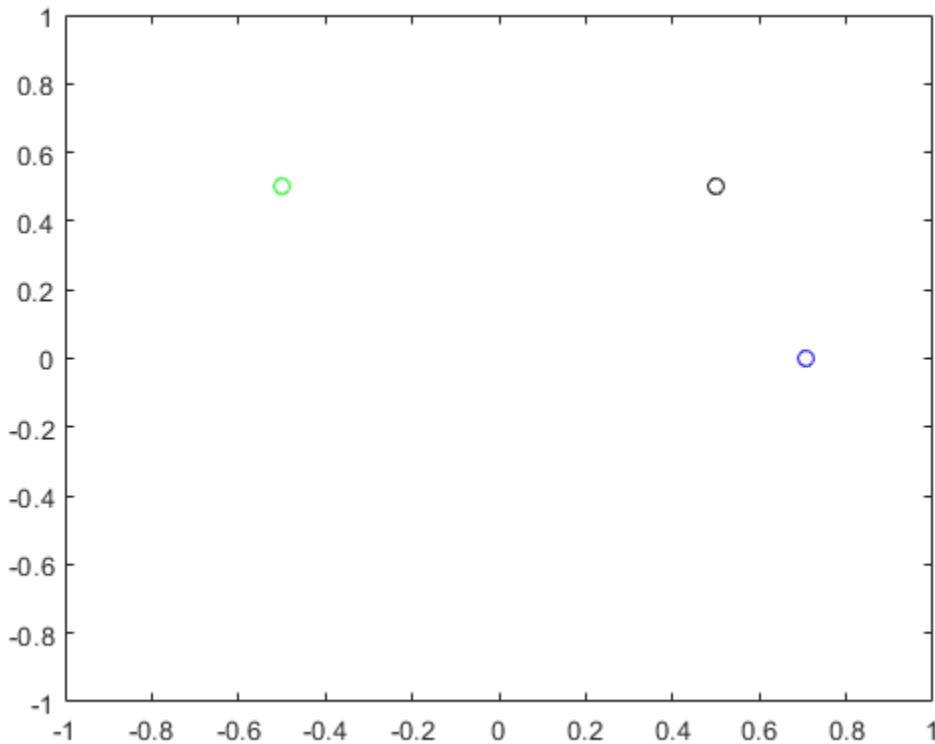
rereferencedPoint = rotateframe(quat,[x,y,z])

rereferencedPoint = 2×3

    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
```

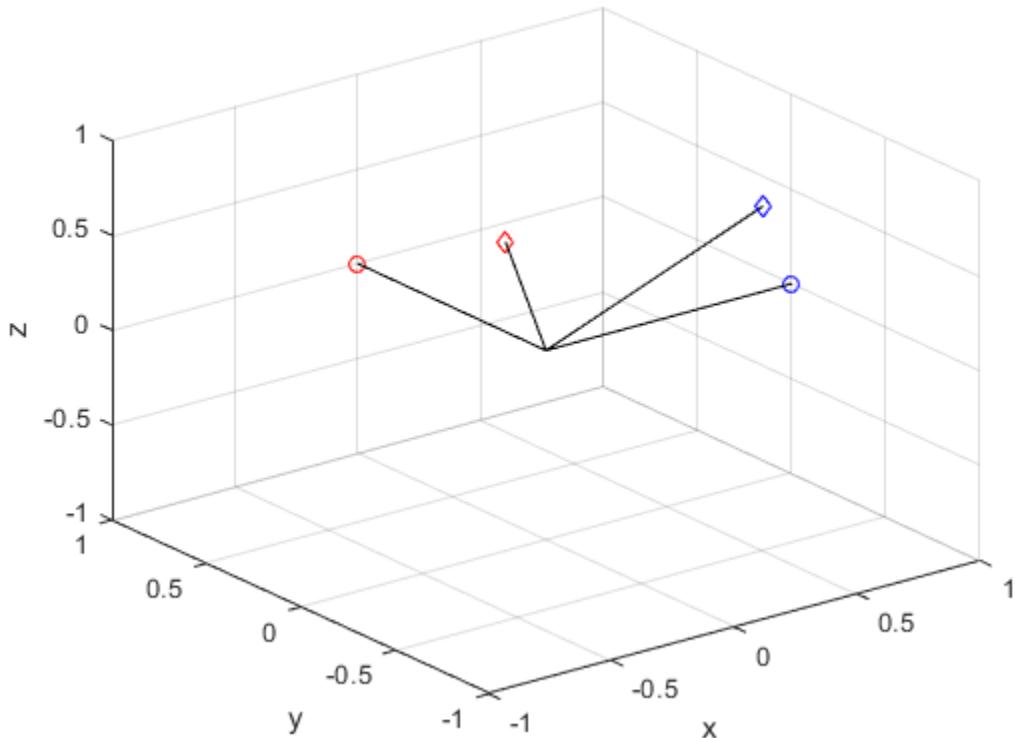
```

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`

### Topics

“Rotations, Orientation, and Quaternions”

**Introduced in R2018b**

## rotatepoint

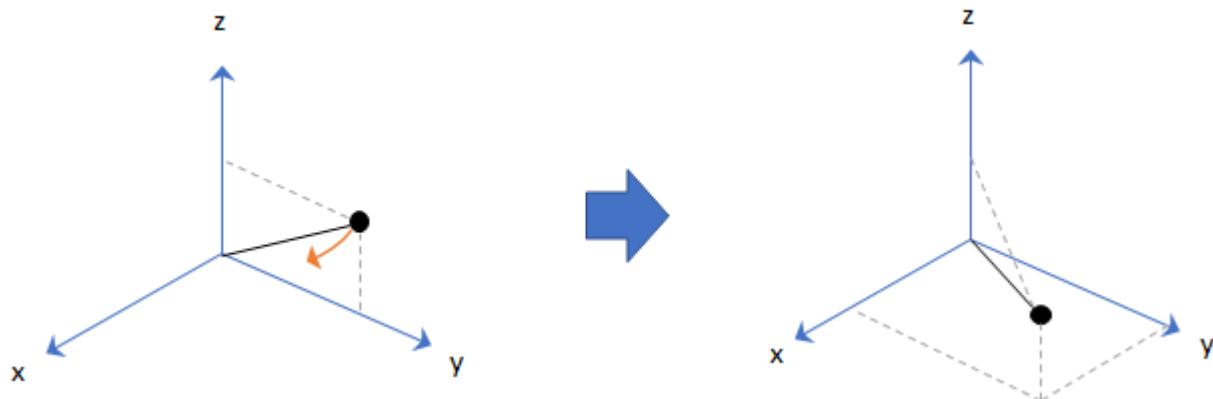
Quaternion point rotation

### 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.

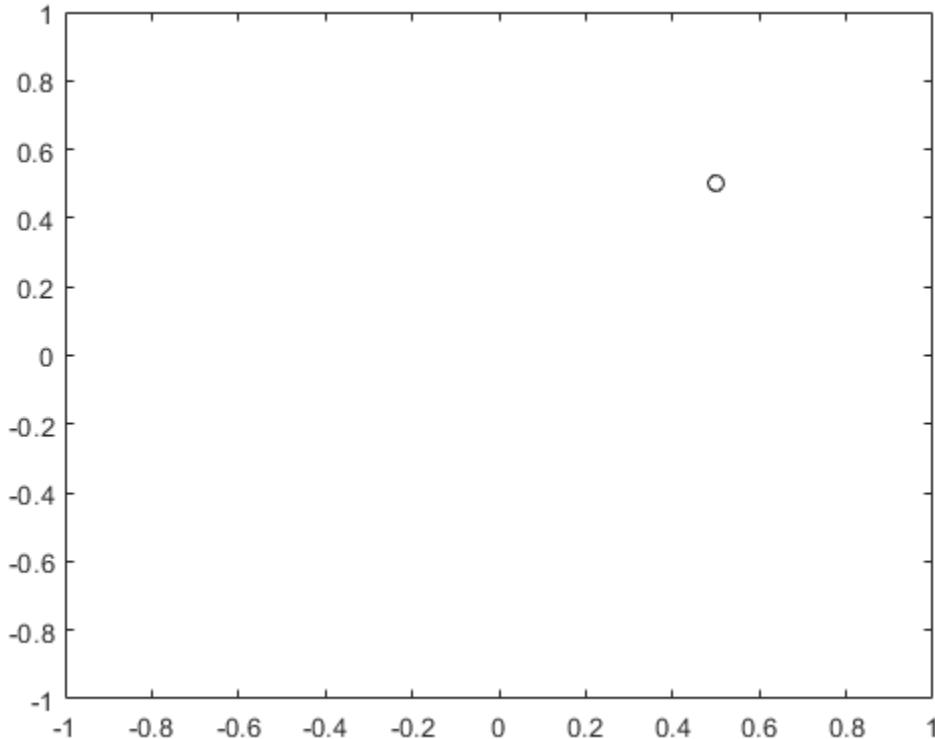


### Examples

#### 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.

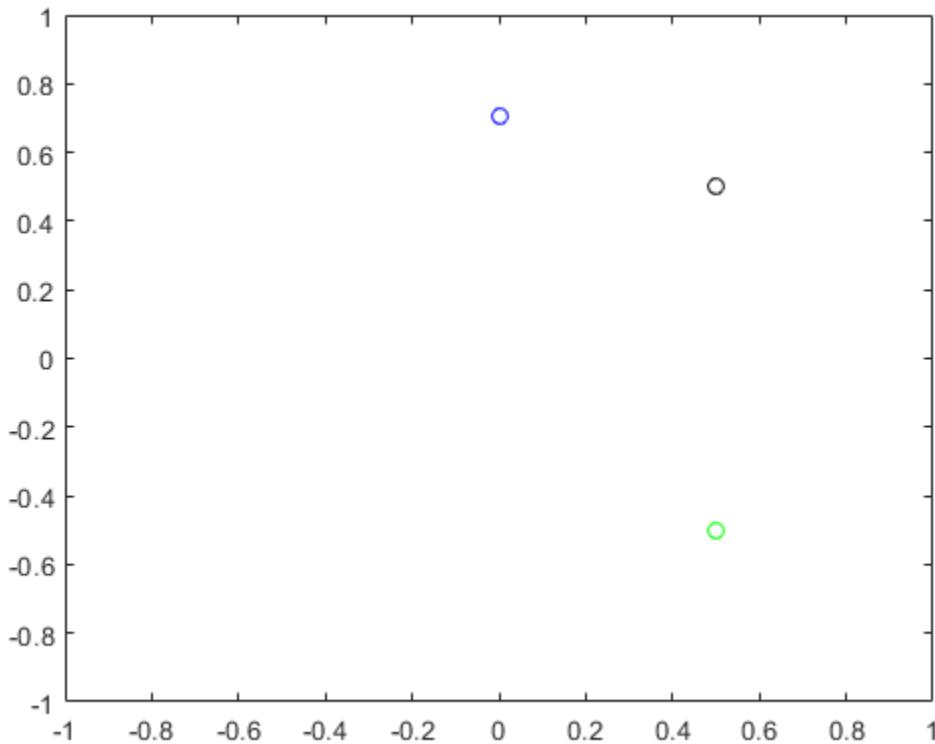
```
quat = quaternion([0,0,pi/4; ...
                  0,0,-pi/2], 'euler', 'XYZ', 'point');

rotatedPoint = rotatepoint(quat,[x,y,z])
rotatedPoint = 2x3

-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 = 2x3
```

```
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
```

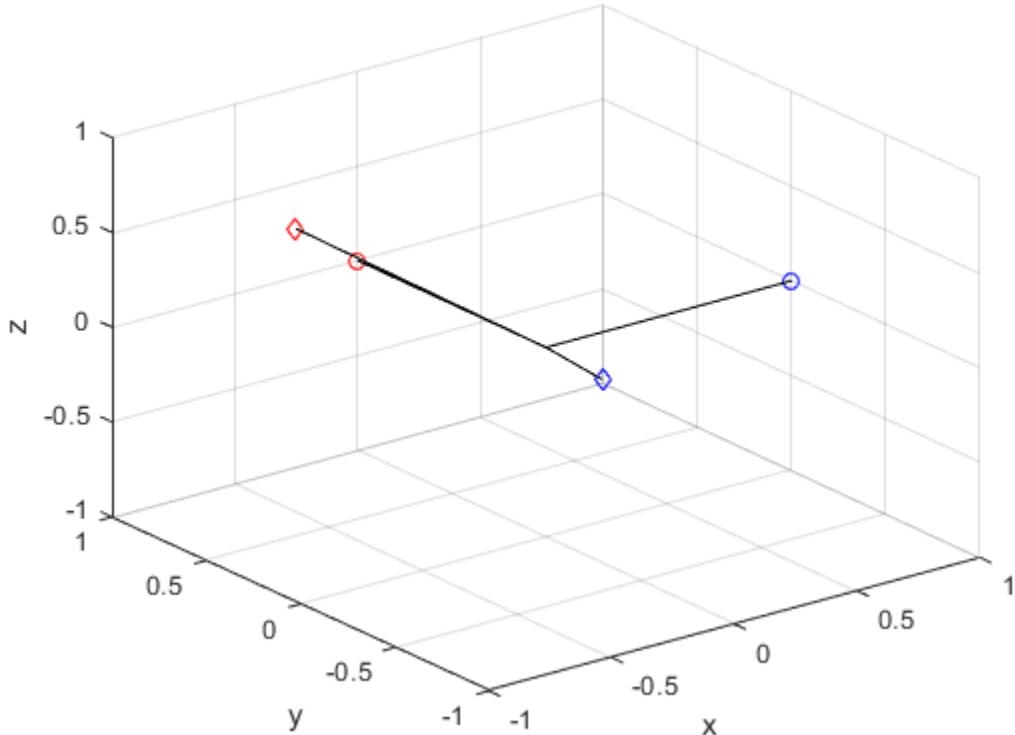
```

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`

### Topics

“Rotations, Orientation, and Quaternions”

**Introduced in R2018b**

## 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 = 3x3

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 = 3x3
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 = 3x3
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 = 3x3
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` = 1x3

```
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` = 3x1

```
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**

`euler` | `eulerrd` | `rotvec` | `rotvecd`

### **Objects**

`quaternion`

**Topics**

"Rotations, Orientation, and Quaternions"

**Introduced in R2018b**

# 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 = 1x3
    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

[euler](#) | [eulerd](#) | [rotvecd](#)

### Objects

[quaternion](#)

### Topics

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

# 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=2×1 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=3×3 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      -2
```

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=1×3 quaternion array  
0 + 0i + 0j + 0k 1 + 0i + 0j + 0k 0k  
  
b = quaternion(randn(4,1))  
b=4×1 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=4×3 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,

$$q = a_q + b_qi + c_qj + d_qk,$$

the product of  $q$  and a real scalar  $\beta$  is

$$\beta q = \beta a_q + \beta b_qi + \beta c_qj + \beta d_qk$$

**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,  $q = a_q + b_qi + c_qj + d_qk$ , and  $p = a_p + b_pi + c_pj + d_pk$ , the multiplication can be expanded as:

$$\begin{aligned} z = pq &= (a_p + b_p i + c_p j + d_p k)(a_q + b_q i + c_q j + d_q k) \\ &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\ &\quad + b_p a_q i + b_p b_q i^2 + b_p c_q i j + b_p d_q i k \\ &\quad + c_p a_q j + c_p b_q j i + c_p c_q j^2 + c_p d_q j k \\ &\quad + d_p a_q k + d_p b_q k i + d_p c_q k j + d_p d_q k^2 \end{aligned}$$

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

`mtimes`, `*` | `prod`

### Objects

`quaternion`

### Topics

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

# 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`

### **Topics**

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

# 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 + 0.71474
  1.8339 - 1.3077i + 2.7694j - 0.063055k      0.86217 + 0.34262i + 3.0349j - 0.20497
```

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 - 0.71474
 -1.8339 + 1.3077i - 2.7694j + 0.063055k      -0.86217 - 0.34262i - 3.0349j + 0.20497
```

## 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`

### Topics

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

## **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

### Topics

“Rotations, Orientation, and Quaternions”

## Introduced in R2018b

# 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 = 4x1

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 = 4x1

2.5000
1.0000
3.5000
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  $D$ -by- $N$  matrix

State noise, specified as a scalar or real-valued  $D$ -by- $N$  matrix.  $D$  is the number of motion dimensions 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

[cameas](#) | [cameasjac](#) | [constacc](#) | [constaccjac](#) | [constturn](#) | [constturnjac](#) | [constveljac](#) | [ctmeas](#) | [ctmeasjac](#) | [cvmeas](#) | [cvmeasjac](#)

### Objects

[trackingCKF](#) | [trackingEKF](#) | [trackingKF](#) | [trackingMSCEKF](#) | [trackingPF](#) | [trackingUKF](#)

### Introduced in R2018b

## constveljac

Jacobian for constant-velocity motion

### Syntax

```
jacobian = constveljac(state)
jacobian = constveljac(state,dt)
[jacobian,noisecjacobian] = 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,noisecjacobian] = constveljac(state,w,dt)` specifies the state noise, `w`, and returns the Jacobian, `noisecjacobian`, 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.

**noisejacobiann — 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**

`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac`

**Objects**

`trackingCKF` | `trackingEKF` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

**Introduced in R2018b**

# 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)
```

## 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`.

## 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 = 3x1

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 = 4x1

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 = 4x1

-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 = 4x1

-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 = 4x1

-116.5651
    0
  42.4853
 -17.8885

```

## 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]

Field	Description	Example
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 <b>HasElevation</b> 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 <b>HasVelocity</b> is false, the measurements are reported as [x y z]. If <b>HasVelocity</b> is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if <b>Orientation</b> performs a frame rotation from the parent coordinate frame to the child coordinate frame. When <b>IsParentToChild</b> is false, then <b>Orientation</b> performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

Data Types: struct

## Output Arguments

### **measurement — Measurement vector**

N-by-1 column vector

Measurement vector, returned as an  $N$ -by-1 column vector. The form of the 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.

<b>frame</b>	<b>measurement</b>																		
' <code>spherical</code> '	<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"><b>HasElevation</b></th></tr> <tr> <td></td><td></td><td>false</td><td>true</td></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>						<b>HasElevation</b>				false	true	<b>HasVelocity</b>	false	$[az; r]$	$[az; el; r]$	true	$[az; r; rr]$	$[az; el; r; rr]$
		<b>HasElevation</b>																	
		false	true																
<b>HasVelocity</b>	false	$[az; r]$	$[az; el; r]$																
	true	$[az; r; rr]$	$[az; el; r; rr]$																
' <code>rectangular</code> '	<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><b>HasVelocity</b></th><th>false</th><th><math>[x; y; z]</math></th></tr> </thead> <tbody> <tr> <td></td><td>true</td><td><math>[x; y; z; vx; vy; vz]</math></td></tr> </tbody> </table> <p>Position units are in meters and velocity units are in m/s.</p>				<b>HasVelocity</b>	false	$[x; y; z]$		true	$[x; y; z; vx; vy; vz]$									
<b>HasVelocity</b>	false	$[x; y; z]$																	
	true	$[x; y; z; vx; vy; vz]$																	

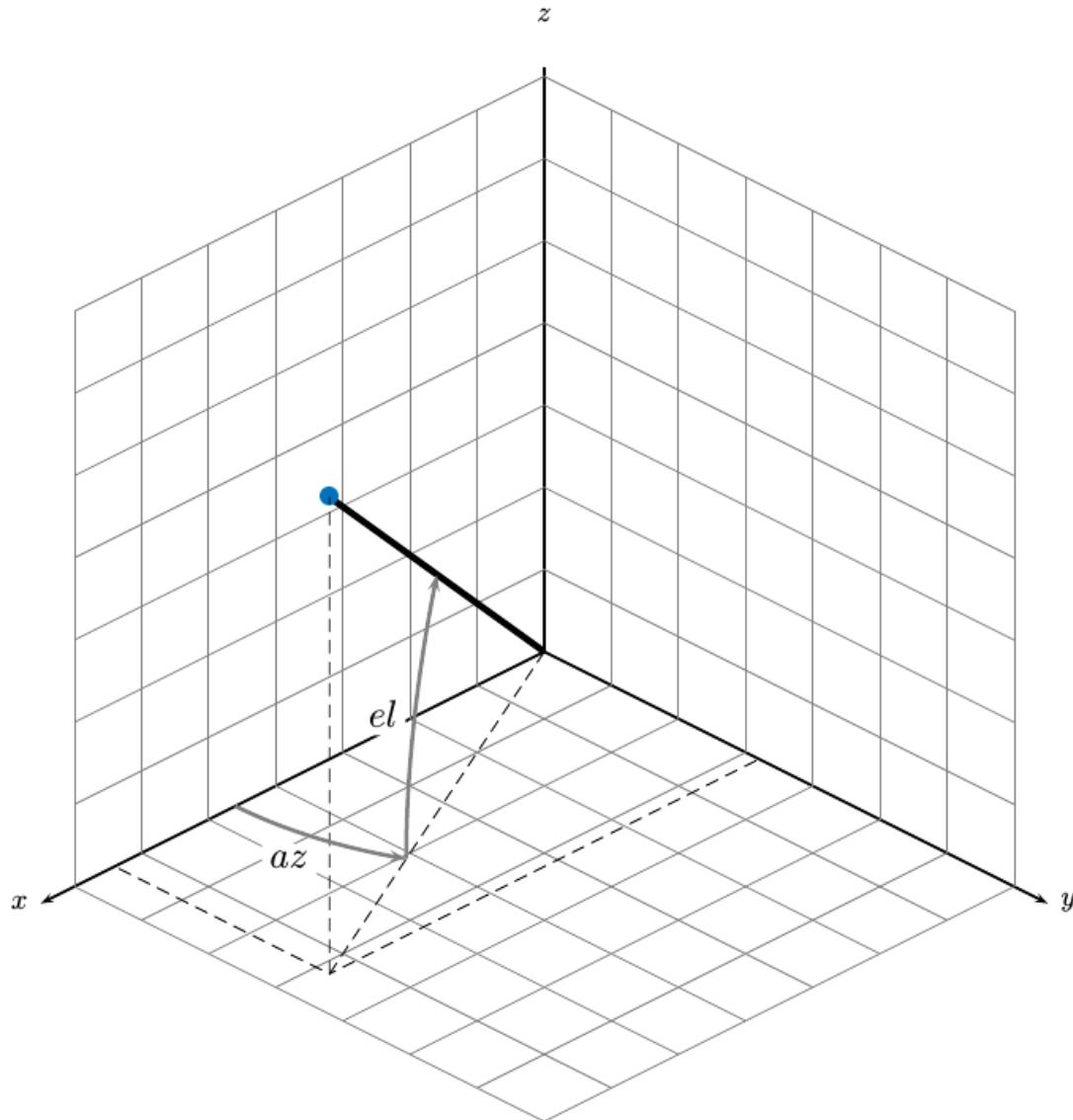
Data Types: double

## More About

### Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Sensor Fusion and Tracking 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

`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` |  
`constveljac` | `ctmeas` | `ctmeasjac` | `cvmeasjac`

### Objects

`trackingCKF` | `trackingEKF` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

**Introduced in R2018b**

# 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 = 3x4
```

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 <b>HasElevation</b> 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 <b>HasVelocity</b> is false, the measurements are reported as [x y z]. If <b>HasVelocity</b> is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if <b>Orientation</b> performs a frame rotation from the parent coordinate frame to the child coordinate frame. When <b>IsParentToChild</b> is false, then <b>Orientation</b> performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

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.

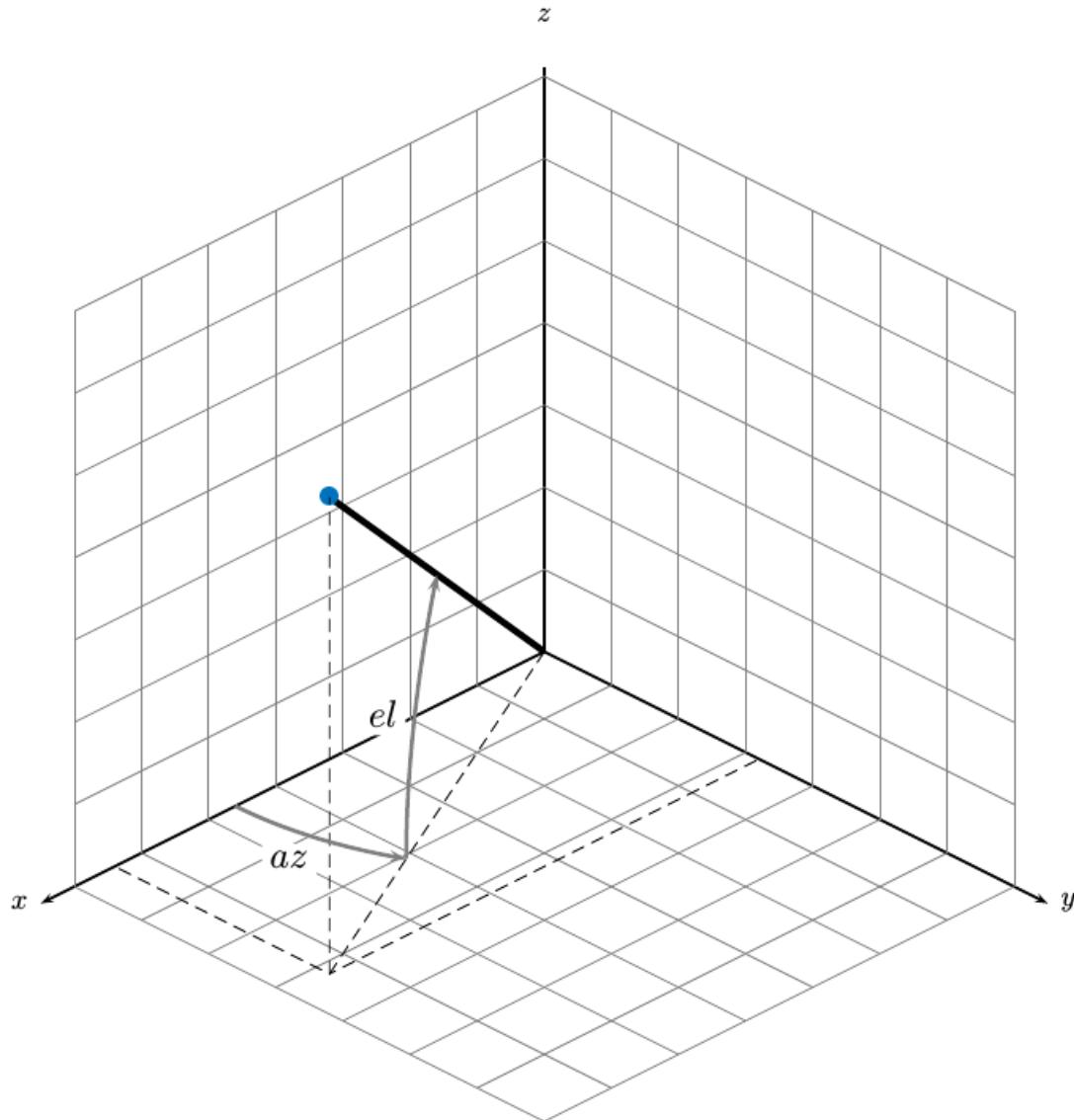
Frame	Measurement Jacobian
'spherical'	Jacobian of the measurement vector [az;el;r;rr] with respect to the state vector. Measurement vector components specify the azimuth angle, elevation angle, range, and range rate of the object with respect to the local sensor coordinate system. Angle units are in 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 Sensor Fusion and Tracking 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

`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas`

#### Objects

`trackingCKF` | `trackingEKF` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

**Introduced in R2018b**

# 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 = 6x1
```

```
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 = 6x1  
1.6250  
1.5000  
1.0000  
2.5000  
1.0000  
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,  $vx$  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 motion dimensions 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**

`cameas` | `cameasjac` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac`

### **Objects**

`trackingCKF` | `trackingEKF` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

## Introduced in R2018b

## constaccjac

Jacobian for constant-acceleration motion

### Syntax

```
jacobian = constaccjac(state)
jacobian = constaccjac(state,dt)
[jacobian,noisecjacobian] = 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,noisecjacobian] = constaccjac(state,w,dt)` specifies the state noise, `w`, and returns the Jacobian, `noisecjacobian`, 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 = 6x6
```

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 = 6x6
```

```
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**

`cameas` | `cameasjac` | `constacc` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac`

**Objects**

`trackingCKF` | `trackingEKF` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

**Introduced in R2018b**

## constvelmsc

Constant velocity (CV) motion model in MSC frame

### Syntax

```
state = constvelmsc(state,vNoise)
state = constvelmsc(state,vNoise,dt)
state = constvelmsc(state,vNoise,dt,u)
```

### Description

`state = constvelmsc(state,vNoise)` calculates the state at the next time-step based on current state and target acceleration noise, `vNoise`, in the scenario. The function assumes a time interval, `dt`, of one second, and zero observer acceleration in all dimensions.

`state = constvelmsc(state,vNoise,dt)` specifies the time interval, `dt`. The function assumes zero observer acceleration in all dimensions.

`state = constvelmsc(state,vNoise,dt,u)` specifies the observer input, `u`, during the time interval, `dt`.

### Examples

#### Predict Constant Velocity MSC State with Different Inputs

Define a state vector for a 3-D MSC state.

```
mscState = [0.1;0.01;0.1;0.01;0.001;1];
dt = 0.1;
```

Predict the state with zero observer acceleration.

```
mscState = constvelmsc(mscState,zeros(3,1),dt)
```

```
mscState = 6x1
```

```
0.1009
0.0083
0.1009
0.0083
0.0009
0.9091
```

Predict the state with [5;3;1] observer acceleration in scenario.

```
mscState = constvelmsc(mscState,zeros(3,1),dt,[5;3;1])
```

```
mscState = 6x1
```

```
0.1017
```

```
0.0067
0.1017
0.0069
0.0008
0.8329
```

Predict the state with observer maneuver and unit standard deviation random noise in target acceleration. Let observer acceleration in the time interval be  $[\sin(t) \cos(t)]$ .

```
velManeuver = [1 - cos(dt); sin(dt); 0];
posManeuver = [-sin(dt); cos(dt) - 1; 0];
u = zeros(6, 1);
u(1:2:end) = posManeuver;
u(2:2:end) = velManeuver;
mscState = constvelmsc(mscState, randn(3, 1), dt, u)

mscState = 6×1

0.1023
0.0058
0.1023
0.0057
0.0008
0.7689
```

### Predict and Measure State of Constant Velocity Target in Modified Spherical Coordinates

Define a state vector for a motion model in 2-D. The time interval is 2 seconds.

```
mscState = [0.5; 0.02; 1/1000; -10/1000];
dt = 2;
```

As modified spherical coordinates (MSC) state is relative, let the observer state be defined by a constant acceleration model in 2-D.

```
observerState = [100; 10; 0.5; 20; -5; 0.1];
```

Pre-allocate memory. rPlot is the range for plotting bearing measurements.

```
observerPositions = zeros(2, 10);
targetPositions = zeros(2, 10);
azimuthMeasurement = zeros(1, 10);
bearingHistory = zeros(2, 30);
rPlot = 2000;
```

Use a loop to predict the state multiple times. Use `constvelmsc` to create a trajectory with constant velocity target and measure the angles using the measurement function, `cvmeasmc`.

```
for i = 1:10
    obsAcceleration = observerState(3:3:end);
    % Use zeros(2,1) as process noise to get true predictions
    mscState = constvelmsc(mscState, zeros(2, 1), dt, obsAcceleration);

    % Update observer state using constant acceleration model
```

```

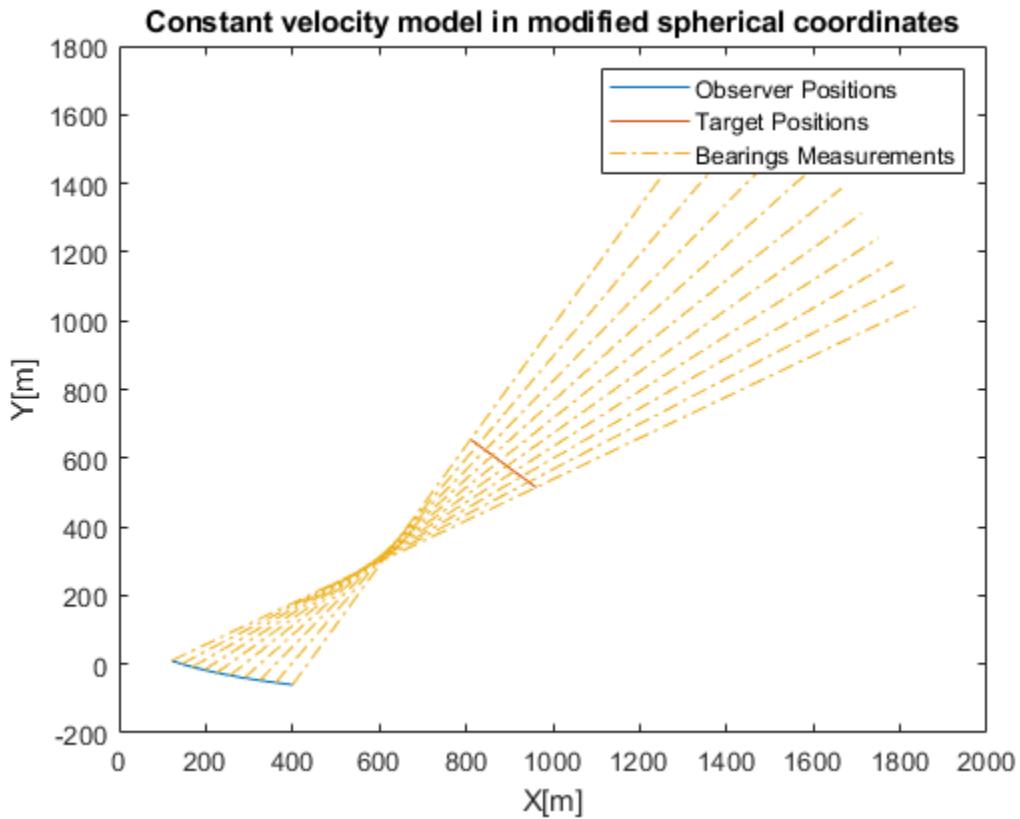
observerState = constacc(observerState,dt);
observerPositions(:,i) = observerState(1:3:end);

% Update bearing history with current measurement.
az = cvmeasmc(mscState);
bearingHistory(:,3*i-2) = observerState(1:3:end);
bearingHistory(:,3*i-1) = observerState(1:3:end) + [rPlot*cosd(az);rPlot*sind(az)];
bearingHistory(:,3*i) = [NaN;NaN];

% Use the 'rectangular' frame to get relative positions of the
% target using cvmeasmc function.
relativePosition = cvmeasmc(mscState,'rectangular');
relativePosition2D = relativePosition(1:2);
targetPositions(:,i) = relativePosition2D + observerPositions(:,i);
end

plot(observerPositions(1,:),observerPositions(2,:)); hold on;
plot(targetPositions(1,:),targetPositions(2,:));
plot(bearingHistory(1,:),bearingHistory(2,:),'-.');
title('Constant velocity model in modified spherical coordinates'); xlabel('X[m]'); ylabel('Y[m]');
legend('Observer Positions', 'Target Positions', 'Bearings Measurements'); hold off;

```



## Input Arguments

### **state — Relative state**

vector | 2-D matrix

State that is defined relative to an observer in modified spherical coordinates, specified as a vector or a 2-D matrix. For example, if there is a constant velocity target state,  $xT$ , and a constant velocity observer state,  $xO$ , then the `state` is defined as  $xT - xO$  transformed in modified spherical coordinates.

The two-dimensional version of modified spherical coordinates (MSC) is also referred to as the modified polar coordinates (MPC). In the case of:

- 2-D space -- State is equal to  $[az \ azRate \ 1/r \ vr/r]$
- 3-D space -- State is equal to  $[az \ omega \ el \ elRate \ 1/r \ vr/r]$

If specified as a matrix, states must be concatenated along columns, where each column represents a state following the convention specified above.

The variables used in the convention are:

- `az` -- Azimuth angle (rad)
- `el` -- Elevation angle (rad)
- `azRate` -- Azimuth rate (rad/s)
- `elRate` -- Elevation rate (rad/s)
- `omega` --  $azRate \times \cos(el)$  (rad/s)
- `1/r` -- 1/range (1/m)
- `vr/r` -- range-rate/range or inverse time-to-go (1/s)

Data Types: `single` | `double`

### **vNoise — Target acceleration noise**

`vector` | `matrix`

Target acceleration noise in the scenario, specified as a vector of 2 or 3 elements or a matrix with dimensions corresponding to `state`. That is, if the dimensions of the `state` matrix is 6-by-10, then the acceptable dimensions for `vNoise` is 3-by-10. If the dimensions of the `state` matrix is 4-by-10, then the acceptable dimensions for `vNoise` is 2-by-10. For more details, see “Orientation, Position, and Coordinate”.

Data Types: `double`

### **dt — Time difference**

`scalar`

Time difference between current state and the time at which the state is to be calculated, specified as a real finite numeric scalar.

Data Types: `single` | `double`

### **u — Observer input**

`vector`

Observer input, specified as a vector. The observer input can have the following impact on state-prediction based on its dimensions:

- When the number of elements in `u` equals the number of elements in `state`, the input `u` is assumed to be the maneuver performed by the observer during the time interval, `dt`. A maneuver is defined as motion of the observer higher than first order (or constant velocity).

- When the number of elements in `u` equals half the number of elements in `state`, the input `u` is assumed to be constant acceleration of the observer, specified in the scenario frame during the time interval, `dt`.

Data Types: double

## Output Arguments

### **state — State at next time step**

vector | 2-D matrix | 3-D matrix

State at the next time step, returned as a vector and a matrix of two or three dimensions. The state at the next time step is calculated based on the current state and the target acceleration noise, `vNoise`.

Data Types: double

## Algorithms

The function provides a constant velocity transition function in modified spherical coordinates (MSC) using a non-additive noise structure. The MSC frame assumes a single observer and the state is defined relative to it.

## Extended Capabilities

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

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

## See Also

### **Objects**

`trackingEKF` | `trackingMSCEKF`

### **Functions**

`constvelmscjac`

## Introduced in R2018b

# constvelmscjac

Jacobian of constant velocity (CV) motion model in MSC frame

## Syntax

```
[jacobianState,jacobianNoise] = constvelmscjac(state,vNoise)
[jacobianState,jacobianNoise] = constvelmscjac(state,vNoise,dt)
[jacobianState,jacobianNoise] = constvelmscjac(state,vNoise,dt,u)
```

## Description

[jacobianState,jacobianNoise] = constvelmscjac(state,vNoise) calculates the Jacobian matrix of the motion model with respect to the state vector and the noise. The input `state` defines the current state, and `vNoise` defines the target acceleration noise in the observer's Cartesian frame. The function assumes a time interval, `dt`, of one second, and zero observer acceleration in all dimensions.

The `trackingEKF` object allows you to specify the `StateTransitionJacobianFcn` property. The function can be used as a `StateTransitionJacobianFcn` when the `HasAdditiveProcessNoise` is set to `false`.

[jacobianState,jacobianNoise] = constvelmscjac(state,vNoise,dt) specifies the time interval, `dt`. The function assumes zero observer acceleration in all dimensions.

[jacobianState,jacobianNoise] = constvelmscjac(state,vNoise,dt,u) specifies the observer input, `u`, during the time interval, `dt`.

## Examples

### Compute Jacobian of State Transition Function

Define a state vector for 2-D MSC.

```
state = [0.5;0.01;0.001;0.01];
```

Calculate the Jacobian matrix assuming `dt` = 1 second, no observer maneuver, and zero target acceleration noise.

```
[jacobianState,jacobianNoise] = constvelmscjac(state,zeros(2,1)) %#ok
```

```
jacobianState = 4x4
```

1.0000	0.9900	-0.0000	-0.0098
-0.0000	0.9800	-0.0000	-0.0194
0.0000	-0.0000	0.9901	-0.0010
-0.0000	0.0194	-0.0000	0.9800

```
jacobianNoise = 4x2
10^-3 x
```

```
-0.2416    0.4321
-0.4851    0.8574
-0.0004    -0.0002
0.8574    0.4851
```

Calculate the Jacobian matrix, given dt = 0.1 seconds, no observer maneuver, and a unit standard deviation target acceleration noise.

```
[jacobianState,jacobianNoise] = constvelmscjac(state,randn(2,1),0.1) %#ok
```

```
jacobianState = 4x4
```

```
1.0000    0.0999    0.0067    -0.0001
-0.0001    0.9980    0.1348    -0.0020
-0.0000   -0.0000    0.9990    -0.0001
0.0001    0.0020    0.1351    0.9980
```

```
jacobianNoise = 4x2
10^-4 ×
```

```
-0.0240    0.0438
-0.4800    0.8755
-0.0000   -0.0000
0.8755    0.4800
```

Calculate the Jacobian matrix, given dt = 0.1 seconds and observer acceleration = [0.1 0.3] in the 2-D observer's Cartesian coordinates.

```
[jacobianState,jacobianNoise] = constvelmscjac(state,randn(2,1),0.1,[0.1;0.3])
```

```
jacobianState = 4x4
```

```
1.0000    0.0999    0.0081    -0.0001
0.0002    0.9980    0.1625    -0.0020
-0.0000   -0.0000    0.9990    -0.0001
0.0002    0.0020    -0.1795    0.9980
```

```
jacobianNoise = 4x2
10^-4 ×
```

```
-0.0240    0.0438
-0.4800    0.8756
-0.0000   -0.0000
0.8756    0.4800
```

## Input Arguments

### **state — Relative state**

vector

State that is defined relative to an observer in modified spherical coordinates, specified as a vector. For example, if there is a constant velocity target state,  $xT$ , and a constant velocity observer state,  $xO$ , then the **state** is defined as  $xT - xO$  transformed in modified spherical coordinates.

The two-dimensional version of modified spherical coordinates (MSC) is also referred to as the modified polar coordinates (MPC).

In case the motion is in:

- 2-D space -- State is equal to [ $az \ azRate \ 1/r \ vr/r$ ]
- 3-D space -- State is equal to [ $az \ omega \ el \ elRate \ 1/r \ vr/r$ ]

The variables used in the convention are:

- $az$  -- Azimuth angle (rad)
- $el$  -- Elevation angle (rad)
- $azRate$  -- Azimuth rate (rad/s)
- $elRate$  -- Elevation rate (rad/s)
- $\omega$  --  $azRate \times \cos(el)$  (rad/s)
- $1/r$  -- 1/range (1/m)
- $vr/r$  -- range-rate/range or inverse time-to-go (1/s)

Data Types: single | double

#### **vNoise — Target acceleration noise**

vector

Target acceleration noise in scenario, specified as a vector of 2 or 3 elements.

Data Types: double

#### **dt — Time difference**

scalar

Time difference between the current state and the time at which the state is to be calculated, specified as a real finite numeric scalar.

Data Types: single | double

#### **u — Observer input**

vector | 2-D matrix | 3-D matrix

Observer input, specified as a vector or a matrix. The observer input can have the following impact on state-prediction based on its dimensions:

- When the number of elements in **u** equals the number of elements in **state**, the input **u** is assumed to be the maneuver performed by the observer during the time interval, **dt**. A maneuver is defined as motion of the observer higher than first order (or constant velocity).
- When the number of elements in **u** equals half the number of elements in **state**, the input **u** is assumed to be constant acceleration of the observer, specified in the scenario frame during the time interval, **dt**.

Data Types: double

## Output Arguments

**jacobianState — Jacobian of predicted state**  
matrix

Jacobian of the predicted state with respect to the previous state, returned as an  $n$ -by- $n$  matrix, where  $n$  is the number of states in the state vector.

Data Types: double

**jacobianNoise — Jacobian of predicted state**  
matrix

Jacobian of the predicted state with respect to the noise elements, returned as an  $n$ -by- $m$  matrix. The variable  $n$  is the number of states in the state vector, and the variable  $m$  is the number of process noise terms. That is,  $m = 2$  for state in 2-D space, and  $m = 3$  for state in 3-D space.

For example, if the state vector is a 4-by-1 vector in a 2-D space, `vNoise` must be a 2-by-1 vector, and `jacobianNoise` is a 4-by-2 matrix.

If the state vector is a 6-by-1 vector in 3-D space, `vNoise` must be a 3-by-1 vector, and `jacobianNoise` is a 6-by-3 matrix.

Data Types: double

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

**Objects**  
`trackingEKF`

**Functions**  
`constvelmsc`

**Introduced in R2018b**

# cvmeasmc

Measurement based on constant velocity (CV) model in MSC frame

## Syntax

```
measurement = cvmeasmc(state)
measurement = cvmeasmc(state,frame)
measurement = cvmeasmc(state,frame,laxes)
measurement = cvmeasmc(state,measurementParameters)
```

## Description

`measurement = cvmeasmc(state)` provides the angular measurement (azimuth and elevation) of the state in the sensor frame described by the `state`.

Tracking filters require a definition of the `MeasurementFcn` property. The `cvmeasmc` function can be used as the `MeasurementFcn`. To use this `MeasurementFcn` with `trackerGNN` and `trackerTOMHT`, you can use the `trackingMSCEKF` filter.

`measurement = cvmeasmc(state,frame)` provides the measurement in the frame specified. The allowed values for `frame` are '`rectangular`' and '`spherical`'.

`measurement = cvmeasmc(state,frame,laxes)` specifies the axes of the sensor's coordinate system. The `laxes` input is a 3-by-3 matrix with each column specifying the direction of local `x`, `y` and `z` axes in the observer's Cartesian frame. The default for `laxes` is `[1 0 0;0 1 0;0 0 1]`.

`measurement = cvmeasmc(state,measurementParameters)` specifies the measurement parameters as a scalar struct or an array of struct.

## Examples

### Obtain Measurements in MSC Frame

Using the `cvmeasmc` function, you can obtain measurements of the state in the spherical and the rectangular frames.

#### Spherical Frame

Obtain the azimuth and elevation measurements from an MSC state.

```
mscState = [0.5;0;0.3;0;1e-3;1e-2];
cvmeasmc(mscState)
```

```
ans = 2x1
```

```
28.6479
17.1887
```

## Rectangular Frame

Obtain the position measurement from an MSC state. Specify the frame as a second input.

```
cvmeasmSC(mscState, 'rectangular')  
ans = 3x1  
838.3866  
458.0127  
295.5202
```

Alternatively, you can specify the frame using `measurementParameters`.

```
cvmeasmSC(mscState, struct('Frame', 'rectangular'))  
ans = 3x1  
838.3866  
458.0127  
295.5202
```

## Input Arguments

### **state — Relative state**

vector | matrix

State that is defined relative to an observer in modified spherical coordinates, specified as a vector or a 2-D matrix. For example, if there is a constant velocity target state,  $xT$ , and a constant velocity observer state,  $xO$ , then the `state` is defined as  $xT - xO$  transformed in modified spherical coordinates.

The two-dimensional version of modified spherical coordinates (MSC) is also referred to as the modified polar coordinates (MPC). In the case of:

- 2-D space -- State is equal to  $[az \ azRate \ 1/r \ vr/r]$ .
- 3-D space -- State is equal to  $[az \ omega \ el \ elRate \ 1/r \ vr/r]$ .

The variables used in the convention are:

- $az$  -- Azimuth angle (rad)
- $el$  -- Elevation angle (rad)
- $azRate$  -- Azimuth rate (rad/s)
- $elRate$  -- Elevation rate (rad/s)
- $\omega$  --  $azRate \times \cos(el)$  (rad/s)
- $1/r$  --  $1/range$  (1/m)
- $vr/r$  -- range-rate/range or inverse time-to-go (1/s)

If the input state is specified as a matrix, states must be concatenated along columns, where each column represents a state following the convention specified above. The output is a matrix with the same number of columns as the input, where each column represents the measurement from the corresponding state.

If the motion model is in 2-D space, values corresponding to elevation are assumed to be zero if elevation is requested as an output.

Data Types: `single` | `double`

#### **frame — Measurement frame**

`'spherical'` (default) | `'rectangular'`

Measurement frame, specified as `'spherical'` or `'rectangular'`. If using the `'rectangular'` frame, the three elements present in the measurement represent  $x$ ,  $y$ , and  $z$  position of the target in the observer's Cartesian frame. If using the `'spherical'` frame, the two elements present in the measurement represent azimuth and elevation measurement of the target. If not specified, the function provides the measurements in `'spherical'` frame.

#### **laxes — Direction of local axes**

`[1 0 0; 0 1 0; 0 0 1]` (default) | 3-by-3 matrix

Direction of local  $x$ ,  $y$ , and  $z$  axes in the scenario, specified as a 3-by-3 matrix. If not specified, `laxes` is equal to `[1 0 0; 0 1 0; 0 0 1]`.

Data Types: `double`

#### **measurementParameters — Measurement parameters**

scalar struct | array of struct

Measurement parameters, specified as a scalar struct or an array of struct. The structures must have the following fields (or a subset of them):

- **Frame** -- Either `'rectangular'` or `'spherical'` or an enumeration with the same values. Default: `'spherical'`.
- **Orientation** -- A 3-by-3 `laxes` matrix.
- **HasElevation** -- A logical scalar, `true` if elevation is measured. Default: `true` if `state` is in 3-D space, `false` if `state` is in 2-D space.
- **IsParentToChild** -- A logical scalar, `true` if the orientation is given as a parent to child frame rotation.

Data Types: `struct`

## **Output Arguments**

#### **measurement — Measurement from MSC state**

vector

Target measurement in MSC frame, returned as a:

- One-element vector -- When `HasElevation` is set to `false`, the vector contains azimuth as the only measurement.
- Two-element vector -- When the `frame` is set to `'spherical'`, the function measures the azimuth and elevation measurements from an MSC state.
- Three-element vector -- When the `frame` is set to `'rectangular'`, the function measures the position measurement from an MSC state.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Objects

`trackingMSCEKF`

### Functions

`constvelmsc` | `cvmeasmscjac` | `initcvmscekf`

## Introduced in R2018b

# cvmeasmcjac

Jacobian of measurement using constant velocity (CV) model in MSC frame

## Syntax

```
jacobian = cvmeasmcjac(state)
jacobian = cvmeasmcjac(state,frame)
jacobian = cvmeasmcjac(state,frame,laxes)
jacobian = cvmeasmcjac(state,measurementParameters)
```

## Description

`jacobian = cvmeasmcjac(state)` calculates the Jacobian with respect to angular measurement (azimuth and elevation) of the state in the sensor frame. The motion can be either in 2-D or 3-D space. If motion model is in 2-D space, values corresponding to elevation are assumed to be zero.

The `trackingEKF` and `trackingMSCEKF` filters require a definition of the `MeasurementJacobianFcn` property. The `cvmeasmcjac` function can be used as the `MeasurementJacobianFcn`. To use this `MeasurementFcn` with `trackerGNN` and `trackerTOMHT`, you can use the `trackingMSCEKF` filter.

`jacobian = cvmeasmcjac(state,frame)` provides the Jacobian measurement in the frame specified. The allowed values for frame are '`rectangular`' and '`spherical`'.

`jacobian = cvmeasmcjac(state,frame,laxes)` specifies the axes of the sensor's coordinate system. The `laxes` input is a 3-by-3 matrix with each column specifying the direction of local `x`, `y`, and `z` axes in the sensor coordinate system. The default for `laxes` is `[1 0 0; 0 1 0; 0 0 1]`.

`jacobian = cvmeasmcjac(state,measurementParameters)` specifies the measurement parameters as a struct.

## Examples

### Obtain Jacobian of State Measurements in MSC Frame

Using the `cvmeasmcjac` function, you can obtain the jacobian of the state measurements in the spherical and the rectangular frames.

#### Spherical Frame

Obtain the Jacobian of the azimuth and elevation measurements from an MSC state.

```
mscState = [0.5;0;0.3;0;1e-3;1e-2];
cvmeasmcjac(mscState)
```

`ans = 2×6`

57.2958	0	0	0	0	0
0	0	57.2958	0	0	0

## Rectangular Frame

Obtain the Jacobian of the position measurement from an MSC state. Specify the frame as a second input.

```
cvmeasmcjac(mscState, 'rectangular')

ans = 3x6
105 ×

-0.0046      0     -0.0026      0    -8.3839      0
 0.0084      0     -0.0014      0    -4.5801      0
    0      0     0.0096      0   -2.9552      0
```

Alternatively, you can specify the frame using `measurementParameters`.

```
cvmeasmcjac(mscState, struct('Frame', 'rectangular'))

ans = 3x6
105 ×

-0.0046      0     -0.0026      0    -8.3839      0
 0.0084      0     -0.0014      0    -4.5801      0
    0      0     0.0096      0   -2.9552      0
```

## Input Arguments

### state — Relative state

vector

State that is defined relative to an observer in modified spherical coordinates, as a vector. For example, if there is a target state,  $xT$ , and an observer state,  $xO$ , the `state` used by the function is  $xT - xO$ .

The 2-D version of modified spherical coordinates (MSC) is also referred to as the modified polar coordinates (MPC). In the case of:

- 2-D space -- State equals  $[az \ azRate \ 1/r \ vr/r]$ .
- 3-D space -- State equals  $[az \ omega \ el \ elRate \ 1/r \ vr/r]$ .

The variables used in the convention are:

- $az$  -- Azimuth angle (rad)
- $el$  -- Elevation angle (rad)
- $azRate$  -- Azimuth rate (rad/s)
- $elRate$  -- Elevation rate (rad/s)
- $\omega$  --  $azRate \times \cos(el)$  (rad/s)
- $1/r$  -- 1/range (1/m)
- $vr/r$  -- range-rate/range or inverse time-to-go (1/s)

If the motion model is in 2-D space, values corresponding to elevation are assumed to be zero if elevation is requested as an output.

Data Types: `single` | `double`

**frame — Measurement frame**

`'spherical'` (default) | `'rectangular'`

Measurement frame, specified as `'spherical'` or `'rectangular'`. If using the `'rectangular'` frame, the three rows present in `jacobian` represent the Jacobian of the measurements with respect to  $x$ ,  $y$ , and  $z$  position of the target in the sensor's Cartesian frame. If using the `'spherical'` frame, the two rows present in `jacobian` represent the Jacobian of the azimuth and elevation measurements of the target. If not specified, the function provides the Jacobian of the measurements in the `'spherical'` frame.

**laxes — Direction of local axes**

`[1 0 0;0 1 0;0 0 1]` (default) | 3-by-3 matrix

Direction of local  $x$ ,  $y$ , and  $z$  axes in the scenario, specified as a 3-by-3 matrix. Each column of the matrix specifies the direction of the local  $x$ ,  $y$ , and  $z$  axes in the sensor coordinate system. If not specified, the `laxes` is equal to `[1 0 0;0 1 0;0 0 1]`.

Data Types: `double`

**measurementParameters — Measurement parameters**

struct

Measurement parameters, specified as a struct. The structure must have the following fields (or a subset of them):

- **Frame** -- Either `'rectangular'` or `'spherical'`. Default: `'spherical'`.
- **Orientation** -- A 3-by-3 `laxes` matrix.
- **HasElevation** -- A logical scalar, `true` if elevation is measured. Default: `true` if `state` is in 3-D space, `false` if `state` is in 2-D space.
- **IsParentToChild** -- A logical scalar, `true` if the orientation is given as a parent to child frame rotation.

Data Types: `struct`

## Output Arguments

**jacobian — Measurement from MSC state**

matrix

Target measurement in MSC frame, returned as a:

- One-row matrix -- When `HasElevation` is set to `false`.
- Two-row matrix -- When the `frame` is set to `'spherical'`, the function measures the azimuth and elevation measurements from a MSC state.
- Three-row matrix -- When the `frame` is set to `'rectangular'`, the function measures the position measurement from a MSC state.

## Extended Capabilities

**C/C++ Code Generation**

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

## See Also

### Objects

`trackingMSCEKF`

### Functions

`constvelmsc | cvmeasmsc | initcvmscekf`

**Introduced in R2018b**

# 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)
```

## 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`.

## 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 = 3x1

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 = 4x1

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 = 4x1

-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 = 4x1

-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 = 4x1

-116.5651
    0
42.4853
-17.8885
```

## 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

**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'

Field	Description	Example
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 <b>HasElevation</b> 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 <b>HasVelocity</b> is false, the measurements are reported as [x y z]. If <b>HasVelocity</b> is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if <b>Orientation</b> performs a frame rotation from the parent coordinate frame to the child coordinate frame. When <b>IsParentToChild</b> is false, then <b>Orientation</b> performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

Data Types: struct

## Output Arguments

### **measurement — Measurement vector**

N-by-1 column vector

Measurement vector, returned as an  $N$ -by-1 column vector. The form of the 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.

<b>frame</b>	<b>measurement</b>																		
' <code>spherical</code> '	<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"><b>HasElevation</b></th></tr> <tr> <td></td><td></td><td>false</td><td>true</td></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>						<b>HasElevation</b>				false	true	<b>HasVelocity</b>	false	$[az; r]$	$[az; el; r]$	true	$[az; r; rr]$	$[az; el; r; rr]$
		<b>HasElevation</b>																	
		false	true																
<b>HasVelocity</b>	false	$[az; r]$	$[az; el; r]$																
	true	$[az; r; rr]$	$[az; el; r; rr]$																
' <code>rectangular</code> '	<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><b>HasVelocity</b></th><th>false</th><th><math>[x; y; z]</math></th></tr> </thead> <tbody> <tr> <td></td><td>true</td><td><math>[x; y; z; vx; vy; vz]</math></td></tr> </tbody> </table> <p>Position units are in meters and velocity units are in m/s.</p>				<b>HasVelocity</b>	false	$[x; y; z]$		true	$[x; y; z; vx; vy; vz]$									
<b>HasVelocity</b>	false	$[x; y; z]$																	
	true	$[x; y; z; vx; vy; vz]$																	

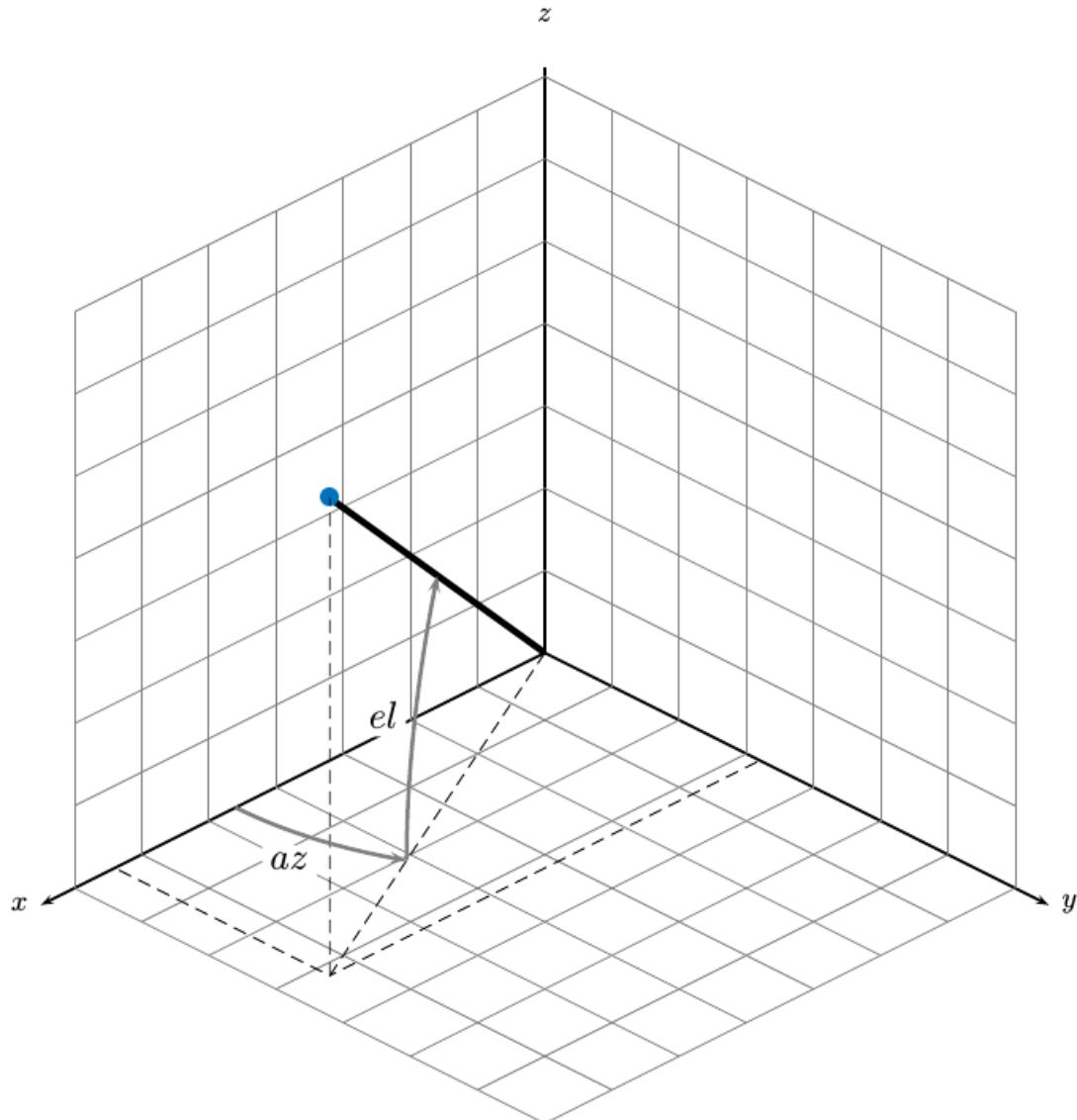
Data Types: double

## More About

### Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Sensor Fusion and Tracking 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

`cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` |  
`ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac`

### Objects

`trackingCKF` | `trackingEKF` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

**Introduced in R2018b**

# 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 = 3x6

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 = 4x6

-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 = 4x6

-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 = 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

```

Put the measurement parameters in a structure and use the alternative syntax.

```

measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel, ...
    'Orientation',laxes);
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,  $vx$  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]

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 <b>HasElevation</b> 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 <b>HasVelocity</b> is false, the measurements are reported as [x y z]. If <b>HasVelocity</b> is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if <b>Orientation</b> performs a frame rotation from the parent coordinate frame to the child coordinate frame. When <b>IsParentToChild</b> is false, then <b>Orientation</b> performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

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.

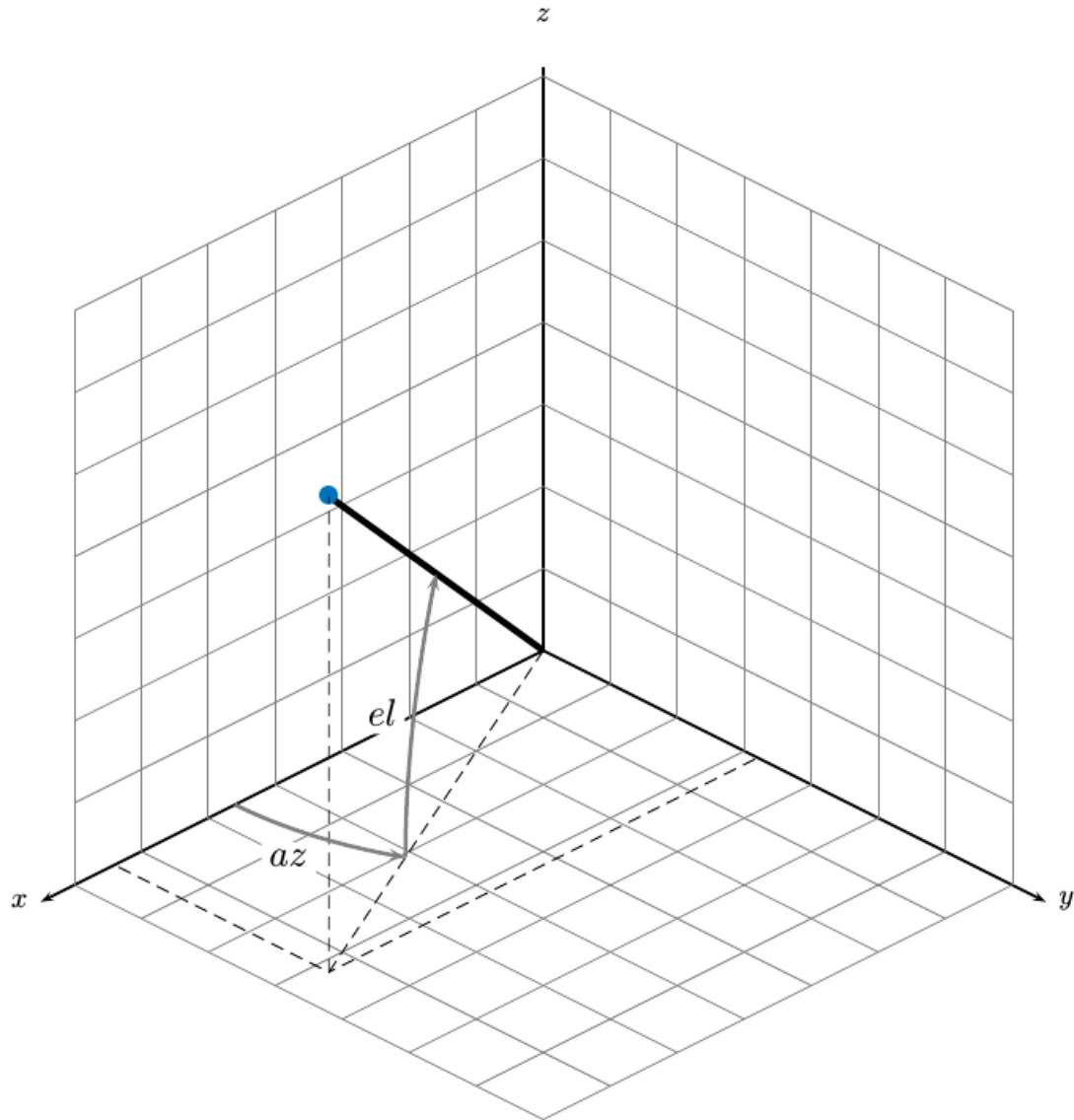
Frame	Measurement Jacobian
'rectangular'	Jacobian of the measurements $[x; y; z]$ with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
'spherical'	Jacobian of the measurement vector $[az; el; r; rr]$ with respect to the state 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 Sensor Fusion and Tracking 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

`cameas` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` |  
`ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac`

#### Objects

`trackingCKF` | `trackingEKF` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

**Introduced in R2018b**

# 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 = 5x1

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 = 5x1
```

```
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**

cameas | cameasjac | constacc | constaccjac | constturnjac | constvel | constveljac | ctmeas | ctmeasjac | cvmeas | cvmeasjac | initctekf | initctukf

### **Objects**

trackingEKF | trackingUKF

### **Introduced in R2018b**

# 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 = 5x5
```

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**

`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac` | `initctekf`

### **Objects**

`trackingEKF`

## Introduced in R2018b

# 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)
```

## 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`.

## 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 = 3x1

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 = 4x1

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 = 4x1

-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 = 4x1

-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 = 4x1

-116.5651
    0
  42.4853
 -17.8885

```

## 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]
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 <b>HasElevation</b> 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 <b>HasVelocity</b> is false, the measurements are reported as [x y z]. If <b>HasVelocity</b> is true, measurements are reported as [x y z vx vy vz].	1

Field	Description	Example
IsParentToChild	Logical scalar indicating if <code>Orientation</code> performs a frame rotation from the parent coordinate frame to the child coordinate frame. When <code>IsParentToChild</code> is <code>false</code> , then <code>Orientation</code> performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

Data Types: struct

## Output Arguments

### **measurement — Measurement vector**

*N*-by-1 column vector

Measurement vector, returned as an *N*-by-1 column vector. The form of the 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>HasElevation</th> </tr> <tr> <td></td> <td></td> <td>false true</td> </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	$[az; r; rr]$	$[az; el; r; rr]$											

frame	measurement								
'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"> <tr> <td><b>HasVelocity</b></td><td>false</td><td>[x;y;y]</td></tr> <tr> <td></td><td>true</td><td>[x;y;z;vx;vy;vz]</td></tr> </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]							

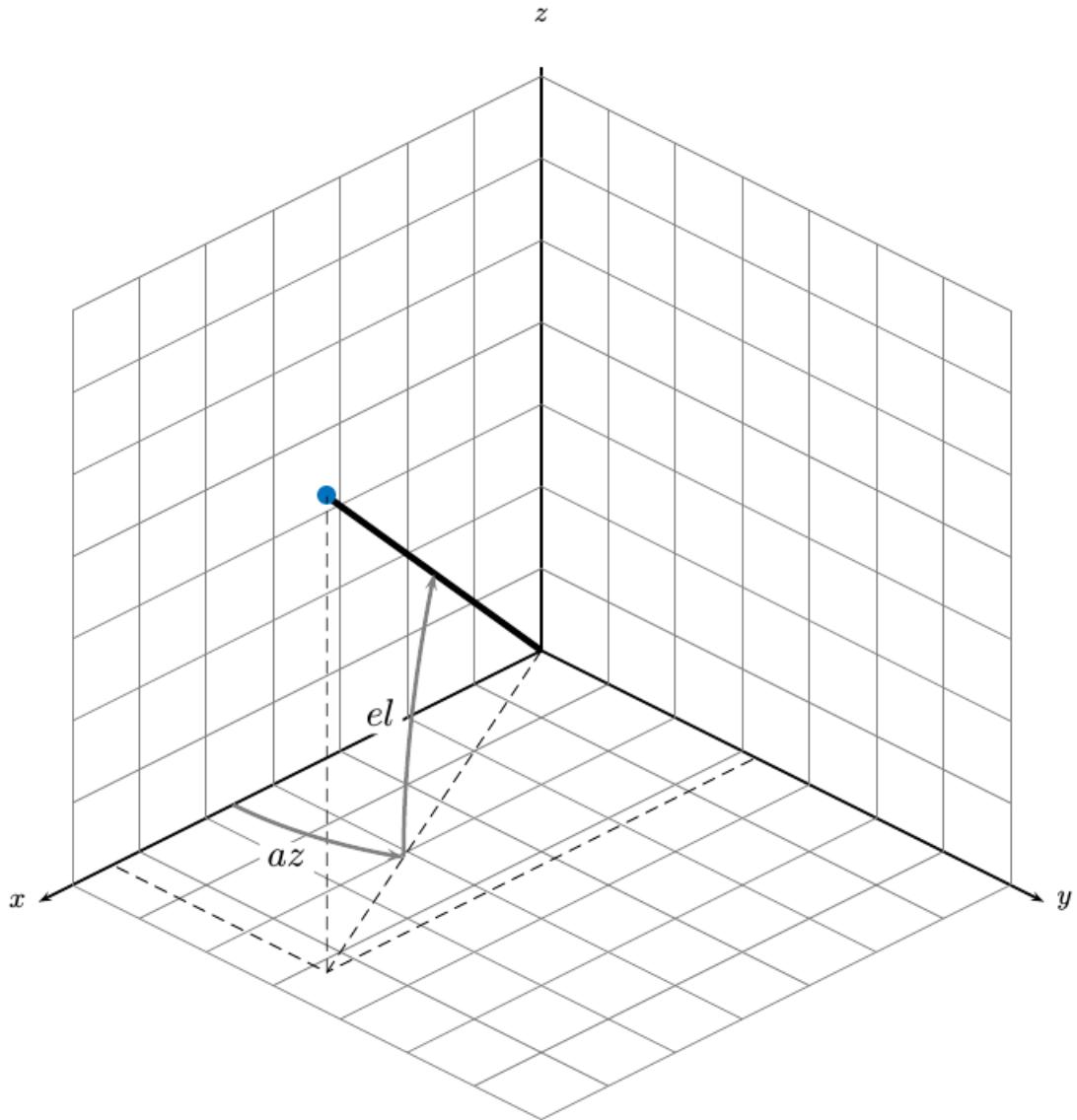
Data Types: double

## More About

### Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Sensor Fusion and Tracking 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

[cameas](#) | [cameasjac](#) | [constacc](#) | [constaccjac](#) | [constturn](#) | [constturnjac](#) | [constvel](#) | [constveljac](#) | [ctmeasjac](#) | [cvmeas](#) | [cvmeasjac](#)

#### Objects

[trackingCKF](#) | [trackingEKF](#) | [trackingKF](#) | [trackingMSCEKF](#) | [trackingPF](#) | [trackingUKF](#)

**Introduced in R2018b**

## 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 = 3x5
```

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'

Field	Description	Example
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 <b>HasElevation</b> 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 <b>HasVelocity</b> is false, the measurements are reported as [x y z]. If <b>HasVelocity</b> is true, measurements are reported as [x y z vx vy vz].	1
IsParentToChild	Logical scalar indicating if <b>Orientation</b> performs a frame rotation from the parent coordinate frame to the child coordinate frame. When <b>IsParentToChild</b> is false, then <b>Orientation</b> performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

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.

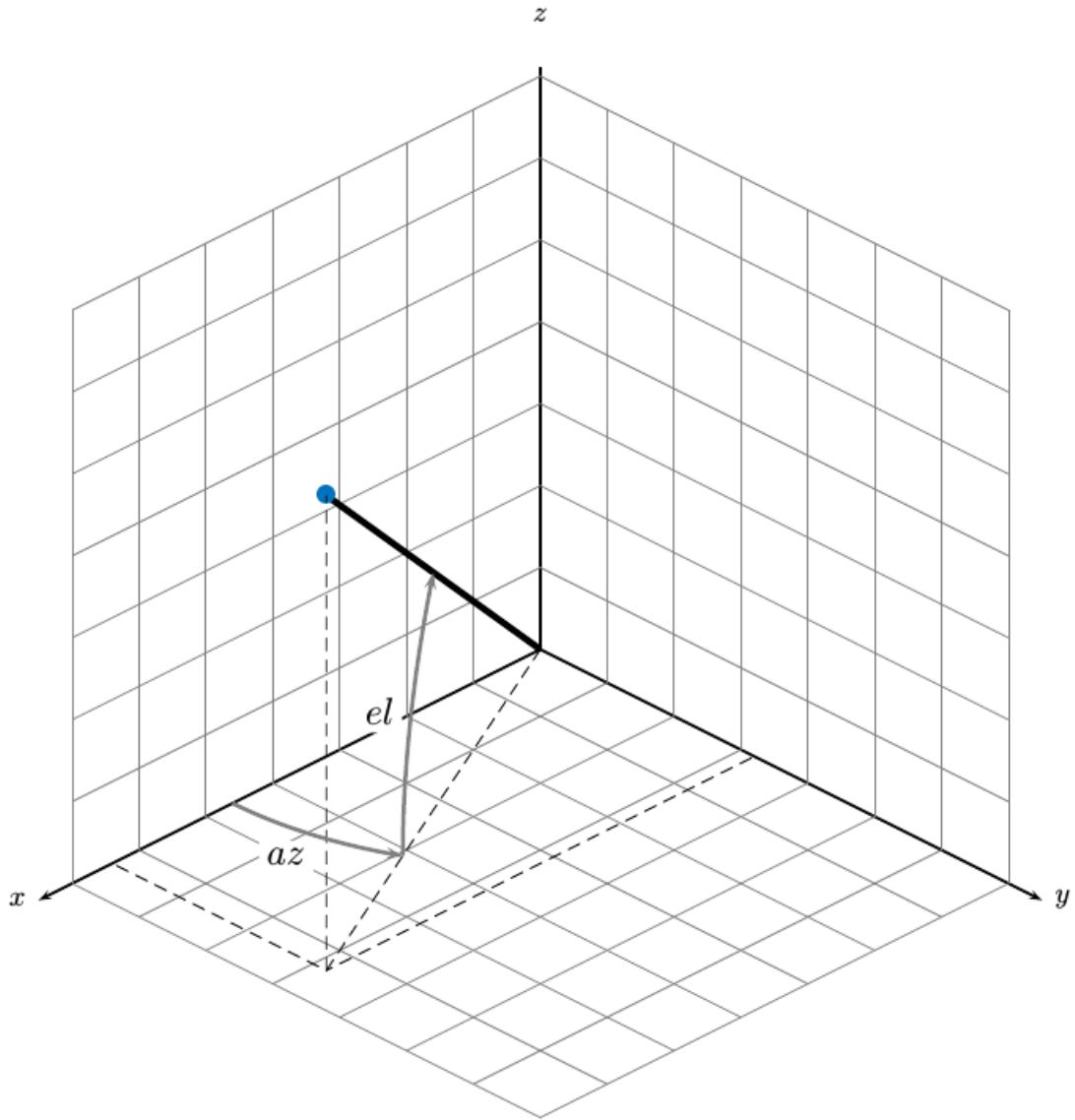
Frame	Measurement Jacobian
'rectangular'	Jacobian of the measurements $[x; y; z]$ with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
'spherical'	Jacobian of the measurement vector $[az; el; r; rr]$ with respect to the state 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 Sensor Fusion and Tracking 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

[cameas](#) | [cameasjac](#) | [constacc](#) | [constaccjac](#) | [constturn](#) | [constturnjac](#) | [constvel](#) | [constveljac](#) | [ctmeas](#) | [cvmeas](#) | [cvmeasjac](#)

#### Objects

[trackingCKF](#) | [trackingEKF](#) | [trackingKF](#) | [trackingMSCEKF](#) | [trackingPF](#) | [trackingUKF](#)

**Introduced in R2018b**

# 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 = trackerTOMHT('FilterInitializationFcn',@initcaekf);
```

Update the tracker with a single detection and get the tracks output.

```
detection = objectDetection(0,[10;-20;4], 'ObjectClassID',3);
tracks = step(tracker,detection,0)
```

```
tracks =
objectTrack with properties:
```

```
    TrackID: 1
    BranchID: 1
    SourceIndex: 0
    UpdateTime: 0
    Age: 1
    State: [9x1 double]
    StateCovariance: [9x9 double]
    StateParameters: [1x1 struct]
    ObjectClassID: 3
    TrackLogic: 'Score'
    TrackLogicState: [13.7102 13.7102]
    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 0 1 0 0 0 0 0; 0 0 0 0 0 0 1 0 0];
position = getTrackPositions(tracks, positionSelector)

position = 1x3

10.0000   -20.0000    4.0000

```

## Find Position and Covariance of 3-D Constant-Velocity Object

Create an extended Kalman filter tracker for 3-D constant-velocity motion.

```
tracker = trackerTOMHT('FilterInitializationFcn',@initcvekf);
```

Update the tracker with a single detection and get the tracks output.

```

detection = objectDetection(0,[10;3;-7], 'ObjectClassID',3);
tracks = step(tracker,detection,0)

tracks =
objectTrack with properties:

    TrackID: 1
    BranchID: 1
    SourceIndex: 0
    UpdateTime: 0
    Age: 1
    State: [6x1 double]
    StateCovariance: [6x6 double]
    StateParameters: [1x1 struct]
    ObjectClassID: 3
    TrackLogic: 'Score'
    TrackLogicState: [13.7102 13.7102]
    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 = 1x3

10.0000    3.0000   -7.0000

positionCovariance = 3x3

1.0000   -0.0000       0
-0.0000    1.0000   -0.0000
  0     -0.0000    1.0000

```

## Input Arguments

### **tracks — Track data structure**

struct array

Tracked object, specified as a struct array. A track struct array is an array of MATLAB® struct types containing sufficient information to obtain the track position vector and, optionally, the position covariance matrix. At a minimum, the struct must contain a State column vector field and a positive-definite StateCovariance matrix field. For an example of a track struct used by Sensor Fusion and Tracking Toolbox, examine the output argument, tracks, returned by the step object function of trackerGNN.

### **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.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

## 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](#) | [trackerGNN](#) | [trackerTOMHT](#)

## Introduced in R2018b

# 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 = trackerGNN('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], '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 1 0 0 0 0; 0 0 0 0 0 0 0 1 0];
```

```
velocity = getTrackVelocities(tracks,velocitySelector)
```

```
velocity = 1x3
```

```
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 = trackerGNN('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 1 0 0 0; 0 0 0 0 0 0 0 1 0];
[velocity,velocityCovariance] = getTrackVelocities(tracks,velocitySelector)

velocity = 1x3
```

```
1.0093 -0.6728 1.0093
```

```
velocityCovariance = 3x3
```

```
70.0685 0 0
0 70.0685 0
0 0 70.0685
```

## Input Arguments

### **tracks — Track data structure**

struct array

Tracked object, specified as a **struct** array. A track **struct** array is an array of MATLAB **struct** types containing sufficient information to obtain the track position vector and, optionally, the position covariance matrix. At a minimum, the **struct** must contain a **State** column vector field and a positive-definite **StateCovariance** matrix field. For an example of a track **struct** used by Sensor Fusion and Tracking Toolbox, examine the output argument, **tracks**, returned by the **step** object function of **trackerGNN**.

### **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.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

## 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` | `trackerGNN` | `trackerTOMHT`

### Introduced in R2018b

# 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`.

## 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**

`objectDetection` | `trackingABF` | `trackingEKF` | `trackingKF` | `trackingUKF`

**Introduced in R2018b**

# 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`.

## 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 = 4x1
```

```
1  
0  
3  
0
```

```
ABF.MeasurementNoise
```

```
ans = 2x2
```

```
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**

`objectDetection` | `trackingABF` | `trackingEKF` | `trackingKF` | `trackingUKF`

**Introduced in R2018b**

# initcackf

Create constant acceleration tracking cubature Kalman filter from detection report

## Syntax

```
ckf = initcackf(detection)
```

## Description

`ckf = initcackf(detection)` initializes a constant acceleration cubature Kalman filter for object tracking based on information provided in an `objectDetection` object, `detection`.

## Examples

### Create Constant Acceleration Tracking CKF Object from Rectangular Measurements

Create a constant acceleration tracking cubature Kalman filter object, `trackingCKF`, from an initial detection report. The detection report is made from an initial 3-D position measurement of the Kalman filter state in rectangular coordinates. You can obtain the 3-D position measurement using the constant acceleration measurement function, `cameas`.

This example uses the coordinates,  $x = 1$ ,  $y = 3$ ,  $z = 0$  and a 3-D position measurement noise of  $[1 \ 0.2 \ 0; \ 0.2 \ 2 \ 0; \ 0 \ 0 \ 1]$ .

```
detection = objectDetection(0, [1;3;0], 'MeasurementNoise', [1 0.2 0; 0.2 2 0; 0 0 1]);
```

Use `initcackf` to create a `trackingCKF` filter initialized at the provided position and using the measurement noise defined above.

```
ckf = initcackf(detection)

ckf =
    trackingCKF with properties:

        State: [9x1 double]
        StateCovariance: [9x9 double]

        StateTransitionFcn: @constacc
        ProcessNoise: [3x3 double]
        HasAdditiveProcessNoise: 0

        MeasurementFcn: @cameas
        MeasurementNoise: [3x3 double]
        HasAdditiveMeasurementNoise: 1
```

Check the values of the state and the measurement noise. Verify that the filter state, `ckf.State`, has the same position components as the detection measurement, `detection.Measurement`.

```
ckf.State
```

```
ans = 9x1
```

```
1  
0  
0  
3  
0  
0  
0  
0  
0
```

Verify that the filter measurement noise, `ckf.MeasurementNoise`, is the same as the `detection.MeasurementNoise` values.

```
ckf.MeasurementNoise
```

```
ans = 3x3
```

```
1.0000    0.2000         0  
0.2000    2.0000         0  
        0         0    1.0000
```

*Copyright 2018 The MathWorks, Inc.*

### Create Constant Acceleration Tracking CKF Object from Spherical Measurements

Create a constant acceleration tracking cubature Kalman filter object, `trackingCKF`, from an initial detection report. The detection report is made from an initial 3-D position measurement of the Kalman filter state in spherical coordinates. You can obtain the 3-D position measurement using the constant acceleration measurement function, `cameas`.

This example uses the coordinates, `az = 30`, `e1 = 5`, `r = 100`, `rr = 4` and a measurement noise of `diag([2.5, 2.5, 0.5, 1].^2)`.

```
meas = [30;5;100;4];  
measNoise = diag([2.5, 2.5, 0.5, 1].^2);
```

Use the `MeasurementParameters` property of the `detection` object to define the frame. When not defined, the fields of the `MeasurementParameters` struct use default values. In this example, sensor position, sensor velocity, orientation, elevation, and range rate flags are default.

```
measParams = struct('Frame','spherical');  
detection = objectDetection(0,meas,'MeasurementNoise',measNoise,...  
    'MeasurementParameters',measParams)  
  
detection =  
    objectDetection with properties:  
  
        Time: 0  
        Measurement: [4x1 double]  
        MeasurementNoise: [4x4 double]  
        SensorIndex: 1
```

```

ObjectClassID: 0
MeasurementParameters: [1x1 struct]
ObjectAttributes: {}

```

Use `initcackf` to create a `trackingCKF` filter initialized at the provided position and using the measurement noise defined above.

```

ckf = initcackf(detection)

ckf =
  trackingCKF with properties:

    State: [9x1 double]
    StateCovariance: [9x9 double]

    StateTransitionFcn: @constacc
    ProcessNoise: [3x3 double]
    HasAdditiveProcessNoise: 0

    MeasurementFcn: @cameas
    MeasurementNoise: [4x4 double]
    HasAdditiveMeasurementNoise: 1

```

Verify that the filter state produces the same measurement as above.

```

meas2 = cameas(ckf.State, measParams)

meas2 = 4×1

30.0000
5.0000
100.0000
4.0000

```

## 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

**ckf — Constant acceleration cubature Kalman filter**  
trackingCKF object

Constant acceleration cubature Kalman filter for object tracking, returned as a `trackingCKF` 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 `trackerTOMHT` and `trackerGNN` System objects.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

`cameas` | `constacc` | `initcaekf` | `initcakf` | `initcaukf` | `initctekf` | `initctukf` | `initcvekf` | `initcvkf` | `initcvukf`

### Objects

`objectDetection` | `trackingCKF` | `trackingEKF` | `trackingKF` | `trackingUKF`

### System Objects

`trackerGNN` | `trackerTOMHT`

**Introduced in R2018b**

# initcapf

Create constant acceleration tracking particle filter from detection report

## Syntax

```
pf = initcapf(detection)
```

## Description

`pf = initcapf(detection)` initializes a constant acceleration particle filter for object tracking based on information provided in an `objectDetection` object, `detection`.

## Examples

### Create Constant Acceleration Tracking PF Object from Rectangular Measurements

Create a constant acceleration tracking particle filter object, `trackingPF`, from an initial detection report. The detection report is made from an initial 3-D position measurement of the particle filter state in rectangular coordinates. You can obtain the 3-D position measurement using the constant acceleration measurement function, `cameas`.

This example uses the coordinates, `x = 1`, `y = 3`, `z = 0` and a 3-D position measurement noise of `[1 0.2 0; 0.2 2 0; 0 0 1]`.

```
detection = objectDetection(0, [1;3;0], 'MeasurementNoise', [1 0.2 0; 0.2 2 0; 0 0 1]);
```

Use `initcapf` to create a `trackingPF` filter initialized at the provided position and using the measurement noise defined above.

```
pf = initcapf(detection)

pf =
  trackingPF with properties:

    State: [9x1 double]
    StateCovariance: [9x9 double]
    IsStateVariableCircular: [0 0 0 0 0 0 0 0 0]

    StateTransitionFcn: @constacc
    ProcessNoiseSamplingFcn: []
    ProcessNoise: [3x3 double]
    HasAdditiveProcessNoise: 0

    MeasurementFcn: @cameas
    MeasurementLikelihoodFcn: []
    MeasurementNoise: [3x3 double]

    Particles: [9x1000 double]
    Weights: [1x1000 double]
    ResamplingPolicy: [1x1 trackingResamplingPolicy]
```

```
ResamplingMethod: 'multinomial'
```

Check the values of the state and the measurement noise. Verify that the filter state, `pf.State`, has approximately the same position components as the detection measurement, `detection.Measurement`.

```
pf.State
```

```
ans = 9x1
```

```
0.9674  
0.3690  
0.3827  
3.0317  
0.3056  
-0.5904  
0.0038  
0.0411  
-0.6815
```

Verify that the filter measurement noise, `pf.MeasurementNoise`, is the same as the `detection.MeasurementNoise` values.

```
pf.MeasurementNoise
```

```
ans = 3x3
```

```
1.0000    0.2000         0  
0.2000    2.0000         0  
0          0            1.0000
```

### Create Constant Acceleration Tracking PF Object from Spherical Measurements

Create a constant acceleration tracking particle filter object, `trackingPF`, from an initial detection report. The detection report is made from an initial 3-D position measurement of the particle filter state in spherical coordinates. You can obtain the 3-D position measurement using the constant acceleration measurement function, `cameas`.

This example uses the coordinates, `az = 30`, `e1 = 5`, `r = 100`, `rr = 4` and a measurement noise of `diag([2.5, 2.5, 0.5, 1].^2)`.

```
meas = [30;5;100;4];  
measNoise = diag([2.5, 2.5, 0.5, 1].^2);
```

Use the `MeasurementParameters` property of the `detection` object to define the frame. When not defined, the fields of the `MeasurementParameters` struct use default values. In this example, sensor position, sensor velocity, orientation, elevation, and range rate flags are default.

```
measParams = struct('Frame','spherical');  
detection = objectDetection(0,meas,'MeasurementNoise',measNoise,...  
'MeasurementParameters',measParams)
```

```

detection =
objectDetection with properties:

    Time: 0
    Measurement: [4x1 double]
    MeasurementNoise: [4x4 double]
    SensorIndex: 1
    ObjectClassID: 0
    MeasurementParameters: [1x1 struct]
    ObjectAttributes: {}

```

Use `initcapf` to create a `trackingPF` filter initialized at the provided position and using the measurement noise defined above.

```

pf = initcapf(detection)

pf =
trackingPF with properties:

    State: [9x1 double]
    StateCovariance: [9x9 double]
    IsStateVariableCircular: [0 0 0 0 0 0 0 0 0]

    StateTransitionFcn: @constacc
    ProcessNoiseSamplingFcn: []
        ProcessNoise: [3x3 double]
    HasAdditiveProcessNoise: 0

    MeasurementFcn: @cameas
    MeasurementLikelihoodFcn: []
        MeasurementNoise: [4x4 double]

    Particles: [9x1000 double]
        Weights: [1x1000 double]
    ResamplingPolicy: [1x1 trackingResamplingPolicy]
    ResamplingMethod: 'multinomial'

```

Verify that the filter state produces approximately the same measurement as `detection.Measurement`.

```

meas2 = cameas(pf.State, detection.MeasurementParameters)

meas2 = 4x1

29.9188
5.0976
99.8303
4.0255

```

## 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

**pf — Constant acceleration particle filter**  
`trackingPF` object

Constant acceleration particle filter for object tracking, returned as a `trackingPF` object.

## Algorithms

- The function configures the filter with 1000 particles. In creating the filter, 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 `trackerTOMHT` and `trackerGNN` System objects.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

`cameas` | `constacc` | `initcackf` | `initcaekf` | `initcakf` | `initcaukf` | `initctpf` | `initcvpf`

### Objects

`objectDetection` | `trackingEKF` | `trackingKF` | `trackingPF` | `trackingUKF`

### System Objects

`trackerGNN` | `trackerTOMHT`

## Introduced in R2018b

# initcvckf

Create constant velocity tracking cubature Kalman filter from detection report

## Syntax

```
ckf = initcvckf(detection)
```

## Description

`ckf = initcvckf(detection)` initializes a constant velocity cubature Kalman filter for object tracking based on information provided in an `objectDetection` object, `detection`.

## Examples

### Create Constant Velocity Tracking CKF Object from Rectangular Measurements

Create a constant velocity tracking cubature Kalman filter object, `trackingCKF`, from an initial detection report. The detection report is made from an initial 3-D position measurement of the Kalman filter state in rectangular coordinates. You can obtain the 3-D position measurement using the constant velocity measurement function, `cvmeas`.

This example uses the coordinates, `x = 1`, `y = 3`, `z = 0` and a 3-D position measurement noise of `[1 0.2 0; 0.2 2 0; 0 0 1]`.

```
detection = objectDetection(0, [1;3;0], 'MeasurementNoise', [1 0.2 0; 0.2 2 0; 0 0 1]);
```

Use `initcvckf` to create a `trackingCKF` filter initialized at the provided position and using the measurement noise defined above.

```
ckf = initcvckf(detection)

ckf =
    trackingCKF with properties:

        State: [6x1 double]
        StateCovariance: [6x6 double]

        StateTransitionFcn: @constvel
        ProcessNoise: [3x3 double]
        HasAdditiveProcessNoise: 0

        MeasurementFcn: @cvmeas
        MeasurementNoise: [3x3 double]
        HasAdditiveMeasurementNoise: 1
```

Check the values of the state and the measurement noise. Verify that the filter state, `ckf.State`, has the same position components as the detection measurement, `detection.Measurement`.

```
ckf.State
```

```
ans = 6x1
```

```
1  
0  
3  
0  
0  
0
```

Verify that the filter measurement noise, `ckf.MeasurementNoise`, is the same as the `detection.MeasurementNoise` values.

```
ckf.MeasurementNoise
```

```
ans = 3x3
```

```
1.0000    0.2000         0  
0.2000    2.0000         0  
0          0           1.0000
```

### Create Constant Velocity Tracking CKF Object from Spherical Measurements

Create a constant velocity tracking cubature Kalman filter object, `trackingCKF`, from an initial detection report. The detection report is made from an initial 3-D position measurement of the Kalman filter state in spherical coordinates. You can obtain the 3D position measurement using the constant velocity measurement function, `cvmeas`.

This example uses the coordinates, `az = 30`, `e1 = 5`, `r = 100`, `rr = 4` and a measurement noise of `diag([2.5, 2.5, 0.5, 1].^2)`.

```
meas = [30;5;100;4];  
measNoise = diag([2.5, 2.5, 0.5, 1].^2);
```

Use the `MeasurementParameters` property of the `detection` object to define the frame. When not defined, the fields of the `MeasurementParameters` struct use default values. In this example, sensor position, sensor velocity, orientation, elevation, and range rate flags are default.

```
measParams = struct('Frame','spherical');  
detection = objectDetection(0,meas,'MeasurementNoise',measNoise,...  
    'MeasurementParameters',measParams)  
  
detection =  
    objectDetection with properties:  
  
        Time: 0  
        Measurement: [4x1 double]  
        MeasurementNoise: [4x4 double]  
        SensorIndex: 1  
        ObjectClassID: 0  
        MeasurementParameters: [1x1 struct]  
        ObjectAttributes: {}
```

Use `initcvckf` to create a `trackingCKF` filter initialized at the provided position and using the measurement noise defined above.

```
ckf = initcvckf(detection)

ckf =
    trackingCKF with properties:

        State: [6x1 double]
        StateCovariance: [6x6 double]

        StateTransitionFcn: @constvel
        ProcessNoise: [3x3 double]
        HasAdditiveProcessNoise: 0

        MeasurementFcn: @cvmeas
        MeasurementNoise: [4x4 double]
        HasAdditiveMeasurementNoise: 1
```

Verify that the filter state produces the same measurement as above.

```
meas2 = cvmeas(ckf.State, measParams)

meas2 = 4×1

30.0000
5.0000
100.0000
4.0000
```

## 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

### **ckf — Constant velocity cubature Kalman filter for object tracking**

trackingCKF object

Constant velocity cubature Kalman filter for object tracking, returned as a `trackingCKF` 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 `trackerTOMHT` and `trackerGNN` System objects.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

`constvel` | `cvmeas` | `cvmeasjac` | `initcackf` | `initcaekf` | `initcakf` | `initcaukf` | `initctckf` | `initctekf` | `initctukf` | `initcvekf` | `initcvkf` | `initcvukf`

### Objects

`objectDetection` | `trackingCKF` | `trackingEKF` | `trackingKF` | `trackingUKF`

### System Objects

`trackerGNN` | `trackerTOMHT`

**Introduced in R2018b**

# initcvpf

Create constant velocity tracking particle filter from detection report

## Syntax

```
pf = initcvpf(detection)
```

## Description

`pf = initcvpf(detection)` initializes a constant velocity particle filter for object tracking based on information provided in an `objectDetection` object, `detection`.

## Examples

### Create Constant Velocity Tracking PF Object from Rectangular Measurements

Create a constant velocity tracking particle filter object, `trackingPF`, from an initial detection report. The detection report is made from an initial 3-D position measurement of the particle filter state in rectangular coordinates. You can obtain the 3-D position measurement using the constant velocity measurement function, `cvmeas`.

This example uses the coordinates, `x = 1`, `y = 3`, `z = 0` and a 3-D position measurement noise of `[1 0.2 0; 0.2 2 0; 0 0 1]`.

```
detection = objectDetection(0, [1;3;0], 'MeasurementNoise', [1 0.2 0; 0.2 2 0; 0 0 1]);
```

Use `initcvpf` to create a `trackingPF` filter initialized at the provided position and using the measurement noise defined above.

```
pf = initcvpf(detection)

pf =
  trackingPF with properties:

    State: [6×1 double]
    StateCovariance: [6×6 double]
    IsStateVariableCircular: [0 0 0 0 0 0]

    StateTransitionFcn: @constvel
    ProcessNoiseSamplingFcn: []
    ProcessNoise: [3x3 double]
    HasAdditiveProcessNoise: 0

    MeasurementFcn: @cvmeas
    MeasurementLikelihoodFcn: []
    MeasurementNoise: [3x3 double]

    Particles: [6×1000 double]
    Weights: [1×1000 double]
    ResamplingPolicy: [1x1 trackingResamplingPolicy]
```

```
ResamplingMethod: 'multinomial'
```

Check the values of the state and the measurement noise. Verify that the filter state, `pf.State`, has approximately the same position components as the detection measurement, `detection.Measurement`.

```
pf.State
```

```
ans = 6x1
```

```
0.9674  
0.3690  
3.0471  
0.2733  
0.0306  
-0.5904
```

Verify that the filter measurement noise, `pf.MeasurementNoise`, is the same as the `detection.MeasurementNoise` values.

```
pf.MeasurementNoise
```

```
ans = 3x3
```

```
1.0000    0.2000         0  
0.2000    2.0000         0  
0          0           1.0000
```

### Create Constant Velocity Tracking PF Object from Spherical Measurements

Create a constant velocity tracking particle filter object, `trackingPF`, from an initial detection report. The detection report is made from an initial 3-D position measurement of the particle filter state in spherical coordinates. You can obtain the 3-D position measurement using the constant velocity measurement function, `cvmeas`.

This example uses the coordinates, `az = 30`, `e1 = 5`, `r = 100`, `rr = 4` and a measurement noise of `diag([2.5, 2.5, 0.5, 1].^2)`.

```
meas = [30;5;100;4];  
measNoise = diag([2.5, 2.5, 0.5, 1].^2);
```

Use the `MeasurementParameters` property of the `detection` object to define the frame. When not defined, the fields of the `MeasurementParameters` struct use default values. In this example, sensor position, sensor velocity, orientation, elevation, and range rate flags are default.

```
measParams = struct('Frame','spherical');  
detection = objectDetection(0,meas,'MeasurementNoise',measNoise,...  
    'MeasurementParameters',measParams)  
  
detection =  
    objectDetection with properties:
```

```

    Time: 0
    Measurement: [4x1 double]
    MeasurementNoise: [4x4 double]
    SensorIndex: 1
    ObjectClassID: 0
    MeasurementParameters: [1x1 struct]
    ObjectAttributes: {}

```

Use `initcvpf` to create a `trackingPF` filter initialized at the provided position and using the measurement noise defined above.

```

pf = initcvpf(detection)

pf =
  trackingPF with properties:

    State: [6x1 double]
    StateCovariance: [6x6 double]
    IsStateVariableCircular: [0 0 0 0 0 0]

    StateTransitionFcn: @constvel
    ProcessNoiseSamplingFcn: []
      ProcessNoise: [3x3 double]
    HasAdditiveProcessNoise: 0

    MeasurementFcn: @cvmeas
    MeasurementLikelihoodFcn: []
      MeasurementNoise: [4x4 double]

    Particles: [6x1000 double]
      Weights: [1x1000 double]
    ResamplingPolicy: [1x1 trackingResamplingPolicy]
    ResamplingMethod: 'multinomial'

```

Verify that the filter state produces approximately the same measurement as `detection.Measurement`.

```

meas2 = cvmeas(pf.State, detection.MeasurementParameters)

meas2 = 4x1
29.9188
 5.0976
99.8303
 4.0255

```

## 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

**pf — Constant velocity particle filter**  
trackingPF object

Constant velocity particle filter for object tracking, returned as a trackingPF object.

## Algorithms

- The function configures the filter with 1000 particles. In creating the filter, the function computes the process noise matrix assuming a unit acceleration standard deviation.
- You can use this function as the FilterInitializationFcn property of trackerTOMHT and trackerGNN System objects.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

[constvel](#) | [cvmeas](#) | [initcapf](#) | [initctpf](#) | [initcvckf](#) | [initcvekf](#) | [initcvkf](#) | [initcvukf](#)

### Objects

[objectDetection](#) | [trackingEKF](#) | [trackingKF](#) | [trackingPF](#) | [trackingUKF](#)

### System Objects

[trackerGNN](#) | [trackerTOMHT](#)

### Introduced in R2018b

# initctckf

Create constant turn rate tracking cubature Kalman filter from detection report

## Syntax

```
ckf = initctckf(detection)
```

## Description

`ckf = initctckf(detection)` initializes a constant turn rate cubature Kalman filter for object tracking based on information provided in an `objectDetection` object, `detection`.

## Examples

### Create Constant Turn Rate Tracking CKF Object from Rectangular Measurements

Create a turn rate tracking cubature Kalman filter object, `trackingCKF`, from an initial detection report. The detection report is made from an initial 3-D position measurement of the Kalman filter state in rectangular coordinates. You can obtain the 3-D position measurement using the constant turn rate measurement function, `ctmeas`.

This example uses the coordinates,  $x = 1$ ,  $y = 3$ ,  $z = 0$  and a 3-D position measurement noise of  $[1 \ 0.2 \ 0; \ 0.2 \ 2 \ 0; \ 0 \ 0 \ 1]$ .

```
detection = objectDetection(0, [1;3;0], 'MeasurementNoise', [1 0.2 0; 0.2 2 0; 0 0 1]);
```

Use `initctckf` to create a `trackingCKF` filter initialized at the provided position and using the measurement noise defined above.

```
ckf = initctckf(detection)

ckf =
    trackingCKF with properties:

        State: [7x1 double]
        StateCovariance: [7x7 double]

        StateTransitionFcn: @constturn
        ProcessNoise: [4x4 double]
        HasAdditiveProcessNoise: 0

        MeasurementFcn: @ctmeas
        MeasurementNoise: [3x3 double]
        HasAdditiveMeasurementNoise: 1
```

Check the values of the state and the measurement noise. Verify that the filter state, `ckf.State`, has the same position components as the detection measurement, `detection.Measurement`.

```
ckf.State
```

```
ans = 7x1
```

```
1  
0  
3  
0  
0  
0  
0
```

Verify that the filter measurement noise, `ckf.MeasurementNoise`, is the same as the `detection.MeasurementNoise` values.

```
ckf.MeasurementNoise
```

```
ans = 3x3
```

```
1.0000    0.2000      0  
0.2000    2.0000      0  
0          0          1.0000
```

### Create Constant Turn Rate Tracking CKF Object from Spherical Measurements

Create a constant turn rate tracking cubature Kalman filter object, `trackingCKF`, from an initial detection report. The detection report is made from an initial 3-D position measurement of the Kalman filter state in spherical coordinates. You can obtain the 3-D position measurement using the constant turn rate measurement function, `ctmeas`.

This example uses the coordinates, `az = 30`, `e1 = 5`, `r = 100`, `rr = 4` and a measurement noise of `diag([2.5, 2.5, 0.5, 1].^2)`.

```
meas = [30;5;100;4];  
measNoise = diag([2.5, 2.5, 0.5, 1].^2);
```

Use the `MeasurementParameters` property of the `detection` object to define the frame. When not defined, the fields of the `MeasurementParameters` struct use default values. In this example, sensor position, sensor velocity, orientation, elevation, and range rate flags are default.

```
measParams = struct('Frame','spherical');  
detection = objectDetection(0,meas,'MeasurementNoise',measNoise,...  
    'MeasurementParameters',measParams)  
  
detection =  
    objectDetection with properties:  
  
        Time: 0  
        Measurement: [4x1 double]  
        MeasurementNoise: [4x4 double]  
        SensorIndex: 1  
        ObjectClassID: 0  
        MeasurementParameters: [1x1 struct]  
        ObjectAttributes: {}
```

Use `initctckf` to create a `trackingCKF` filter initialized at the provided position and using the measurement noise defined above.

```
ckf = initctckf(detection)

ckf =
  trackingCKF with properties:

    State: [7x1 double]
    StateCovariance: [7x7 double]

    StateTransitionFcn: @constturn
    ProcessNoise: [4x4 double]
    HasAdditiveProcessNoise: 0

    MeasurementFcn: @ctmeas
    MeasurementNoise: [4x4 double]
    HasAdditiveMeasurementNoise: 1
```

Verify that the filter state produces the same measurement as above.

```
meas2 = ctmeas(ckf.State, measParams)

meas2 = 4×1

30.0000
5.0000
100.0000
4.0000
```

## 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

**ckf — Constant turn rate cubature Kalman filter for object tracking**  
`trackingCKF` object

Constant turn rate cubature Kalman filter for object tracking, returned as a `trackingCKF` object.

## Algorithms

- The function computes the process noise matrix assuming a unit acceleration standard deviation and a unit angular acceleration standard deviation.
- You can use this function as the `FilterInitializationFcn` property of `trackerTOMHT` and `trackerGNN` System objects.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

[constturn](#) | [ctmeas](#) | [initcackf](#) | [initcaekf](#) | [initcakf](#) | [initcaukf](#) | [initctekf](#) | [initctukf](#) | [initcvckf](#) | [initcvekf](#) | [initcvkf](#) | [initcvukf](#)

### Objects

[objectDetection](#) | [trackingCKF](#) | [trackingEKF](#) | [trackingKF](#) | [trackingUKF](#)

### System Objects

[trackerGNN](#) | [trackerTOMHT](#)

## Introduced in R2018b

# initctpf

Create constant turn rate tracking particle filter from detection report

## Syntax

```
pf = initctpf(detection)
```

## Description

`pf = initctpf(detection)` initializes a constant turn rate particle filter for object tracking based on information provided in an `objectDetection` object, `detection`.

## Examples

### Create Constant Turn Rate Tracking PF Object from Rectangular Measurements

Create a constant turn rate tracking particle filter object, `trackingPF`, from an initial detection report. The detection report is made from an initial 3-D position measurement of the particle filter state in rectangular coordinates. You can obtain the 3-D position measurement using the constant turn rate measurement function, `ctmeas`.

This example uses the coordinates, `x = 1`, `y = 3`, `z = 0` and a 3-D position measurement noise of `[1 0.2 0; 0.2 2 0; 0 0 1]`.

```
detection = objectDetection(0, [1;3;0], 'MeasurementNoise', [1 0.2 0; 0.2 2 0; 0 0 1]);
```

Use `initctpf` to create a `trackingPF` filter initialized at the provided position and using the measurement noise defined above.

```
pf = initctpf(detection)

pf =
  trackingPF with properties:

    State: [7×1 double]
    StateCovariance: [7×7 double]
    IsStateVariableCircular: [0 0 0 0 0 0 0]

    StateTransitionFcn: @constturn
    ProcessNoiseSamplingFcn: []
    ProcessNoise: [4×4 double]
    HasAdditiveProcessNoise: 0

    MeasurementFcn: @ctmeas
    MeasurementLikelihoodFcn: []
    MeasurementNoise: [3×3 double]

    Particles: [7×1000 double]
    Weights: [1×1000 double]
    ResamplingPolicy: [1×1 trackingResamplingPolicy]
```

```
ResamplingMethod: 'multinomial'
```

Check the values of the state and the measurement noise. Verify that the filter state, `pf.State`, has approximately the same position components as the detection measurement, `detection.Measurement`.

```
pf.State
```

```
ans = 7x1
```

```
0.9674  
0.3690  
3.0471  
0.2733  
0.3056  
-0.0590  
0.0382
```

Verify that the filter measurement noise, `pf.MeasurementNoise`, is the same as the `detection.MeasurementNoise` values.

```
pf.MeasurementNoise
```

```
ans = 3x3
```

```
1.0000    0.2000         0  
0.2000    2.0000         0  
     0         0    1.0000
```

### Create Constant Turn Rate Tracking PF Object from Spherical Measurements

Create a constant turn rate tracking particle filter object, `trackingPF`, from an initial detection report. The detection report is made from an initial 3-D position measurement of the particle filter state in spherical coordinates. You can obtain the 3D position measurement using the constant turn rate measurement function, `ctmeas`.

This example uses the coordinates, `az = 30`, `e1 = 5`, `r = 100`, `rr = 4` and a measurement noise of `diag([2.5, 2.5, 0.5, 1].^2)`.

```
meas = [30;5;100;4];  
measNoise = diag([2.5, 2.5, 0.5, 1].^2);
```

Use the `MeasurementParameters` property of the `detection` object to define the frame. When not defined, the fields of the `MeasurementParameters` struct use default values. In this example, sensor position, sensor velocity, orientation, elevation, and range rate flags are default.

```
measParams = struct('Frame','spherical');  
detection = objectDetection(0,meas,'MeasurementNoise',measNoise,...  
    'MeasurementParameters',measParams)  
  
detection =  
    objectDetection with properties:
```

```

    Time: 0
    Measurement: [4x1 double]
    MeasurementNoise: [4x4 double]
    SensorIndex: 1
    ObjectClassID: 0
    MeasurementParameters: [1x1 struct]
    ObjectAttributes: {}

```

Use `initctpf` to create a `trackingPF` filter initialized at the provided position and using the measurement noise defined above.

```

pf = initctpf(detection)

pf =
  trackingPF with properties:

    State: [7x1 double]
    StateCovariance: [7x7 double]
    IsStateVariableCircular: [0 0 0 0 0 0 0]

    StateTransitionFcn: @constturn
    ProcessNoiseSamplingFcn: []
      ProcessNoise: [4x4 double]
    HasAdditiveProcessNoise: 0

    MeasurementFcn: @ctmeas
    MeasurementLikelihoodFcn: []
      MeasurementNoise: [4x4 double]

    Particles: [7x1000 double]
      Weights: [1x1000 double]
    ResamplingPolicy: [1x1 trackingResamplingPolicy]
    ResamplingMethod: 'multinomial'

```

Verify that the filter state produces approximately the same measurement as `detection.Measurement`.

```

meas2 = ctmeas(pf.State, detection.MeasurementParameters)

meas2 = 4×1

29.9188
 5.0976
99.8303
 4.0255

```

## 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

**pf — Constant turn rate particle filter**  
trackingPF object

Constant turn rate particle filter for object tracking, returned as a trackingPF object.

## Algorithms

- The function configures the filter with 1000 particles. In creating the filter, the function computes the process noise matrix assuming a unit acceleration standard deviation and a unit angular acceleration standard deviation.
- You can use this function as the FilterInitializationFcn property of trackerTOMHT and trackerGNN System objects.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

constturn | ctmeas | initcapf | initctckf | initctekf | initctukf | initcvpf

### Objects

objectDetection | trackingEKF | trackingKF | trackingPF | trackingUKF

### System Objects

trackerGNN | trackerTOMHT

## Introduced in R2018b

# initcaggiwphd

Create constant acceleration ggiwphd filter

## Syntax

```
phd = initcaggiwphd
phd = initcaggiwphd(detections)
```

## Description

`phd = initcaggiwphd` initializes a constant acceleration ggiwphd filter with no zeros components in the filter.

`phd = initcaggiwphd(detections)` initializes a constant acceleration ggiwphd filter based on information provided in object detections, `detections`. The function initializes a constant acceleration state with the same convention as `constacc` and `cameas`,  $[x;v_x;a_x;y;v_y;a_y;z;v_z;a_z]$ .

---

**Note** This initialization function is not compatible with `trackerGNN`, `trackerJPDA`, and `trackerTOMHT` system objects.

---

## Examples

### Initialize Constant Acceleration ggiwphd filter

Consider an object located at position [1;2;3] with detections uniformly spread around it's extent. The size of the extent is 1.2, 2.3 and 3.5 in x, y and z directions, respectively.

```
detections = cell(20,1);
location = [1;2;3];
dimensions = [1.2;2.3;3.5];
rng(2018) % Reproducible results
measurements = location + dimensions.*(-1 + 2*rand(3,20));
for i = 1:20
    detections{i} = objectDetection(0,measurements(:,i));
end
```

Initialize a constant acceleration ggiwphd filter with the generated detections.

```
phd = initcaggiwphd(detections);
```

Check the filter has the same position estimates as the mean of measurements.

```
states = phd.States
```

```
states = 9x1
```

```
1.2856
0
0
```

```
1.9950
    0
    0
2.9779
    0
    0

measurementMean = mean(measurements,2)
measurementMean = 3×1

1.2856
1.9950
2.9779
```

Check the extent and expected number of detections.

```
extent = phd.ScaleMatrices/(phd.DegreesOfFreedom - 4)
extent = 3×3

1.4603    0.0885   -0.2403
0.0885    3.0050   -0.0225
-0.2403   -0.0225   4.8365

expDetections = phd.Shapes/phd.Rates
expDetections = 20
```

## Input Arguments

### **detections — Object detections**

cell array of `objectDetection` objects

Object detections, specified as a cell array of `objectDetection` objects. You can create `detections` directly, or you can obtain `detections` from the outputs of sensor objects, such as `radarSensor`, `monostaticRadarSensor`, `irSensor`, and `sonarSensor`.

Example: `detection = objectDetection(0,[1;4.5;3], 'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **phd — ggiwphd filter**

`ggiwphd` object

`ggiwphd` filter, returned as a `ggiwphd` object.

## Algorithms

- You can use `initcaggiwphd` as the `FilterInitializationFcn` property of `trackingSensorConfiguration`.

- When detections are provided as input, the function adds one component to the density which reflects the mean of the detections. When the function is called without any inputs, a filter is initialized with no components in the density.
- The function uses the spread of measurements to describe the Inverse-Wishart distribution.
- The function uses the number of detections to describe the Gamma distribution.
- The function configures the process noise of the filter by assuming a unit standard deviation for the acceleration change rate.
- The function specifies a maximum of 500 components in the filter.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`ggiwphd` | `initctggiwphd` | `initcvggiwphd` | `trackerPHD`

### Introduced in R2019a

# initctggiwphd

Create constant turn-rate ggiwphd filter

## Syntax

```
phd = initctggiwphd  
phd = initctggiwphd(detections)
```

## Description

`phd = initctggiwphd` initializes a constant turn-rate ggiwphd filter with zero components in the filter.

`phd = initctggiwphd(detections)` initializes a constant turn-rate ggiwphd filter based on information provided in object detections, `detections`. 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.

---

**Note** This initialization function is not compatible with `trackerGNN`, `trackerJPDA`, and `trackerTOMHT` system objects.

---

## Examples

### Initialize Constant Turn-Rate ggiwphd filter

Consider an object located at position [1;2;3] with detections uniformly spread around it's extent. The size of the extent is 1.2, 2.3 and 3.5 in x, y and z directions, respectively.

```
detections = cell(20,1);  
location = [1;2;3];  
dimensions = [1.2;2.3;3.5];  
rng(2018) % Reproducible results  
measurements = location + dimensions.*(-1 + 2*rand(3,20));  
for i = 1:20  
    detections{i} = objectDetection(0,measurements(:,i));  
end
```

Initialize a constant turn-rate ggiwphd filter with the generated detections.

```
phd = initctggiwphd(detections);
```

Check the values of state in the filter has the same position estimates as the mean of measurements.

```
states = phd.States  
states = 7x1
```

```
1.2856  
0
```

```

1.9950
    0
    0
2.9779
    0

measurementMean = mean(measurements,2)
measurementMean = 3×1

1.2856
1.9950
2.9779

```

Check the extent and expected number of detections.

```

extent = phd.ScaleMatrices/(phd.DegreesOfFreedom - 4)
extent = 3×3

1.4603    0.0885   -0.2403
0.0885    3.0050   -0.0225
-0.2403   -0.0225   4.8365

expDetections = phd.Shapes/phd.Rates
expDetections = 20

```

## Input Arguments

### **detections — Object detections**

cell array of `objectDetection` objects

Object detections, specified as a cell array of `objectDetection` objects. You can create `detections` directly, or you can obtain `detections` from the outputs of sensor objects, such as `radarSensor`, `monostaticRadarSensor`, `irSensor`, and `sonarSensor`.

Example: `detection = objectDetection(0,[1;4.5;3], 'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **phd — ggiwphd filter**

`ggiwphd` object

`ggiwphd` filter, returned as a `ggiwphd` object.

## Algorithms

- You can use `initctggiwphd` as the `FilterInitializationFcn` property of `trackingSensorConfiguration`.

- When detections are provided as input, the function adds one component to the density which reflects the mean of the detections. When the function is called without any inputs, a filter is initialized with no components in the density.
- The function uses the spread of measurements to describe the Inverse-Wishart distribution.
- The function uses the number of detections to describe the Gamma distribution.
- The function configures the process noise of the filter by assuming a unit angular acceleration standard deviation.
- The function specifies a maximum of 500 components in the filter.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[ggiwphd](#) | [initcaggiwphd](#) | [initcvggiwphd](#) | [trackerPHD](#)

### Introduced in R2019a

# initcvrggiwphd

Create constant velocity ggiwphd filter

## Syntax

```
phd = initcvrggiwphd
phd = initcvrggiwphd(detections)
```

## Description

`phd = initcvrggiwphd` initializes a constant velocity ggiwphd filter with zero components in the filter.

`phd = initcvrggiwphd(detections)` initializes a constant velocity ggiwphd filter based on information provided in object detections, `detections`. The function initializes a constant velocity state with the same convention as `constvel` and `cvmeas`,  $[x;v_x;y;v_y;z;v_z]$ .

---

**Note** This initialization function is not compatible with `trackerGNN`, `trackerJPDA`, and `trackerTOMHT` system objects.

---

## Examples

### Initialize Constant Velocity ggiwphd filter

Consider an object located at position [1;2;3] with detections uniformly spread around it's extent. The size of the extent is 1.2, 2.3 and 3.5 in x, y and z directions, respectively.

```
detections = cell(20,1);
location = [1;2;3];
dimensions = [1.2;2.3;3.5];
rng(2018) % Reproducible results
measurements = location + dimensions.*(-1 + 2*rand(3,20));
for i = 1:20
    detections{i} = objectDetection(0,measurements(:,i));
end
```

Initialize a constant velocity ggiwphd filter with the generated detections.

```
phd = initcvrggiwphd(detections);
```

Check the values of state in the filter has the same position estimates as the mean of measurements.

```
states = phd.States
```

```
states = 6×1
```

```
1.2856
    0
1.9950
```

```
0  
2.9779  
0
```

```
measurementMean = mean(measurements,2)  
measurementMean = 3x1  
  
1.2856  
1.9950  
2.9779
```

Check the extent and expected number of detections.

```
extent = phd.ScaleMatrices/(phd.DegreesOfFreedom - 4)  
extent = 3x3  
  
1.4603    0.0885   -0.2403  
0.0885    3.0050   -0.0225  
-0.2403   -0.0225   4.8365  
  
expDetections = phd.Shapes/phd.Rates  
expDetections = 20
```

## Input Arguments

### **detections — Object detections**

cell array of `objectDetection` objects

Object detections, specified as a cell array of `objectDetection` objects. You can create `detections` directly, or you can obtain `detections` from the outputs of sensor objects, such as `radarSensor`, `monostaticRadarSensor`, `irSensor`, and `sonarSensor`.

Example: `detection = objectDetection(0,[1;4.5;3], 'MeasurementNoise', [1.0 0 0;  
0 2.0 0; 0 0 1.5])`

## Output Arguments

### **phd — ggiwphd filter**

`ggiwphd` object

`ggiwphd` filter, returned as a `ggiwphd` object.

## Algorithms

- You can use `initcvrggiwphd` as the `FilterInitializationFcn` property of `trackingSensorConfiguration`.
- When detections are provided as input, the function adds one component to the density which reflects the mean of the detections. When the function is called without any inputs, a filter is initialized with no components in the density.

- The function uses the spread of measurements to describe the Inverse-Wishart distribution.
- The function uses the number of detections to describe the Gamma distribution.
- The function configures the process noise of the filter by assuming a unit acceleration standard deviation.
- The function specifies a maximum of 500 components in the filter.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[ggiwphd](#) | [initcaggiwphd](#) | [initctggiwphd](#) | [trackerPHD](#)

### Introduced in R2019a

# initcagmphd

Create constant acceleration gmphd filter

## Syntax

```
phd = initcagmphd  
phd = initcagmphd(detections)
```

## Description

`phd = initcagmphd` initializes a constant acceleration gmphd filter with zero components in the filter.

`phd = initcagmphd(detections)` initializes a constant acceleration gmphd filter based on information provided in object detections, `detections`. The function initializes a constant acceleration state with the same convention as `constacc` and `cameas`,  $[x; v_x; a_x; y; v_y; a_y; z; v_z; a_z]$ .

---

**Note** This initialization function is not compatible with `trackerGNN`, `trackerJPDA`, and `trackerTOMHT` System objects.

---

## Examples

### Initialize Constant Acceleration gmphd Filter for Point Target

Consider a point target located at  $[1; 2; 3]$ . Create detection for the target using `objectDetection`.

```
detection = objectDetection(0,[1;2;3]);
```

Initialize a constant acceleration gmphd filter using `initcagmphd`.

```
phd = initcagmphd(detection);
```

Illustrate the initial state and the extent setup of the phd filter.

```
state = phd.States  
state = 9×1
```

```
1  
0  
0  
2  
0  
0  
3  
0  
0
```

```
extent = phd.HasExtent
extent = logical
0
```

### Initialize Constant Acceleration gmphd Filter for Extended Object

Consider an extended object located at [1;2;3]. The object's detections are uniformly distributed in x-, y-, and z-directions with dimensions of 1.2, 2.3, and 3.5, respectively. Generate 20 randomly distributed detections for the object using `objectDetection`.

```
detections = cell(20,1);
location = [1;2;3];
dimensions = [1.2;2.3;3.5];
rng(2019);
measurements = location + dimensions.*(-1 + 2*rand(3,20));
for i = 1:20
    detections{i} = objectDetection(0,measurements(:,i));
end
```

Initialize a constant acceleration gmphd filter using `initcagmphd`.

```
phd = initcagmphd(detections);
```

The initial state of the filter is same as the mean of the measurements.

```
state = phd.States
```

```
state = 9×1
```

```
1.1034
0
0
2.5597
0
0
2.4861
0
0
```

```
mean_measure = mean(measurements,2)
```

```
mean_measure = 3×1
```

```
1.1034
2.5597
2.4861
```

By default, the function sets the `HasExtent` property to true if the number of measurements is greater than 1.

```
extent = phd.HasExtent
```

```
extent = logical  
1
```

## Input Arguments

### **detections — Object detections**

cell array of `objectDetection` objects

Object detections, specified as a cell array of `objectDetection` objects. You can create `detections` directly, or you can obtain `detections` from the outputs of sensor objects, such as `radarSensor`, `monostaticRadarSensor`, `irSensor`, and `sonarSensor`.

Example: `detection = objectDetection(0,[1;4.5;3], 'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **phd — gmphd filter**

`gmphd` object

Gaussian mixture PHD filter, returned as a `gmphd` object.

## Algorithms

- You can use `initcagmphd` as the `FilterInitializationFcn` property of `trackingSensorConfiguration`.
- When detections are provided as input, the function adds one component to the density, which reflects the mean of the detections. When the function is called without any inputs, a filter is initialized with no components in the density.
- The function uses the spread of measurements to specify the positional covariance.
- The function configures the process noise of the filter by assuming a unit standard deviation for the acceleration change rate.
- The function specifies a maximum of 500 components in the filter.
- The function sets the `HasExtent` property of the filter to `true` if the number of input detections are greater than one.

## Extended Capabilities

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

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

## See Also

`gmphd` | `initctgmphd` | `initcvgmphd` | `trackerPHD`

## Introduced in R2019b

# initctgmphd

Create constant turn-rate gmphd filter

## Syntax

```
phd = initctgmphd
phd = initctgmphd(detections)
```

## Description

`phd = initctgmphd` initializes a constant turn-rate gmphd filter with zero components in the filter.

`phd = initctgmphd(detections)` initializes a constant turn-rate gmphd filter based on information provided in object detections, `detections`. 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.

---

**Note** This initialization function is not compatible with `trackerGNN`, `trackerJPDA`, and `trackerTOMHT` System objects.

---

## Examples

### Initialize Constant Turn-Rate gmphd Filter for Point Target

Consider a point target located at  $[1; 2; 3]$ . Create detection for the target using `objectDetection`.

```
detection = objectDetection(0,[1;2;3]);
```

Initialize a constant turn-rate gmphd filter using `initctgmphd`.

```
phd = initctgmphd(detection);
```

Display the initial state and the extent setup of the filter.

```
state = phd.States
```

```
state = 7×1
```

```
1
0
2
0
0
3
0
```

```
extent = phd.HasExtent
```

```
extent = logical  
0
```

### Initialize Constant Turn-Rate gmphd Filter for Extended Object

Consider an extended object located at [1;2;3]. The object's detections are uniformly distributed in x-, y-, and z-directions with dimensions of 1.2, 2.3, and 3.5, respectively. Generate 20 randomly distributed detections for the object using `objectDetection`.

```
detections = cell(20,1);  
location = [1;2;3];  
dimensions = [1.2;2.3;3.5];  
rng(2019);  
measurements = location + dimensions.*(-1 + 2*rand(3,20));  
for i = 1:20  
    detections{i} = objectDetection(0,measurements(:,i));  
end
```

Initialize a constant turn-rate gmphd filter using `initctgmphd`.

```
phd = initctgmphd(detections);
```

The initial state of the filter is same as the mean of the measurements.

```
state = phd.States
```

```
state = 7x1
```

```
1.1034  
0  
2.5597  
0  
0  
2.4861  
0
```

```
mean_measure = mean(measurements,2)
```

```
mean_measure = 3x1
```

```
1.1034  
2.5597  
2.4861
```

By default, the function sets the `HasExtent` property to true if the number of measurements is greater than 1.

```
extent = phd.HasExtent  
extent = logical  
1
```

## Input Arguments

### **detections — Object detections**

cell array of `objectDetection` objects

Object detections, specified as a cell array of `objectDetection` objects. You can create `detections` directly, or you can obtain `detections` from the outputs of sensor objects, such as `radarSensor`, `monostaticRadarSensor`, `irSensor`, and `sonarSensor`.

Example: `detection = objectDetection(0,[1;4.5;3], 'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **phd — gmphd filter**

`gmphd` object

Gaussian mixture PHD filter, returned as a `gmphd` object.

## Algorithms

- You can use `initctgmphd` as the `FilterInitializationFcn` property of `trackingSensorConfiguration`.
- When detections are provided as input, the function adds one component to the density, which reflects the mean of the detections. When the function is called without any inputs, a filter is initialized with no components in the density.
- The function uses the spread of measurements to specify the positional covariance.
- The function configures the process noise of the filter by assuming a unit acceleration standard deviation and a unit angular acceleration standard deviation.
- The function specifies a maximum of 500 components in the filter.
- The function sets the `HasExtent` property of the filter to `true` if the number of input detections are greater than one.

## Extended Capabilities

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

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

## See Also

`gmphd` | `initcagmphd` | `initcvgmphd` | `trackerPHD`

## Introduced in R2019b

# initcvgmphd

Create constant velocity `gmphd` filter

## Syntax

```
phd = initcvgmphd  
phd = initcvgmphd(detections)
```

## Description

`phd = initcvgmphd` initializes a constant velocity `gmphd` filter with zero components in the filter.

`phd = initcvgmphd(detections)` initializes a constant velocity `gmphd` filter based on information provided in object detections, `detections`. The function initializes a constant velocity state with the same convention as `constvel` and `cvmeas`,  $[x; v_x; y; v_y; z; v_z]$ .

---

**Note** This initialization function is not compatible with `trackerGNN`, `trackerJPDA`, and `trackerTOMHT` System objects.

---

## Examples

### Initialize Constant Velocity `gmphd` Filter for Point Target

Consider a point target located at  $[1; 2; 3]$ . Create a detection for the target using `objectDetection`.

```
detection = objectDetection(0,[1;2;3]);
```

Initialize a constant velocity `gmphd` filter using `initcvgmphd`.

```
phd = initcvgmphd(detection);
```

Display the initial state and the extent setup of the `gmphd` filter.

```
state = phd.States
```

```
state = 6×1
```

```
1  
0  
2  
0  
3  
0
```

```
extent = phd.HasExtent
```

```
extent = logical
0
```

### Initialize Constant Velocity gmphd Filter for Extended Object

Consider an extended object located at [1;2;3]. The object's detections are uniformly distributed in x-, y-, and z-directions with dimensions of 1.2, 2.3, and 3.5, respectively. Generate 20 randomly distributed detections for the object using `objectDetection`.

```
detections = cell(20,1);
location = [1;2;3];
dimensions = [1.2;2.3;3.5];
rng(2019);
measurements = location + dimensions.*(-1 + 2*rand(3,20));
for i = 1:20
    detections{i} = objectDetection(0,measurements(:,i));
end
```

Initialize a constant velocity gmphd filter using `initcvgmpfd`.

```
phd = initcvgmpfd(detections);
```

The initial state of the filter is same as the mean of the measurements.

```
state = phd.States
```

```
state = 6×1
```

```
1.1034
0
2.5597
0
2.4861
0
```

```
mean_measure = mean(measurements,2)
```

```
mean_measure = 3×1
```

```
1.1034
2.5597
2.4861
```

By default, the function sets the `HasExtent` property to true if the number of measurements is larger than 1.

```
extent = phd.HasExtent
extent = logical
1
```

## Input Arguments

### **detections — Object detections**

cell array of `objectDetection` objects

Object detections, specified as a cell array of `objectDetection` objects. You can create detections directly, or you can obtain detections from the outputs of sensor objects, such as `radarSensor`, `monostaticRadarSensor`, `irSensor`, and `sonarSensor`.

Example: `detection = objectDetection(0,[1;4.5;3], 'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **phd — gmphd filter**

`gmphd` object

Gaussian mixture PHD filter, returned as a `gmphd` object.

## Algorithms

- You can use `initcvgmpfd` as the `FilterInitializationFcn` property of `trackingSensorConfiguration`.
- When detections are provided as input, the function adds one component to the density, which reflects the mean of the detections. When the function is called without any inputs, a filter is initialized with no components in the density.
- The function uses the spread of measurements to specify the positional covariance.
- The function configures the process noise of the filter by assuming a unit acceleration standard deviation.
- The function specifies a maximum of 500 components in the filter.
- The function sets the `HasExtent` property of the filter to `true` if the number of input detections are greater than one.

## Extended Capabilities

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

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

## See Also

`gmphd` | `initcagmpfd` | `initctgmpfd` | `trackerPHD`

## Introduced in R2019b

# initctrectgmpfd

Create constant turn-rate rectangular target `gmphd` filter

## Syntax

```
phd = initctrectgmpfd
phd = initctrectgmpfd(detections)
```

## Description

`phd = initctrectgmpfd` initializes a constant turn-rate rectangular target `gmphd` filter with zero components in the filter.

`phd = initctrectgmpfd(detections)` initializes a constant turn-rate rectangular target `gmphd` filter based on information provided in object detections, `detections`. The function initializes a constant turn-rate rectangular state with the same convention as `ctrect` and `ctrectmeas`,  $[x; y; s; \theta; \omega; L; W]$ . See “Algorithms” on page 1-302 for the meaning of these variables.

---

**Note** This initialization function is not compatible with `trackerGNN`, `trackerJPDA`, and `trackerTOMHT` System objects.

---

## Examples

### Initialize `gmphd` for Rectangular Target

Load detections generated by a rectangular target and the corresponding truth.

```
load ('rectangularTargetDetections','detections','truthState');
```

Initialize the filter using detections.

```
phd = initctrectgmpfd(detections);
```

Display the estimated state and the truth state.

```
estState = phd.States
```

```
estState = 7×1
```

```
-0.0688
49.2233
0
0
0
3.3942
0.9871
```

```
truthState  
truthState = 7x1  
  
    0  
50.0000  
    0  
30.0000  
    0  
4.7000  
1.8000
```

## Input Arguments

### **detections — Object detections**

cell array of `objectDetection` objects

Object detections, specified as a cell array of `objectDetection` objects. You can create `detections` directly, or you can obtain `detections` from the outputs of sensor objects, such as `radarSensor`, `monostaticRadarSensor`, `irSensor`, and `sonarSensor`.

Example: `detection = objectDetection(0,[1;4.5;3], 'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

## Output Arguments

### **phd — gmphd filter**

`gmphd` object

Gaussian mixture PHD filter, returned as a `gmphd` object.

## Algorithms

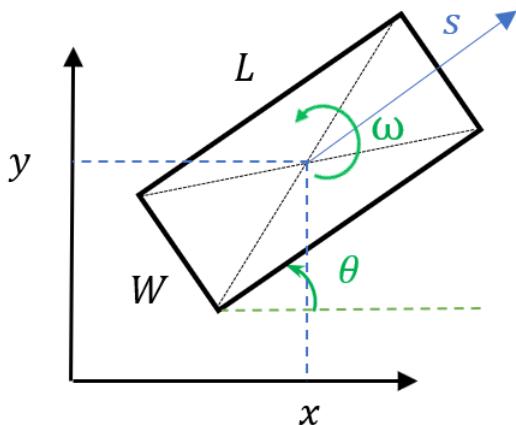
### Initialization Process

- You can use `initctrectgmphd` as the `FilterInitializationFcn` property of `trackingSensorConfiguration`.
- When detections are provided as input, the function adds one component to the density, which reflects the mean of the detections. When the function is called without any inputs, a filter is initialized with no components in the density.
- The function uses the spread of measurements to specify the length and width of the rectangle.
- The function configures the process noise of the filter by assuming a unit acceleration and a unit yaw-acceleration standard deviation.
- The function specifies a maximum of 500 components in the filter.
- The function configures the covariance of the state using a unit covariance in observed dimensions.

### Rectangular Target State

The rectangular target state contains  $[x; y; s;\theta; \omega; L; W]$ :

Variable	Meaning	Unit
$x$	Position of the rectangle center in $x$ direction	m
$y$	Position of the rectangle center in $y$ direction	m
$s$	Speed in the heading direction	m/s
$\theta$	Orientation angle of the rectangle with respect to $x$ direction	degree
$\omega$	Turn-rate	degree/s
$L$	Length of the rectangle	m
$W$	Width of the rectangle	m



## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[ctrect](#) | [ctrectcorners](#) | [ctrectjac](#) | [ctrectmeas](#) | [ctrectmeasjac](#) | [gmphd](#) | [initcagmphd](#) | [initctgmpfd](#) | [initctrectgmpfd](#) | [trackerPHD](#)

**Introduced in R2019b**

# initsingerekf

Singer acceleration trackingEKF initialization

## Syntax

```
filter = initsingerekf(detection)
```

## Description

`filter = initsingerekf(detection)` initializes a Singer acceleration extended Kalman filter (trackingEKF) based on the detection input.

## Examples

### Initialize Singer Acceleration Extended Kalman Filter in Rectangular Frame

For a rectangular frame, the Singer acceleration measurement function, `singermeas`, assumes a position measurement in 3-D space. Define a position measurement [1;3;0] that has measurement noise [1 0.2 0; 0.2 2 0; 0 0 1].

```
detection = objectDetection(0, [1;3;0], ...
    'MeasurementNoise', [1 0.2 0; 0.2 2 0; 0 0 1]);
```

Use the `initsingerekf` function to create a `trackingEKF` filter using the measurements defined above.

```
ekf = initsingerekf(detection);
```

Verify the values of the state and measurement noise.

```
filterState = ekf.State
```

```
filterState = 9x1
```

```
1  
0  
0  
3  
0  
0  
0  
0  
0
```

```
filterMeasureNoise = ekf.MeasurementNoise
```

```
filterMeasureNoise = 3x3
```

```
1.0000    0.2000        0  
0.2000    2.0000        0
```

0	0	1.0000
---	---	--------

## Input Arguments

### **detection — Object detection**

objectDetection object

Object detection, specified as an `objectDetection` object. You can specify the following fields for the `MeasurementParameters` property of the `objectDetection` object. When you do not specify a field, the default value is used.

Field	Description	Default Value
Frame	Frame used to report measurements, specified as one of these values: <ul style="list-style-type: none"> <li>• '<code>'rectangular'</code> — Detections are reported in rectangular coordinates.</li> <li>• '<code>'spherical'</code> — Detections are reported in spherical coordinates.</li> </ul>	<code>'rectangular'</code>
OriginPosition	Position offset of the origin of the frame relative to the parent frame, specified as an $[x \ y \ z]$ real-valued vector.	<code>[0 0 0]</code>
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, specified as a $[vx \ vy \ vz]$ real-valued vector.	<code>[0 0 0]</code>
Orientation	Frame rotation matrix, specified as a 3-by-3 real-valued orthonormal matrix.	<code>[1 0 0; 0 1 0; 0 0 1]</code>
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
HasRange	Logical scalar indicating if range is included in the measurement.	1

Field	Description	Default Value
HasVelocity	Logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if <code>HasVelocity</code> is <code>false</code> , the measurements are reported as <code>[x y z]</code> . If <code>HasVelocity</code> is <code>true</code> , measurements are reported as <code>[x y z vx vy vz]</code> .	0
IsParentToChild	Logical scalar indicating if <code>Orientation</code> performs a frame rotation from the parent coordinate frame to the child coordinate frame. When <code>IsParentToChild</code> is <code>false</code> , then <code>Orientation</code> performs a frame rotation from the child coordinate frame to the parent coordinate frame.	0

## Output Arguments

**filter — Extended Kalman filter**  
**trackingEKF** object

Extended Kalman filter, returned as a `trackingEKF` object.

## Algorithms

- You can use the `initsingerekf` function as the `FilterInitializationFcn` property of `trackingEKF`.
- When creating the Kalman filter, the function configures the process noise assuming a target maneuver time constant,  $\tau = 20\text{s}$  and a unit target maneuver standard deviation,  $\sigma = 1 \text{ m/s}^2$ . The function uses the `singerProcessNoise` function.
- The Singer process noise assumes an invariant time step and additive process noise.

## References

- [1] Singer, Robert A. "Estimating optimal tracking filter performance for manned maneuvering targets." *IEEE Transactions on Aerospace and Electronic Systems* 4 (1970): 473-483.
- [2] Blackman, Samuel S., and Robert Popoli. "Design and analysis of modern tracking systems." (1999).
- [3] Li, X. Rong, and Vesselin P. Jilkov. "Survey of maneuvering target tracking: dynamic models." *Signal and Data Processing of Small Targets 2000*, vol. 4048, pp. 212-235. International Society for Optics and Photonics, 2000.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

- The function returns a filter that uses anonymous functions to define the `StateTransitionFcn` and `StateTransitionCovarianceFcn` properties. As a result, the value of the target maneuver constant ( $\tau = 20\text{s}$ ) cannot be modified after construction. To create a Singer-based filter with a different value of  $\tau$ , you must write your own filter initialization function similar as the `initsingerekf` function.

### See Also

`singer` | `singerProcessNoise` | `singerjac` | `singermeas` | `singermeasjac`

Introduced in R2020b

## ctrectcorners

Corner measurements of constant turn-rate rectangular target

### Syntax

```
zCorners = ctrectcorners(states)
zCorners = ctrectcorners(states,sensorParameters)
```

### Description

`zCorners = ctrectcorners(states)` returns the positions of the corners for constant turn-rate rectangular targets in a rectangular frame.

`zCorners = ctrectcorners(states,sensorParameters)` specifies the parameters of the sensor that measures the corners of rectangular targets.

### Examples

#### Position of Corners in Sensor Reference Frame

Define sensor reference frame by specifying the `sensorParameters` input.

```
sensorPosition = [-5;10;0];
sensorOrientation = rotmat(quaternion([30 0 0],'eulerd','ZYX','frame'),'frame');
sensorParams = struct('Frame','Rectangular',...
    'OriginPosition',sensorPosition,...
    'Orientation',sensorOrientation);
```

Define the constant turn-rate state for the rectangle target.

```
state = [10;5;1.6;30;0.5;4.7;1.8];
```

Compute corner positions in sensor reference frame.

```
corners = ctrectcorners(state,sensorParams);
```

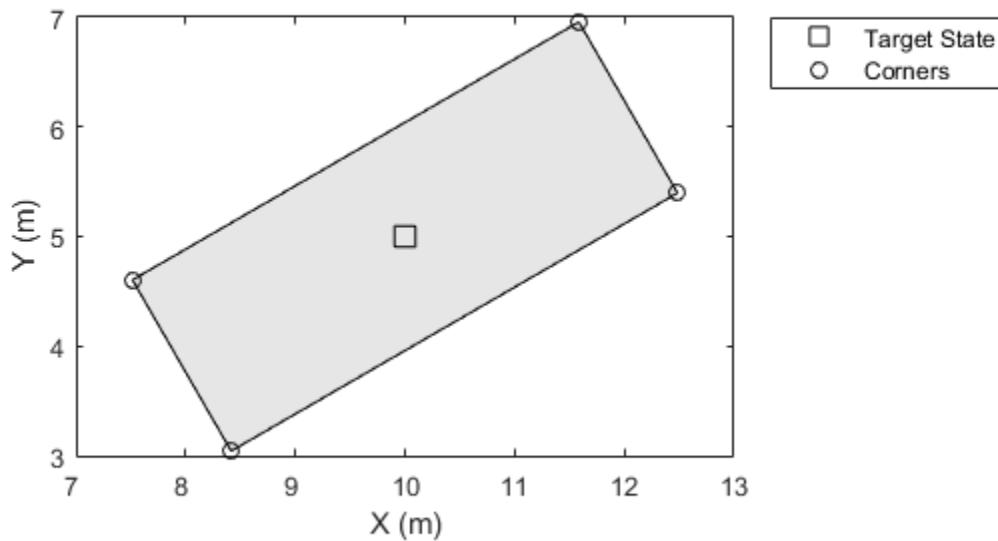
Set up visualization environment using `theaterPlot`.

```
% Create a theater plot.
tp = theaterPlot;
% Plot the state using a track plotter.
statePlotter = trackPlotter(tp,'DisplayName','Target State');
% Plot the corners using a detection plotter.
cornerPlotter = detectionPlotter(tp,'DisplayName','Corners');
```

Compute inputs and plot.

```
targetPos = [state(1) state(2) 0];
targetOrientation = rotmat(quaternion([state(4) 0 0],'eulerd','ZYX','frame'),'frame');
targetDims = struct('Length',state(6),...
    'Width',state(7),...)
```

```
'Height',5,...  
'OriginOffset',[0 0 0]);  
  
cornerPosGlobal = sensorOrientation*corners(:, :) + sensorPosition;  
statePlotter.plotTrack(targetPos,targetDims,targetOrientation);  
cornerPlotter.plotDetection(cornerPosGlobal');
```



## Input Arguments

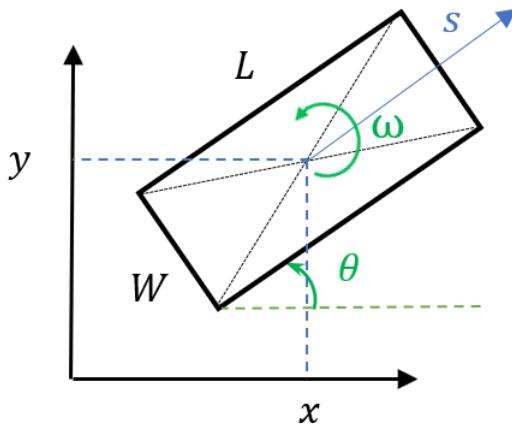
### **states — Current rectangular target states**

7-by- $N$  real-valued matrix

Current rectangular target states, specified as a 7-by- $N$  real-valued matrix, where  $N$  is the number of states. The seven dimensional rectangular state is defined as  $[x; y; s; \theta; \omega; L; W]$ . The meaning of these variables and their units are:

Variable	Meaning	Unit
$x$	Position of the rectangle center in $x$ direction	m
$y$	Position of the rectangle center in $y$ direction	m
$s$	Speed in the heading direction	m/s

$\theta$	Orientation angle of the rectangle with respect to $x$ direction	degree
$\omega$	Turn-rate	degree/s
$L$	Length of the rectangle	m
$W$	Width of the rectangle	m



Example: [1;2;2;30;1;4.7;1.8]

Data Types: single | double

#### **sensorParameters — Parameters for sensor transform function**

structure | array of structures

Parameters for the sensor transform function, returned as a structure or an array of structures. If you only need to transform the state once, specify it as a structure. If you need to transform the state  $n$  times, specify it as an  $n$ -by-1 array of structures. For example, to transform a state from the scenario frame to the sensor frame, you usually need to first transform the state from the scenario rectangular frame to the platform rectangular frame, and then transform the state from the platform rectangular frame to the sensor spherical frame.

The fields of the structure are:

Field	Description
Frame	Child coordinate frame type, specified as 'Rectangular' or 'Spherical'.
OriginPosition	Child frame origin position expressed in the parent frame, specified as a 3-by-1 vector.
OriginVelocity	Child frame origin velocity expressed in the parent frame, specified as a 3-by-1 vector.

<b>Orientation</b>	Relative orientation between frames, specified as a 3-by-3 rotation matrix. If the <b>IsParentToChild</b> property is set to <b>false</b> , then specify <b>Orientation</b> as the rotation from the child frame to the parent frame. If the <b>IsParentToChild</b> property is set to <b>true</b> , then specify <b>Orientation</b> as the rotation from the parent frame to the child frame.
<b>IsParentToChild</b>	Flag to indicate the direction of rotation between parent and child frame, specified as <b>true</b> or <b>false</b> . The default is <b>false</b> . See description of the <b>Orientation</b> field for details.
<b>HasAzimuth</b>	Indicates whether outputs contain azimuth components, specified as <b>true</b> or <b>false</b> .
<b>HasElevation</b>	Indicates whether outputs contain elevation components, specified as <b>true</b> or <b>false</b> .
<b>HasRange</b>	Indicates whether outputs contain range components, specified as <b>true</b> or <b>false</b> .
<b>HasVelocity</b>	Indicates whether outputs contain velocity components, specified as <b>true</b> or <b>false</b> .

Note that here the scenario frame is the parent frame of the platform frame, and the platform frame is the parent frame of the sensor frame.

When **frame** is '**Rectangular**', **HasVelocity** determines if the measurement is returned in the form of  $[x; y; z; v_x; v_y; v_z]$  or  $[x; y; z]$ .

When **frame** is '**spherical**', the returned measurements are in the order of [azimuth, elevation, range, range-rate]. The elements of the returned measurements are determined by:

- **HasAzimuth** — Determines if output contains azimuth measurement.
- **HasElevation** — Determines if output contains elevation measurement.
- **HasRange** — Determines if output contains range measurement.
- **HasVelocity** — Determines if output contains range-rate measurement on the condition that **HasRange** is '**true**'. If **HasRange** is '**false**', the returned measurement does not contain range-rate (even though **HasVelocity** is '**true**').

Data Types: **struct**

## Output Arguments

### **zCorners — States of corners**

real-valued  $M$ -by- $N$ -by-4 array.

States of corners, returned as a real-valued  $M$ -by- $N$ -by-4 array. Each page (an  $M$ -by- $N$  matrix) of the array corresponds to one corner for all the states given in the **states** input.  $N$  is the number of states.  $M$  is the dimension of output specified by the **sensorParameters** input. If unspecified, the default value of  $M$  is three, which corresponds to 3-D Cartesian position coordinates.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`ctrect` | `ctrectjac` | `ctrectmeas` | `ctrectmeasjac` | `gmphd` | `initctrectgmphd` | `trackerPHD`

Introduced in R2018b

# switchimm

Model conversion function for `trackingIMM` object

## Syntax

```
x = switchimm(modelType1,x1,modelType2)
x = switchimm(___,x2)
```

## Description

`x = switchimm(modelType1,x1,modelType2)` converts the `State` or `StateCovariance` properties of the `trackingIMM` object from `modelType1` state definition to `modelType2` state definition.

- `modelType1` -- Specifies the string name of the current motion model.
- `x1` -- Specifies `State` or `StateCovariance` corresponding to `modelType1`.
- `modelType2` -- Specifies the string name of the motion model to which `x1` needs to be converted.

`x = switchimm(___,x2)` additionally lets you specify the size and type of the output. When not specified, `x` has the same data type and dimensionality as `x1`.

`x2` specifies `State` or `StateCovariance` corresponding to `modelType2`.

## Examples

### Convert State from Constant Acceleration to Constant Velocity

Convert state from constant acceleration model to constant velocity model using the `switchimm` function.

#### Initialization

Set the current model to '`constacc`' and the destination model to '`constvel`'. The variable `x1` defines the state in the current model.

```
modelType1 = 'constacc';
modelType2 = 'constvel';
x1 = single([1;2;3;4;5;6]);
```

#### Conversion

The `switchimm` function converts the 2-D constant acceleration state input to a 2-D constant velocity state output. The output has the same dimensionality and data type as the input `x1`.

```
x = switchimm(modelType1,x1,modelType2)
x = 4x1 single column vector
```

1

2

4  
5

## Convert State from Constant Acceleration to Constant Turn

Convert state from constant acceleration model to constant turn model using the `switchimm` function. Specify `x2` as an input parameter.

### Initialization

Set the current model to '`constacc`' and the destination model to '`constturn`'. The variable `x1` defines the state in the current model. The size and data type of the output is determined by the optional input `x2`.

```
modelType1 = 'constacc';
modelType2 = 'constturn';
x1 = [1;2;3;4;5;6];
x2 = [0;0;0;0;0;0];
```

### Conversion

The `switchimm` function converts the 2-D constant acceleration state input to a 3-D constant turn model state output. The output has the same size and data type as the input `x2`.

```
x = switchimm(modelType1,x1,modelType2,x2)
x = 7x1
1
2
4
5
0
0
0
```

## Input Arguments

### **modelType1 — Current motion model**

`'constvel'` | `'constacc'` | `'constturn'`

Current motion model, specified as:

- '`constvel`' -- Constant velocity motion model.
- '`constacc`' -- Constant-acceleration motion model.
- '`constturn`' -- Constant turn-rate motion model.

### **x1 — State or state covariance of current model**

vector | matrix

State vector or state covariance matrix corresponding to the current model in `modelType1`, specified as an  $L$ -by-1 real vector or an  $L$ -by- $L$  real matrix.

The size of the state vector must fit the motion model. For example, if the `modelType` is '`constvel`', the state vector must be of size 2, 4, or 6. Similarly, if the `modelType` is '`constacc`', the state vector must be of size 3, 6, or 9. If the `modelType` is '`constturn`', the state vector must be of size 5, 7, 10, 15, 14, or 21. The relationship between model type, state size, and the space dimension is given by the following table:

<b>modelType1</b>	<b>Supported Space Dimension</b>	<b>State size</b>
' <code>constvel</code> '	1-D, 2-D, 3-D	$2 \times$ Space dimension
' <code>constacc</code> '	1-D, 2-D, 3-D	$3 \times$ Space dimension
' <code>constturn</code> '	2-D and 3-D	5 for 2-D space and 7 for 3-D space

The '`constturn`' model type supports only 2-D and 3-D spaces, since a turn cannot be made in 1-D space. If the space dimension is computed to be 1-D, that is, the state size equals 5 or 7, the function treats the output dimension as 2 and the values corresponding to the second dimension are set to 0. For example, run the following in the MATLAB command prompt:

```
switchimm('constvel',rand(2,1),'constturn')
```

Data Types: `single` | `double`

#### **modelType2 — Motion model to which x1 needs to be converted**

`'constvel'` | `'constacc'` | `'constturn'`

Motion model to which `x1` needs to be converted, specified as:

- '`constvel`' -- Constant velocity motion model.
- '`constacc`' -- Constant-acceleration motion model.
- '`constturn`' -- Constant turn-rate motion model.

#### **x2 — Specify size and type of output state or state covariance**

`vector` | `matrix`

The optional input `x2` has the same size and data type as the output state vector or the state covariance matrix, `x`. The variable `x2` does not contain the actual output state information, but only holds the size and the data type of the output state. For example, when `x2` is set to `[0;0;0;0;0;0]`, the function determines the output state vector to be a vector of size 7 with a data type of double.

The size of the state vector must fit the motion model. For example, if the `modelType` is '`constvel`', the state vector must be of size 2, 4, or 6. Similarly, if the `modelType` is '`constacc`', the state vector must be of size 3, 6, or 9. The relationship between model type, state size, and the space dimension is given by the following table:

<b>modelType1</b>	<b>Supported Space Dimension</b>	<b>State size</b>
' <code>constvel</code> '	1-D, 2-D, 3-D	$2 \times$ Space dimension
' <code>constacc</code> '	1-D, 2-D, 3-D	$3 \times$ Space dimension
' <code>constturn</code> '	2-D and 3-D	5 for 2-D space and 7 for 3-D space

Example: `[0;0;0;0;0;0]`

Data Types: `single` | `double`

## Output Arguments

**x — State or state covariance corresponding to modelType2**

vector | matrix

State vector or state covariance matrix, corresponding to the motion model specified in `modelType2`.

The relationship between model type, state size, and the space dimension is given by the following table:

modelType1	Supported Space Dimension	State size
'constvel'	1-D, 2-D, 3-D	2 × Space dimension
'constacc'	1-D, 2-D, 3-D	3 × Space dimension
'constturn'	2-D and 3-D	5 for 2-D space and 7 for 3-D space

### If x2 is not specified:

Given `modelType1` and `x1`, the function determines the input state dimension based on the relationship specified in the table. For example, if `modelType1` is 'constvel', and `x1` is a 4-by-1 vector, the input state dimension is given by 4/2, which equals 2.

If `modelType1` is 'constacc' and `x1` is a 6-by-1 vector, the input state dimension is given by 6/3, which equals 2.

In this case when `x2` is not specified, the output `x` has the same data type as `x1` and the dimension is calculated using `modelType1` and `x1`.

### If x2 is specified:

The function calculates the output space dimension using `modelType2` and `x2`. For example, if `modelType2` is 'constacc' and `x2` is a 6-by-1 vector, the output state dimension is given by 6/3, which equals 2.

The output `x` has the same data type and dimensionality as `x2`.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Objects

`trackingIMM`

### Functions

`constacc` | `constturn` | `constvel` | `initcvmscekf`

### Introduced in R2018b

# initcvmscekf

Constant velocity trackingMSCEKF initialization

## Syntax

```
msckf = initcvmscekf(detection)
msckf = initcvmscekf(detection,rangeEstimation)
```

## Description

`msckf = initcvmscekf(detection)` initializes a `trackingMSCEKF` class (extended Kalman filter for tracking in modified spherical coordinates) based on information provided in an `objectDetection` object, `detection`. The function assumes a target range of  $3e^4$  units and a range-covariance of  $1e^{10}$  units<sup>2</sup>.

The `trackingMSCEKF` object can be used with trackers for tracking targets with angle-only measurements from a single observer.

`msckf = initcvmscekf(detection,rangeEstimation)` allows specifying the range information to the filter. The `rangeEstimation` variable is a two-element vector, where the first element specifies the range of the target, and the second element specifies the standard deviation in range.

## Examples

### Initialize a `trackingMSCEKF` Object Using Angle-Only Detection

Create an angle-only detection.

```
detection = objectDetection(0,[30;20], 'MeasurementParameters', ...
    struct('Frame','Spherical','HasRange',false));
```

Use `initcvmscekf` to create a `trackingMSCEKF` filter initialized using the angle-only detection.

```
filter = initcvmscekf(detection)

filter =
  trackingMSCEKF with properties:

    State: [6x1 double]
    StateCovariance: [6x6 double]

    StateTransitionFcn: @constvelmsc
    StateTransitionJacobianFcn: @constvelmscjac
    ProcessNoise: [3x3 double]
    HasAdditiveProcessNoise: 0
    ObserverInput: [3x1 double]

    MeasurementFcn: @cvmeasmse
    MeasurementJacobianFcn: @cvmeasmsejac
```

```
MeasurementNoise: [2x2 double]
HasAdditiveMeasurementNoise: 1
```

### Initialize trackingMSCEKF Object with Detection from Rotating Sensor

Create measurement parameters for subsequent rotation.

```
measParamSensorToPlat = struct('Frame','Spherical','HasRange',false,...
'Orientation',rotmat(quaternion([0 0 30],'rotvecd'),'frame'))  
  
measParamSensorToPlat = struct with fields:
    Frame: 'Spherical'
    HasRange: 0
    Orientation: [3x3 double]  
  
measParamPlatToScenario = struct('Frame','Rectangular','HasRange',false,...
'Orientation',rotmat(quaternion([30 0 0],'rotvecd'),'frame'))  
  
measParamPlatToScenario = struct with fields:
    Frame: 'Rectangular'
    HasRange: 0
    Orientation: [3x3 double]  
  
measParam = [measParamSensorToPlat;measParamPlatToScenario];
detection = objectDetection(0,[30;20],'MeasurementParameters',measParam);
```

Initialize a filter.

```
filter = initcvmscekf(detection);
```

Check that filter's measurement is same as detection.

```
cvmeasmSC(filter.State,measParam)
```

```
ans = 2x1
```

```
30.0000
20.0000
```

### Track a Constant Velocity Target Using trackerGNN

Consider a scenario when the target is moving at a constant velocity along and the observer is moving at a constant acceleration. Define target's initial state using a constant velocity model.

```
tgtState = [2000;-3;500;-5;0;0];
```

Define observer's initial state using a constant acceleration model.

```
observerState = [0;2;0;490;-10;0.2;0;0;0];
```

Create a `trackerGNN` object to use with `initcvmscekf` with some prior information about range and range-covariance.

```
range = 1000;
rangeStdDev = 1e3;
rangeEstimate = [range rangeStdDev];
tracker = trackerGNN('FilterInitializationFcn',@(det)initcvmscekf(det,rangeEstimate));
```

Simulate synthetic data by using measurement models. Get `az` and `el` information using the `cvmeas` function.

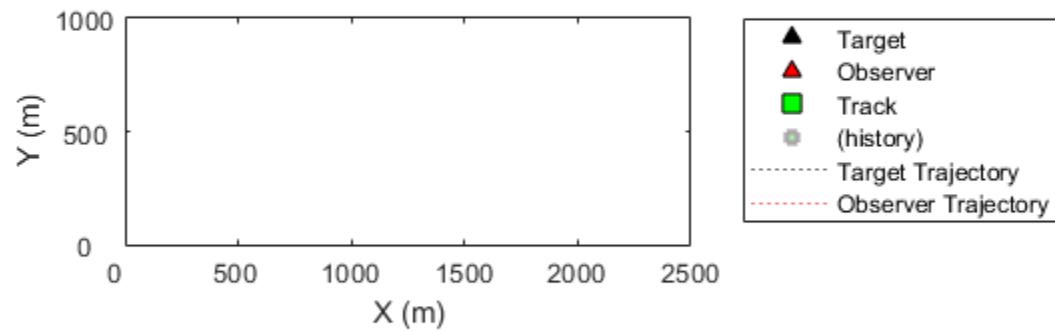
```
syntheticParams = struct('Frame','Spherical','HasRange',false,...
    'OriginPosition',observerState(1:3:end));
meas = cvmeas(tgtState,syntheticParams);
```

Create an angle-only objectDetection to simulate synthetic detection.

```
detection = objectDetection(0,meas,'MeasurementParameters',...
    struct('Frame','Spherical','HasRange',false),'MeasurementNoise',0.033*eye(2));
```

Create `trackPlotter` and `platformPlotter` to visualize the scenario.

```
tp = theaterPlot('XLimits',[0 2500], 'YLimits',[0 1000]);
targetPlotter = platformPlotter(tp,'DisplayName','Target','MarkerFaceColor','k');
observerPlotter = platformPlotter(tp,'DisplayName','Observer','MarkerFaceColor','r');
trkPlotter = trackPlotter(tp,'DisplayName','Track','MarkerFaceColor','g','HistoryDepth',50);
tgtTrajPlotter = trajectoryPlotter(tp,'DisplayName','Target Trajectory','Color','k');
obsTrajPlotter = trajectoryPlotter(tp,'DisplayName','Observer Trajectory','Color','r');
```

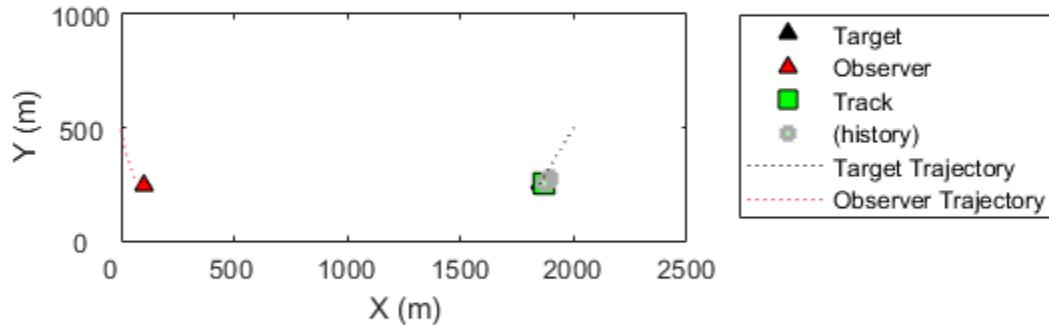


Run the tracker.

```
time = 0; dT = 0.1;
tgtPoses = [];
obsPoses = [];
while time < 50
    [confTracks,tentTracks,allTracks] = tracker(detection,time);
    for i = 1:numel(allTracks)
        setTrackFilterProperties(tracker,allTracks(i).TrackID,'ObserverInput',observerState(3:3:end));
    end

    % Update synthetic detection.
    observerState = constacc(observerState,dT);
    tgtState = constvel(tgtState,dT);
    syntheticParams.OriginPosition = observerState(1:3:end);
    detection.Measurement = cvmeas(tgtState,syntheticParams);
    time = time + dT;
    detection.Time = time;

    % Update plots
    tgtPoses = [tgtPoses;tgtState(1:2:end)']; %#ok
    obsPoses = [obsPoses;observerState(1:3:end)']; %#ok
    targetPlotter.plotPlatform(tgtState(1:2:end)');
    observerPlotter.plotPlatform(observerState(1:3:end)');
    tgtTrajPlotter.plotTrajectory({tgtPoses});
    obsTrajPlotter.plotTrajectory({obsPoses});
    % Plot the first track as there are no false alarms, this should be
    % the target.
    % Get positions from the MSC state of the track.
    cartState = cvmeasmc(allTracks(i).State,'rectangular') + observerState(1:3:end);
    trkPlotter.plotTrack(cartState');
end
```



## 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])`

**rangeEstimation — Range information**  
two-element vector

Range information, specified as a two-element vector, where the first element specifies the range of the target, and the second element specifies the standard deviation in range.

Data Types: `single` | `double`

## Output Arguments

**mscekf — Constant velocity tracking extended Kalman filter in MSC frame**  
trackingMSCEKF object

Constant velocity tracking extended Kalman filter in an MSC frame, returned as a `trackingMSCEKF` object.

## Algorithms

- The function configures the filter with process noise assuming a unit target acceleration standard deviation.
- The function configures the covariance of the state in an MSC frame by using a linear transformation of covariance in a Cartesian frame.
- You can use this function as the `FilterInitializationFcn` property of `trackerTOMHT` and `trackerGNN` System objects.
- The function initializes the `ObserverInput` of the `trackingMSCEKF` class with zero observer acceleration in all directions. You must use the `setTrackFilterProperties` function of the trackers to update the `ObserverInput`.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

`constvelmsc` | `constvelmscjac` | `cvmeasmsc` | `cvmeasmscjac`

### Objects

`objectDetection` | `trackingMSCEKF`

### Introduced in R2018b

# initapekf

Constant velocity angle-parameterized EKF initialization

## Syntax

```
filter = initapekf(detection)
filter = initapekf(detection,numFilters)
filter = initapekf(detection,numFilters,angleLimits)
```

## Description

`filter = initapekf(detection)` configures the filter with 10 extended Kalman filters (EKFs). The function configures the process noise with unit standard deviation in acceleration.

The angle-parameterized extended Kalman filter (APEKF) is a Gaussian-sum filter (`trackingGSF`) with multiple EKFs, each initialized at an estimated angular position of the target. Angle-parametrization is a commonly used technique to initialize a filter from a range-only detection.

`filter = initapekf(detection,numFilters)` specifies the number of EKFs in the filter.

`filter = initapekf(detection,numFilters,angleLimits)` specifies the limits on angular position of the target.

## Examples

### Initialize APEKF from Range Only Detection and Visualize Filter

The APEKF is a special type of filter that can be initialized using range-only measurements. When the '`Frame`' is set to '`spherical`', the detection has [azimuth elevation range range-rate] measurements. Specify the measurement parameters appropriately to define a range-only measurement.

```
measParam = struct('Frame','Spherical','HasAzimuth',false,'HasElevation',false,'HasVelocity',false)
```

The `objectDetection` class defines an interface to the range-only detection measured by the sensor. The `MeasurementParameters` field of `objectDetection` carries information about what the sensor is measuring.

```
detection = objectDetection(0,100,'MeasurementNoise',100,'MeasurementParameters',measParam)

detection =
  objectDetection with properties:

    Time: 0
    Measurement: 100
    MeasurementNoise: 100
    SensorIndex: 1
    ObjectClassID: 0
    MeasurementParameters: [1x1 struct]
    ObjectAttributes: {}
```

The `initapekf` function uses the range-only detection to initialize the APEKF.

```
apekf = initapekf(detection) %#ok  
apekf =  
    trackingGSF with properties:  
  
        State: [6x1 double]  
        StateCovariance: [6x6 double]  
  
        TrackingFilters: {10x1 cell}  
        ModelProbabilities: [10x1 double]  
  
        MeasurementNoise: 100
```

You can also initialize the APEKF with 10 filters and to operate within the angular limits of [-30 30] degrees.

```
angleLimits = [-30 30];  
numFilters = 10;  
apekf = initapekf(detection, numFilters, angleLimits)  
  
apekf =  
    trackingGSF with properties:  
  
        State: [6x1 double]  
        StateCovariance: [6x6 double]  
  
        TrackingFilters: {10x1 cell}  
        ModelProbabilities: [10x1 double]  
  
        MeasurementNoise: 100
```

You can also specify the `initapekf` function as a `FilterInitializationFcn` to the `trackerGNN` object.

```
funcHandle = @(detection)initapekf(detection,numFilters,angleLimits)  
funcHandle = function_handle with value:  
    @(detection)initapekf(detection,numFilters,angleLimits)  
  
tracker = trackerGNN('FilterInitializationFcn',funcHandle);
```

Visualize the filter.

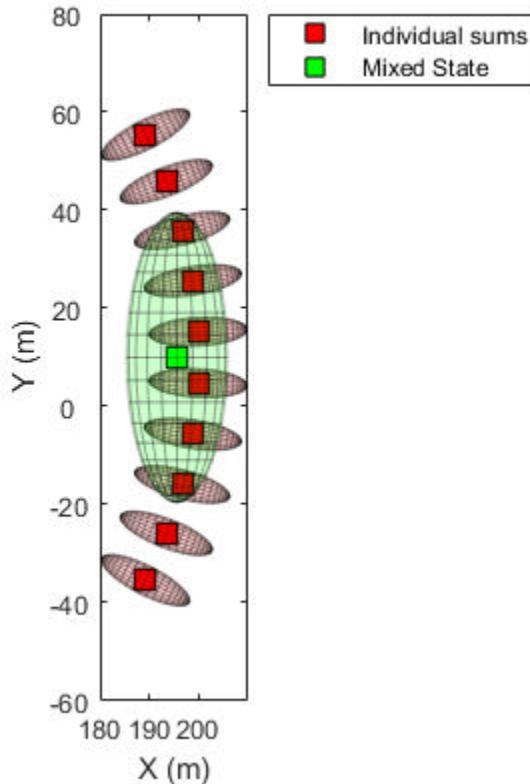
```
tp = theaterPlot;  
componentPlot = trackPlotter(tp,'DisplayName','Individual sums','MarkerFaceColor','r');  
sumPlot = trackPlotter(tp,'DisplayName','Mixed State','MarkerFaceColor','g');  
  
indFilters = apekf.TrackingFilters;  
pos = zeros(numFilters,3);  
cov = zeros(3,3,numFilters);  
for i = 1:numFilters  
    pos(i,:) = indFilters{i}.State(1:2:end);  
    cov(1:3,1:3,i) = indFilters{i}.StateCovariance(1:2:end,1:2:end);  
end
```

```

componentPlot.plotTrack(pos,cov);

mixedPos = apekf.State(1:2:end)';
mixedPosCov = apekf.StateCovariance(1:2:end,1:2:end);
sumPlot.plotTrack(mixedPos,mixedPosCov);

```



### Initialize APEKF from Azimuth and Range Detection and Visualize Filter

Create an angle-parameterized EKF from an [az r] detection.

```
measParam = struct('Frame','Spherical','HasAzimuth',true,'HasElevation',false,'HasVelocity',false)
```

The `objectDetection` class defines an interface to the range-only detection measured by the sensor. The `MeasurementParameters` field of `objectDetection` carries information about what the sensor is measuring.

```
det = objectDetection(0,[30;100], 'MeasurementParameters',measParam, 'MeasurementNoise',10);
```

The `initapekf` function parameterizes the `apekf` filter on the elevation measurement.

```

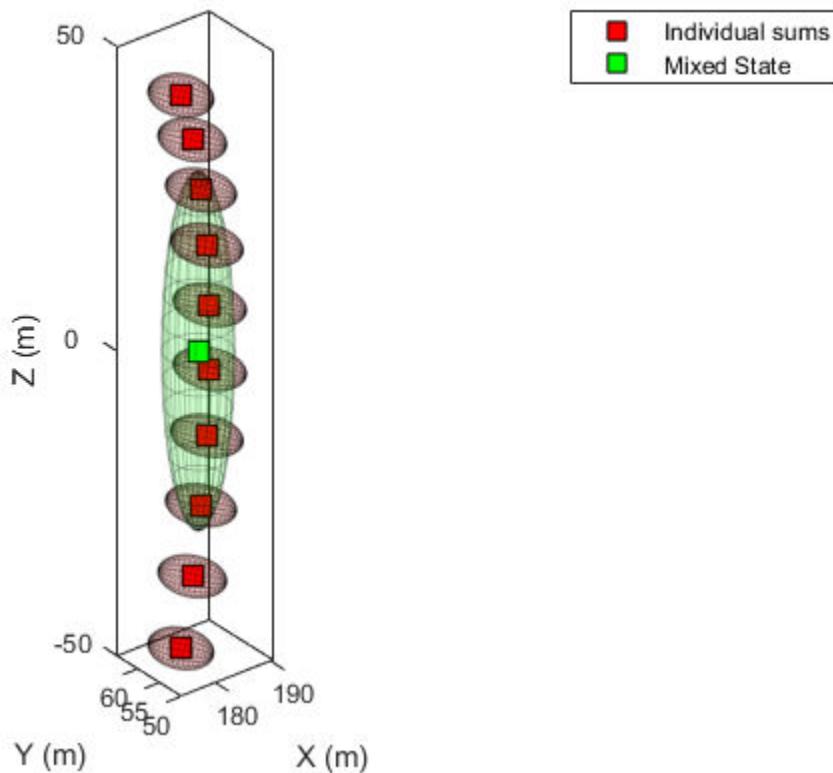
numFilters = 10;
apekf = initapekf(det,numFilters,[-30 30]);
indFilters = apekf.TrackingFilters;
pos = zeros(numFilters,3);
cov = zeros(3,3,numFilters);

```

```
for i = 1:numFilters
    pos(i,:) = indFilters{i}.State(1:2:end);
    cov(1:3,1:3,i) = indFilters{i}.StateCovariance(1:2:end,1:2:end);
end
```

Visualize the filter.

```
tp = theaterPlot;
componentPlot = trackPlotter(tp,'DisplayName','Individual sums','MarkerFaceColor','r');
sumPlot = trackPlotter(tp,'DisplayName','Mixed State','MarkerFaceColor','g');
componentPlot.plotTrack(pos,cov);
mixedPos = apekf.State(1:2:end)';
mixedPosCov = apekf.StateCovariance(1:2:end,1:2:end);
sumPlot.plotTrack(mixedPos,mixedPosCov);
view(3);
```



## 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])`

**numFilters – Number of EKFs**

10 (default) | positive integer

Number of EKFs each initialized at an estimated angular position of the target, specified as a positive integer. When not specified, the default number of EKFs is 10.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**angleLimits – Angular limits of target**

two-element vector

Angular limits of the target, specified as a two-element vector. The two elements in the vector represent the lower and upper limits of the target angular position.

When the function detects:

- Range measurements -- Default angular limits are [-180 180].
- Azimuth and range measurements -- Default angular limits are [-90 90].

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Output Arguments

**filter – Constant velocity angle-parameterized EKF**

trackingGSF object

Constant velocity angle-parameterized extended Kalman filter (EKF), returned as a trackingGSF object.

## Algorithms

The function can support the following types of measurements in the detection.

- Range measurements -- Parameterization is done on the azimuth of the target, and the angular limits are [-180 180] by default.
- Azimuth and range measurements -- Parameterization is done on the elevation of the target, and the angular limits are [-90 90] by default.

## References

[1] Ristic, Branko, Sanjeev Arulampalam, and James McCarthy. "Target motion analysis using range-only measurements: algorithms, performance and application to ISAR data." Signal Processing 82, no. 2 (2002): 273-296.

## Extended Capabilities

**C/C++ Code Generation**

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

## See Also

### Functions

`initcvekf`

### Objects

`objectDetection` | `trackingEKF` | `trackingGSF`

**Introduced in R2018b**

# initrpekf

Constant velocity range-parameterized EKF initialization

## Syntax

```
filter = initrpekf(detection)
filter = initrpekf(detection,numFilters)
filter = initrpekf(detection,numFilters,rangeLimits)
```

## Description

`filter = initrpekf(detection)` configures the filter with 6 extended Kalman filters (EKFs), and the target range is assumed to be within 1e3 and 1e5 scenario units. The function configures the process noise with unit standard deviation in acceleration.

The range-parameterized extended Kalman filter (RPEKF) is a Gaussian-sum filter (`trackingGSF`) with multiple EKFs, each initialized at an estimated range of the target. Range-parameterization is a commonly used technique to initialize a filter from an angle-only detection.

`filter = initrpekf(detection,numFilters)` specifies the number of EKFs in the filter.

`filter = initrpekf(detection,numFilters,rangeLimits)` specifies the range limits of the target.

## Examples

### Initialize RPEKF from Angle-only Detection and Visualize Filter

The RPEKF is a special type of filter that can be initialized using angle-only measurements, that is, azimuth and/or elevation. When the 'Frame' is set to 'spherical' and 'HasRange' is set to 'false', the detection has [azimuth elevation] measurements. Specify the measurement parameters appropriately to define an angle-only measurement with no range information.

```
measParam = struct('Frame','spherical','HasRange',false,'OriginPosition',[100;10;0]);
```

The `objectDetection` class defines an interface to the angle-only detection measured by the sensor. The `MeasurementParameters` field of `objectDetection` carries information about what the sensor is measuring.

```
detection = objectDetection(0,[30;30], 'MeasurementParameters',measParam, 'MeasurementNoise', 2*eye(6));
```

The `initrpekf` function uses the angle-only detection to initialize the RPEKF.

```
rpekf = initrpekf(detection) %#ok
rpekf =
  trackingGSF with properties:
    State: [6x1 double]
    StateCovariance: [6x6 double]
```

```
    TrackingFilters: {6x1 cell}
    ModelProbabilities: [6x1 double]

    MeasurementNoise: [2x2 double]
```

You can also initialize the RPEKF with 10 filters and to operate within the range limits of [1000, 10,000] scenario units.

```
rangeLimits = [1000 10000];
numFilters = 10;
rpekf = initrpekf(detection, numFilters, rangeLimits)

rpekf =
    trackingGSF with properties:

        State: [6x1 double]
        StateCovariance: [6x6 double]

        TrackingFilters: {10x1 cell}
        ModelProbabilities: [10x1 double]

        MeasurementNoise: [2x2 double]
```

You can also specify the `initrpekf` function as a `FilterInitializationFcn` to the `trackerGNN` object.

```
funcHandle = @(detection)initrpekf(detection,numFilters,rangeLimits)

funcHandle = function_handle with value:
    @(detection)initrpekf(detection,numFilters,rangeLimits)

tracker = trackerGNN('FilterInitializationFcn',funcHandle)

tracker =
    trackerGNN with properties:

        TrackerIndex: 0
        FilterInitializationFcn: [function_handle]
            Assignment: 'MatchPairs'
            AssignmentThreshold: [30 Inf]
            MaxNumTracks: 100
            MaxNumSensors: 20

            TrackLogic: 'History'
            ConfirmationThreshold: [2 3]
            DeletionThreshold: [5 5]

            HasCostMatrixInput: false
            HasDetectableTrackIDsInput: false
            StateParameters: [1x1 struct]

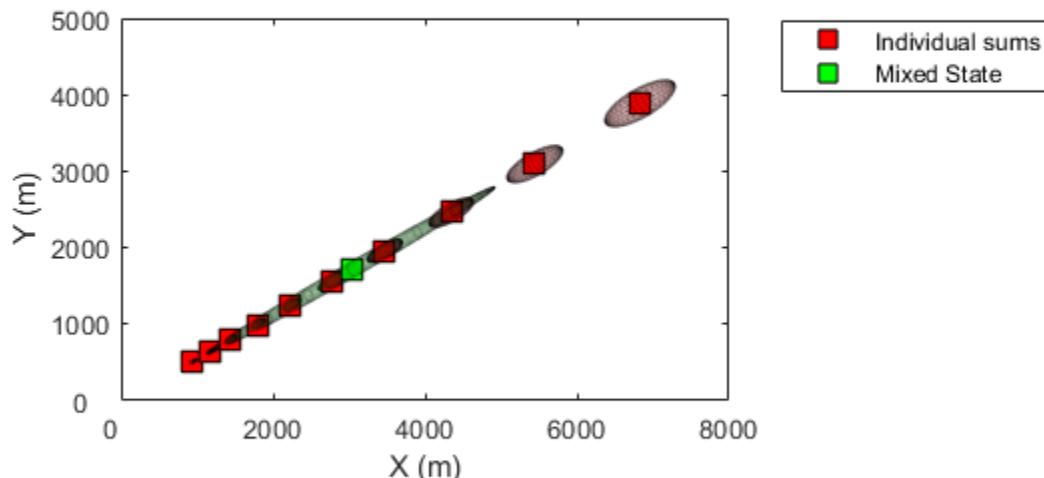
            NumTracks: 0
            NumConfirmedTracks: 0
```

Visualize the filter.

```
tp = theaterPlot;
componentPlot = trackPlotter(tp,'DisplayName','Individual sums','MarkerFaceColor','r');
sumPlot = trackPlotter(tp,'DisplayName','Mixed State','MarkerFaceColor','g');

indFilters = rpekf.TrackingFilters;
pos = zeros(numFilters,3);
cov = zeros(3,3,numFilters);
for i = 1:numFilters
    pos(i,:) = indFilters{i}.State(1:2:end);
    cov(1:3,1:3,i) = indFilters{i}.StateCovariance(1:2:end,1:2:end);
end
componentPlot.plotTrack(pos,cov);

mixedPos = rpekf.State(1:2:end)';
mixedPosCov = rpekf.StateCovariance(1:2:end,1:2:end);
sumPlot.plotTrack(mixedPos,mixedPosCov);
```



## 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])`

**numFilters — Number of EKFs**

6 (default) | positive integer

Number of EKFs each initialized at an estimated range of the target, specified as a positive integer. When not specified, the default number of EKFs is 6.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

**rangeLimits — Range limits of target**

`[1e3 1e5]` (default) | two-element vector

Range limits of the target, specified as a two-element vector. The two elements in the vector represent the lower and upper limits of the target range. When not specified, the default range limits are `[1e3 1e5]` scenario units.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

**filter — Constant velocity range-parameterized EKF**

trackingGSF object

Constant velocity range-parameterized extended Kalman filter (EKF), returned as a `trackingGSF` object.

## References

- [1] Peach, N. "Bearings-only tracking using a set of range-parameterised extended Kalman filters." IEE Proceedings-Control Theory and Applications 142, no. 1 (1995): 73-80.

## Extended Capabilities

**C/C++ Code Generation**

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

## See Also

**Functions**

`initapekf` | `initcvekf` | `initcvmscekf`

**Objects**

`objectDetection` | `trackingEKF` | `trackingGSF`

**Introduced in R2018b**

# initekfimm

Initialize trackingIMM object

## Syntax

```
imm = initekfimm(detection)
```

## Description

`imm = initekfimm(detection)` initializes a constant velocity (CV), constant acceleration (CA), and a constant turn (CT) trackingIMM (`imm`) object based on information provided in an `objectDetection` object, `detection`.

## Examples

### Detection with Position Measurement in Rectangular Frame

A 3-D position measurement in rectangular frame is provided. For example,  $x = 1$ ,  $y = 3$ , and  $z = 0$ . Use a 3-D position measurement noise  $[1 \ 0.4 \ 0; \ 0.4 \ 4 \ 0; \ 0 \ 0 \ 1]$ .

```
detection = objectDetection(0, [1;3;0], 'MeasurementNoise', [1 0.4 0; 0.4 4 0; 0 0 1]);
```

Use `initekfimm` to create a `trackingIMM` filter initialized at the provided position and using the measurement noise defined above.

```
imm = initekfimm(detection);
```

Check the values of the state and measurement noise. Verify that the filter state, `imm.State`, has the same position components as detection measurement, `detection.Measurement`.

```
imm.State
```

```
ans = 6x1
```

```
1  
0  
3  
0  
0  
0
```

Verify that the filter measurement noise, `imm.MeasurementNoise`, is the same as the `detection.MeasurementNoise` values.

```
imm.MeasurementNoise
```

```
ans = 3x3
```

1.0000	0.4000	0
0.4000	4.0000	0

```
0          0      1.0000
```

## Detection with Position Measurement in Spherical Frame

A 3-D position measurement in spherical frame is provided. For example:  $\text{az} = 40$ ,  $\text{el} = 6$ ,  $r = 100$ ,  $\text{rr} = 5$ . Measurement noise is  $\text{diag}([2.5, 2.5, 0.5, 1].^2)$ .

```
meas = [40;6;100;5];
measNoise = diag([2.5,2.5,0.5,1].^2);
```

Use the `MeasurementParameters` to define the frame. You can leave out other fields of the `MeasurementParameters` struct, and they will be completed by default values. In this example, sensor position, sensor velocity, orientation, elevation, and range rate flags are default.

```
measParams = struct('Frame','spherical');
detection = objectDetection(0,meas,'MeasurementNoise',measNoise, ...
    'MeasurementParameters', measParams);
```

Use `initekfimm` to create a `trackingIMM` filter initialized at the provided position and using the measurement noise defined above.

```
imm = initekfimm(detection)
imm =
  trackingIMM with properties:

    State: [6x1 double]
    StateCovariance: [6x6 double]

    TrackingFilters: {3x1 cell}
    ModelConversionFcn: @switchimm
    TransitionProbabilities: [3x3 double]

    MeasurementNoise: [4x4 double]
    ModelProbabilities: [3x1 double]
```

## 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

**imm — trackingIMM object**  
`trackingIMM` object

Constant velocity (CV), constant acceleration (CA), and a constant turn (CT) `trackingIMM(imm)` object based on information provided in `detection`, returned as a `trackingIMM` object.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Objects

`objectDetection` | `trackingIMM`

### Functions

`initcaekf` | `initctekf` | `initcvekf`

**Introduced in R2018b**

# 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`.

## 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
        MeasurementNoise: [3x3 double]
        HasAdditiveMeasurementNoise: 1
```

Show the filter state.

```
filter.State
```

```
ans = 9x1
```

```
-200
0
0
-30
0
0
0
0
0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 9x9
```

```
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°, the elevation to 22°, 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 `trackerGNN` or `trackerTOMHT` object.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

[initcakf](#) | [initcaukf](#) | [initctekf](#) | [initctukf](#) | [initcvekf](#) | [initcvkf](#) | [initcvukf](#)

### Objects

[objectDetection](#) | [trackerGNN](#) | [trackerTOMHT](#) | [trackingEKF](#) | [trackingKF](#) | [trackingUKF](#)

**Introduced in R2018b**

# 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`.

## 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 = 6x1
```

```
10  
0  
0  
-5  
0  
0
```

Show the state transition model.

```
filter.StateTransitionModel
```

```
ans = 6x6
```

```
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 `trackerGNN` or `trackerTOMHT` object.

## Extended Capabilities

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

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

## See Also

### **Functions**

`initcaekf` | `initcaukf` | `initctekf` | `initctukf` | `initcvekf` | `initcvkf` | `initcvukf`

### **Objects**

`objectDetection` | `trackerGNN` | `trackerTOMHT` | `trackingEKF` | `trackingKF` | `trackingUKF`

### **Introduced in R2018b**

# 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`.

## 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
        MeasurementNoise: [3x3 double]
        HasAdditiveMeasurementNoise: 1

        Alpha: 1.0000e-03
        Beta: 2
        Kappa: 0
```

Show the state.

```
filter.State
```

```
ans = 9x1
```

```
-200
0
0
-30
0
0
0
5
0
0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 9x9
```

```
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°, 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 `trackerGNN` or `trackerTOMHT` object.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

#### Functions

[initcaekf](#) | [initcakf](#) | [initctekf](#) | [initctukf](#) | [initcvekf](#) | [initcvkf](#) | [initcvukf](#)

#### Objects

[objectDetection](#) | [trackerGNN](#) | [trackerTOMHT](#) | [trackingEKF](#) | [trackingKF](#) | [trackingUKF](#)

**Introduced in R2018b**

# 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`.

## 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
    MeasurementNoise: [3x3 double]
    HasAdditiveMeasurementNoise: 1
```

Show the state.

```
filter.State
```

```
ans = 7x1
```

```
-250
0
-40
0
0
0
0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 7x7
```

```
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 `trackerGNN` or `trackerTOMHT` object.

## Extended Capabilities

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

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

## See Also

### Functions

[initcaekf](#) | [initcakf](#) | [initcaukf](#) | [initctukf](#) | [initcvekf](#) | [initcvkf](#) | [initcvukf](#)

### Objects

[objectDetection](#) | [trackerGNN](#) | [trackerTOMHT](#) | [trackingEKF](#) | [trackingKF](#) | [trackingUKF](#)

### Introduced in R2018b

# 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`.

## 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
    MeasurementNoise: [3x3 double]
    HasAdditiveMeasurementNoise: 1

    Alpha: 1.0000e-03
    Beta: 2
    Kappa: 0
```

Show the filter state.

```
filter.State
```

```
ans = 7x1
```

```
-250
0
-40
0
0
0
0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 7x7
```

```
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 `trackerGNN` or `trackerTOMHT` object.

## Extended Capabilities

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

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

## See Also

### Functions

[initcaekf](#) | [initcakf](#) | [initcaukf](#) | [initctekf](#) | [initcvekf](#) | [initcvkf](#) | [initcvukf](#)

### Objects

[objectDetection](#) | [trackerGNN](#) | [trackerTOMHT](#) | [trackingEKF](#) | [trackingKF](#) | [trackingUKF](#)

### Introduced in R2018b

# 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`.

## 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
      MeasurementNoise: [3x3 double]
      HasAdditiveMeasurementNoise: 1
```

Show the filter state.

```
filter.State
```

```
ans = 6×1
```

```
10
0
20
0
-5
0
```

Show the state covariance.

```
filter.StateCovariance
```

```
ans = 6x6
```

```
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 `trackerGNN` or `trackerTOMHT` object.

## Extended Capabilities

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

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

## See Also

### **Functions**

`initcaekf` | `initcakf` | `initcaukf` | `initctekf` | `initctukf` | `initcvkf` | `initcvukf`

### **Objects**

`objectDetection` | `trackerGNN` | `trackerTOMHT` | `trackingEKF` | `trackingKF` | `trackingUKF`

### **Introduced in R2018b**

# 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`.

## 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'
    ControlModel: []
    ProcessNoise: [4x4 double]

    MeasurementModel: [2x4 double]
    MeasurementNoise: [2x2 double]
```

Show the state.

```
filter.State
```

```
ans = 4x1
```

```
10
0
20
```

0

Show the state transition model.

```
filter.StateTransitionModel
```

```
ans = 4x4
```

```
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: [6x1 double]  
StateCovariance: [6x6 double]  
  
MotionModel: '3D Constant Velocity'  
ControlModel: []  
ProcessNoise: [6x6 double]  
  
MeasurementModel: [3x6 double]  
MeasurementNoise: [3x3 double]
```

Show the state.

```
filter.State
```

```
ans = 6x1
```

```
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 `trackerGNN` or `trackerTOMHT` object.

## Extended Capabilities

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

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

## See Also

### **Functions**

`initcaekf` | `initcakf` | `initcaukf` | `initctekf` | `initctukf` | `initcvekf` | `initcvukf`

### **Objects**

`objectDetection` | `trackerGNN` | `trackerTOMHT` | `trackingEKF` | `trackingKF` | `trackingUKF`

### **Introduced in R2018b**

# 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`.

## 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
    MeasurementNoise: [3x3 double]
    HasAdditiveMeasurementNoise: 1

    Alpha: 1.0000e-03
    Beta: 2
    Kappa: 0
```

Display the state.

```
filter.State
```

```
ans = 6x1
```

```
10
0
200
0
-5
0
```

Show the state covariance.

```
filter.StateCovariance
```

```
ans = 6x6
```

```
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
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 `trackerGNN` or `trackerTOMHT` object.

## Extended Capabilities

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

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

## See Also

### **Functions**

`initcaekf` | `initcakf` | `initcaukf` | `initctekf` | `initctukf` | `initcvekf` | `initcvkf`

**Objects**

[objectDetection](#) | [trackerGNN](#) | [trackerTOMHT](#) | [trackingEKF](#) | [trackingKF](#) | [trackingUKF](#)

**Introduced in R2018b**

# 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 | trackingABF object | trackingCKF object | trackingIMM object | trackingGSF object | trackingPF object | trackingMSCEKF 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
- `trackingCKF` — Cubature Kalman filter
- `trackingIMM` — Interacting multiple model (IMM) filter
- `trackingGSF` — Gaussian-sum filter
- `trackingPF` — Particle filter
- `trackingMSCEKF` — Extended Kalman filter using modified spherical coordinates (MSC)

## 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` | `correctjpda` | `distance` | `initialize` | `likelihood` | `predict` | `residual`

**Introduced in R2018b**

## 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 | `trackingABF` object | `trackingCKF` object | `trackingIMM` object | `trackingGSF` object | `trackingPF` object | `trackingMSCEKF` 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
- `trackingCKF` — Cubature Kalman filter

- `trackingIMM` — Interacting multiple model (IMM) filter
- `trackingGSF` — Gaussian-sum filter
- `trackingPF` — Particle filter
- `trackingMSCEKF` — Extended Kalman filter using modified spherical coordinates (MSC)

**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` | `correctjpda` | `distance` | `initialize` | `likelihood` | `predict` | `residual`

Introduced in R2018b

## correctjpda

Correct state and state estimation error covariance using tracking filter and JPDA

### Syntax

```
[xcorr,Pcorr] = correctjpda(filter,zmeas)
[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)
correctjpda(filter,__)
xcorr = correctjpda(filter,__)
```

### Description

`[xcorr,Pcorr] = correctjpda(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. 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.

`correctjpda(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 = correctjpda(filter, ___)` updates `filter` with the corrected state and state estimation error covariance but returns only the corrected state, `xcorr`.

## Input Arguments

### **filter — Filter for object tracking**

trackingKF object | trackingEKF object | trackingUKF object | trackingABF object | trackingCKF object | trackingIMM object | trackingGSF object | trackingPF object | trackingMSCEKF 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
- `trackingCKF` — Cubature Kalman filter
- `trackingIMM` — Interacting multiple model (IMM) filter
- `trackingGSF` — Gaussian-sum filter
- `trackingPF` — Particle filter
- `trackingMSCEKF` — Extended Kalman filter using modified spherical coordinates (MSC)

### **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

$$x_k^+ = x_k^- + K_k(y - h(x_k^-))$$

$$P_k^+ = P_k^- - K_k S_k K_k^T$$

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

$$P_k = K_k \sum_{i=1}^N [\beta_i (y_i - h(x_k^-)) (y_i - h(x_k^-))^T - (\delta y) (\delta y)^T] K_k^T$$

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` | `trackerJPDA`

### Introduced in R2019a

# 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 | `trackingABF` object |  
`trackingCKF` object | `trackingIMM` object | `trackingGSF` object | `trackingPF` object |  
`trackingMSCEKF` 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
- `trackingCKF` — Cubature Kalman filter
- `trackingIMM` — Interacting multiple model (IMM) filter
- `trackingGSF` — Gaussian-sum filter
- `trackingPF` — Particle filter
- `trackingMSCEKF` — Extended Kalman filter using modified spherical coordinates (MSC)

### **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<sub>res</sub>**, 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<sup>2</sup>**, is

$$d^2 = z_{\text{res}}^T S^{-1} z,$$

The **distance** function computes the normalized distance, **d<sub>n</sub>**, 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 R2018b

# 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 | `trackingIMM` object | `trackingPF` object | `trackingMSCEKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingIMM` — Interacting multiple model (IMM) filter
- `trackingPF` — Particle filter
- `trackingMSCEKF` — Extended Kalman filter using modified spherical coordinates (MSC)

### **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 R2018b

# 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 | `trackingABF` object | `trackingCKF` object | `trackingIMM` object | `trackingGSF` object | `trackingPF` object | `trackingMSCEKF` 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
- `trackingCKF` — Cubature Kalman filter
- `trackingIMM` — Interacting multiple model (IMM) filter
- `trackingGSF` — Gaussian-sum filter
- `trackingPF` — Particle filter
- `trackingMSCEKF` — Extended Kalman filter using modified spherical coordinates (MSC)

### **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 R2018b**

# **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 | `trackingABF` object | `trackingCKF` object | `trackingIMM` object | `trackingGSF` object | `trackingPF` object | `trackingMSCEKF` object

Filter for object tracking, specified as one of these objects:

- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingABF` — Alpha-beta filter
- `trackingCKF` — Cubature Kalman filter
- `trackingIMM` — Interacting multiple model (IMM) filter
- `trackingGSF` — Gaussian-sum filter
- `trackingPF` — Particle filter
- `trackingMSCEKF` — Extended Kalman filter using modified spherical coordinates (MSC)

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 R2018b**

# **predict**

Predict state and state estimation error covariance of linear Kalman filter

## **Syntax**

```
[xpred,Ppred] = predict(filter)  
  
[xpred,Ppred] = predict(filter,u)  
[xpred,Ppred] = predict(filter,F)  
[xpred,Ppred] = predict(filter,F,Q)  
[xpred,Ppred] = predict(filter,u,F,G)  
[xpred,Ppred] = predict(filter,u,F,G,Q)  
  
[xpred,Ppred] = predict(filter,dt)  
[xpred,Ppred] = predict(filter,u,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 linear Kalman filter. The predicted values overwrite the internal state and state estimation error covariance of `filter`.

This syntax applies when you set the `ControlModel` property of `filter` to an empty matrix.

`[xpred,Ppred] = predict(filter,u)` specifies a control input, or force, `u`, and returns one or more of the outputs from the preceding syntaxes.

This syntax applies when you set the `ControlModel` property of `filter` to a nonempty matrix.

`[xpred,Ppred] = predict(filter,F)` specifies the state transition model, `F`. Use this syntax to change the state transition model during a simulation.

This syntax applies when you set the `ControlModel` property of `filter` to an empty matrix.

`[xpred,Ppred] = predict(filter,F,Q)` specifies the state transition model, `F`, and the process noise covariance, `Q`. Use this syntax to change the state transition model and process noise covariance during a simulation.

This syntax applies when you set the `ControlModel` property of `filter` to an empty matrix.

`[xpred,Ppred] = predict(filter,u,F,G)` specifies the force or control input, `u`, the state transition model, `F`, and the control model, `G`. Use this syntax to change the state transition model and control model during a simulation.

This syntax applies when you set the `ControlModel` property of `filter` to a nonempty matrix.

[*xpred*,*Ppred*] = predict(*filter*,*u*,*F*,*G*,*Q*) specifies the force or control input, *u*, the state transition model, *F*, the control model, *G*, and the process noise covariance, *Q*. Use this syntax to change the state transition model, control model, and process noise covariance during a simulation.

This syntax applies when you set the **ControlModel** property of *filter* to a nonempty matrix.

[*xpred*,*Ppred*] = predict(*filter*,*dt*) returns the predicted outputs after time step *dt*.

This syntax applies when the **MotionModel** property of *filter* is not set to 'Custom' and the **ControlModel** property is set to an empty matrix.

[*xpred*,*Ppred*] = predict(*filter*,*u*,*dt*) also specifies a force or control input, *u*.

This syntax applies when the **MotionModel** property of *filter* is not set to 'Custom' and the **ControlModel** property is set to a nonempty matrix.

**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 Linear Kalman Filter

Create a linear Kalman filter that uses a 2D Constant Velocity motion model. Assume that the measurement consists of the object's x-y location.

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 the measured positions from 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
```

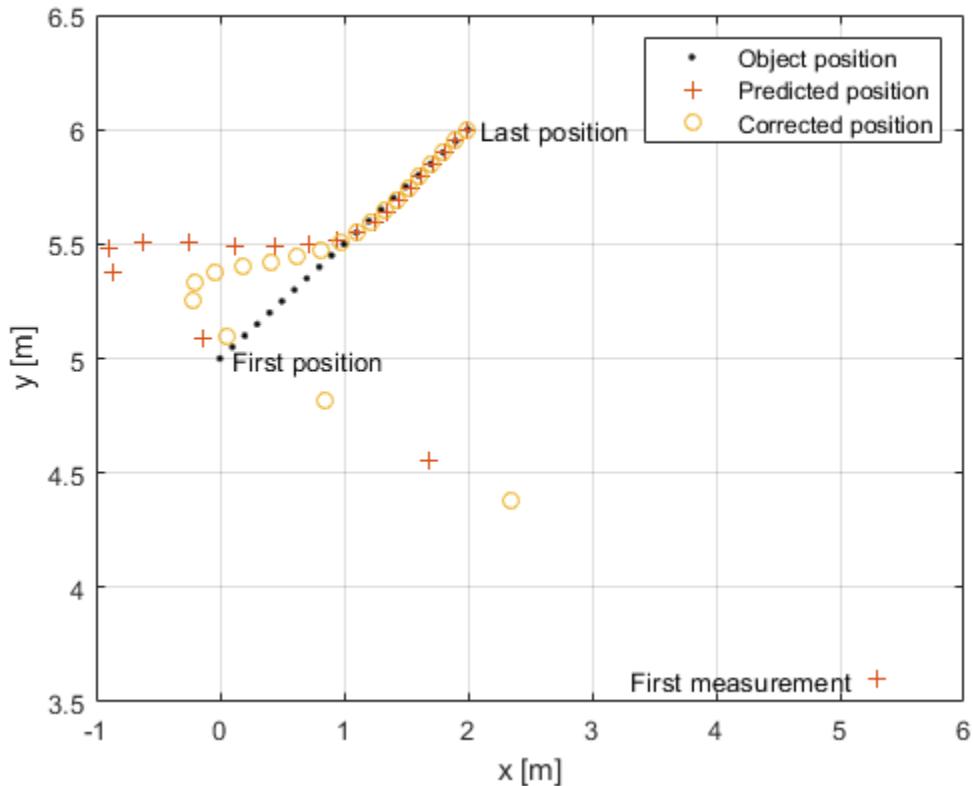
Plot the tracks.

```
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')

```



## Input Arguments

**filter — Linear Kalman filter for object tracking**  
trackingKF object

Linear Kalman filter for object tracking, specified as a `trackingKF` object.

**u — Control vector**  
real-valued  $L$ -element vector

Control vector, specified as a real-valued  $L$ -element vector.

**F — 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 — Process noise covariance matrix**

positive-definite, real-valued  $M$ -by- $M$  matrix

Process noise covariance matrix, specified as a positive-definite, real-valued  $M$ -by- $M$  matrix, where  $M$  is the length of the state vector.

**G — 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.

**dt — Time step**

positive scalar

Time step, specified as a positive scalar. Units are in seconds.

## 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, specified 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](#) | [correctjpda](#) | [distance](#) | [initialize](#) | [likelihood](#) | [residual](#)

**Introduced in R2018b**

# 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 | `trackingCKF` object | `trackingMSCEKF` object

Filter for object tracking, specified as one of these objects:

- `trackingKF` — Linear Kalman filter
- `trackingEKF` — Extended Kalman filter
- `trackingUKF` — Unscented Kalman filter
- `trackingCKF` — Cubature Kalman filter
- `trackingMSCEKF` — Extended Kalman filter using modified spherical coordinates (MSC)

### **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` | `correctjpda` | `distance` | `initialize` | `likelihood` | `predict`

**Introduced in R2018b**

# singer

Singer acceleration motion model

## Syntax

```
updatedstates = singer(states)
updatedstates = singer(states,dt)
updatedstates = singer(states,dt,tau)
```

## Description

`updatedstates = singer(states)` returns the updated states from the current `states` based on the Singer acceleration motion model. The default time step is 1 second.

`updatedstates = singer(states,dt)` specifies the time step, `dt`, in seconds.

`updatedstates = singer(states,dt,tau)` specifies the target maneuver time constant, `tau`, in seconds. The default target maneuver time constant is 20 seconds.

## Examples

### Predict Multiple Singer Acceleration States

Define a state matrix for a 2-D Singer acceleration motion.

```
states = [1 2 2.5;1 2.5 3;0 -1 2;2 3 -1;5 0 3;-2 4 2];
```

Predict the states by using a default time step interval `dt` = 1 second.

```
states = singer(states)

states = 6x3
```

2.0000	4.0082	6.4835
1.0000	1.5246	4.9508
0	-0.9512	1.9025
6.0165	4.9671	2.9835
3.0492	3.9016	4.9508
-1.9025	3.8049	1.9025

Predict the state by using `dt` = 0.1 second.

```
states = singer(states,0.1)

states = 6x3

2.1000    4.1559    6.9881
1.0000    1.4297    5.1406
0         -0.9465   1.8930
6.3119    5.3762    3.4881
```

```
2.8594    4.2812    5.1406
-1.8930   3.7859    1.8930
```

## Predict and Measure Position Using Singer Model

Define a state vector for a 2-D Singer acceleration motion.

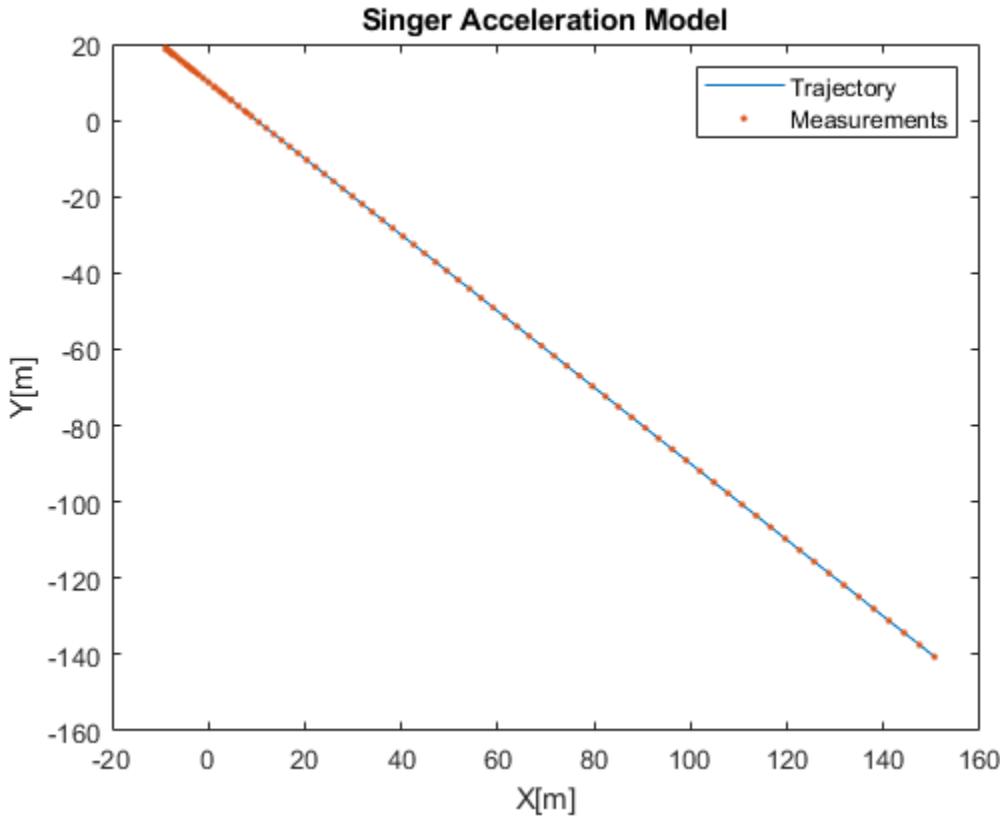
```
state = [10;-10;3;0;10;-3];
dt = 0.2; % time step in seconds
tau = 10; % maneuver time in seconds
```

Use the `singer` function to create a trajectory and measure the positions using the `singermeas` function.

```
positions = zeros(2,100); % Pre-allocate memory
measurements = zeros(3,100); % Pre-allocate memory
for i = 1:1:100
    state = singer(state, dt, tau);
    positions(:,i) = [state(1); state(4)];
    measurements(:,i) = singermeas(state);
end
```

Visualize the results.

```
plot(positions(1,:), positions(2,:))
hold on
plot(measurements(1,:), measurements(2,:), '.')
title('Singer Acceleration Model');
xlabel('X[m]'); ylabel('Y[m]');
legend('Trajectory', 'Measurements');
```



## Input Arguments

### **states — Current states**

real-valued  $3N$ -by-1 vector | real-valued  $3N$ -by- $M$  matrix

Current states, specified as a real-valued  $3N$ -by-1 vector or a real-valued  $3N$ -by- $M$  matrix.  $N$  is the spatial degree of the state, and  $M$  is the number of states.

The state vector in each column takes different forms based on its spatial dimensions.

Spatial Degrees	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  $\text{m/s}^2$ .

Example: [5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]

**dt — Time step**

1.0 (default) | positive scalar

Time step, specified as a positive scalar in seconds.

Example: 0.5

**tau — Target maneuver time constant**20 (default) | positive scalar |  $N$ -element vector of scalarTarget maneuver time constant, specified as a positive scalar or an  $N$ -element vector of scalars in seconds.  $N$  is the spatial degree of the state. When specified as a vector, each element applies to the corresponding spatial dimension.

Example: 30

## Output Arguments

**updatedstates — Updated states**real-valued  $3N$ -by-1 vector | real-valued  $3N$ -by- $M$  matrixUpdated states, returned as a real-valued  $3N$ -by-1 vector or a real-valued  $3N$ -by- $M$  matrix.  $N$  is the spatial degree of the state, and  $M$  is the number of states. The `updatedStates` output has the exactly same form as the `states` input.

## Algorithms

The Singer acceleration model assumes the acceleration at time step  $k+1$ , which depends on the acceleration at time step  $k$  with exponential decay as:

$$a(k+1) = a(k) * \exp(-T/\tau)$$

where  $a(k)$  is the acceleration at time step  $k$ ,  $T$  is the time step, and  $\tau$  is the target maneuver time constant.

For a 1-D singer model state  $p = [x, vx, ax]^T$ , the state propagation is:

$$p(k) = \begin{bmatrix} 1 & T(\alpha T - 1 - e^{-\alpha T})/\alpha^2 \\ 0 & 1 - (1 - e^{-\alpha T})/\alpha \\ 0 & e^{-\alpha T} \end{bmatrix} p(k) + w(k)$$

where  $\alpha = 1/\tau$  is the reciprocal of the target maneuver time constant and  $w(k)$  is the Singer model process noise at time step  $k$ . See `singerProcessNoise` for more details on the process noise.

## References

- [1] Singer, Robert A. "Estimating optimal tracking filter performance for manned maneuvering targets." *IEEE Transactions on Aerospace and Electronic Systems* 4 (1970): 473-483.
- [2] Blackman, Samuel S., and Robert Popoli. "Design and analysis of modern tracking systems." (1999).

- 
- [3] Li, X. Rong, and Vesselin P. Jilkov. "Survey of maneuvering target tracking: dynamic models." Signal and Data Processing of Small Targets 2000, vol. 4048, pp. 212-235. International Society for Optics and Photonics, 2000.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`initsingerekf` | `singerProcessNoise` | `singerjac` | `singermeas` | `singermeasjac`

**Introduced in R2020b**

# singerjac

Jacobian of Singer acceleration motion model

## Syntax

```
jacobian = singerjac(state)
jacobian = singerjac(state,dt)
jacobian = singerjac(state,dt,tau)
```

## Description

`jacobian = singerjac(state)` returns the Jacobian matrix of the Singer motion model with respect to the state vector. The default time step is 1 second.

`jacobian = singerjac(state,dt)` specifies the time step `dt` in seconds.

`jacobian = singerjac(state,dt,tau)` specifies the target maneuver time constant, `tau`, in seconds. The default target maneuver time constant is 20 seconds.

## Examples

### Jacobian of Singer Model

Define a state for a 2-D Singer acceleration motion.

```
state = [1;1;1;2;1;0];
```

Calculate the Jacobian matrix assuming `dt` = 1 second.

```
jac1 = singerjac(state)
```

```
jac1 = 6x6
```

```
1.0000    1.0000    0.4918      0      0      0
      0    1.0000    0.9754      0      0      0
      0      0    0.9512      0      0      0
      0      0      0    1.0000    1.0000    0.4918
      0      0      0      0    1.0000    0.9754
      0      0      0      0      0    0.9512
```

Calculate the Jacobian matrix assuming `dt` = 0.1 second.

```
jac2 = singerjac(state, 0.1)
```

```
jac2 = 6x6
```

```
1.0000    0.1000    0.0050      0      0      0
      0    1.0000    0.0998      0      0      0
      0      0    0.9950      0      0      0
      0      0      0    1.0000    0.1000    0.0050
```

0	0	0	0	1.0000	0.0998
0	0	0	0	0	0.9950

## Input Arguments

### **state — Current state**

real-valued  $3N$ -by-1 vector

Current state, specified as a real-valued  $3N$ -by-1 vector.  $N$  is the spatial degree of the state. The state vector takes the different forms based on its dimensions.

Spatial Degrees	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 m/s. Acceleration coordinates are in  $\text{m/s}^2$ .

Example: [5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]

### **dt — Time step**

1.0 (default) | positive scalar

Time step, specified as a positive scalar in seconds.

Example: 0.5

### **tau — Target maneuver time constant**

20 (default) | positive scalar |  $N$ -element vector of scalar

Target maneuver time constant, specified as a positive scalar or an  $N$ -element vector of scalars in seconds.  $N$  is the spatial degree of the state. When specified as a vector, each element applies to the corresponding spatial dimension.

Example: 30

## Output Arguments

### **jacobian — Jacobian matrix of Singer model**

$3N$ -by- $3N$  matrix of real scalar

The Jacobian matrix of a Singer model, returned as a  $3N$ -by- $3N$  matrix of real scalars.  $N$  is the spatial degree of the state input.

## Algorithms

Given the dimension of the state space, the Jacobian of a Singer model takes different forms.

For 1-D state space, the Jacobian matrix is calculated as

$$J_1 = \begin{bmatrix} 1 & T & \tau^2(-T/\tau + \beta) \\ 0 & 1 & \tau(1 - \beta) \\ 0 & 0 & 0 \end{bmatrix}$$

where  $T$  is the time step interval,  $\tau$  is the target maneuver time constant, and  $\beta = \exp(-T/\tau)$ .

For 2-D state space, the Jacobian matrix is calculated as

$$J_2 = \begin{bmatrix} J_1 & 0 \\ 0 & J_1 \end{bmatrix}$$

For 3-D state space, the Jacobian matrix is calculated as

$$J_3 = \begin{bmatrix} J_1 & 0 & 0 \\ 0 & J_1 & 0 \\ 0 & 0 & J_1 \end{bmatrix}$$

## References

- [1] Singer, Robert A. "Estimating optimal tracking filter performance for manned maneuvering targets." IEEE Transactions on Aerospace and Electronic Systems 4 (1970): 473-483.
- [2] Blackman, Samuel S., and Robert Popoli. "Design and analysis of modern tracking systems." (1999).
- [3] Li, X. Rong, and Vesselin P. Jilkov. "Survey of maneuvering target tracking: dynamic models." Signal and Data Processing of Small Targets 2000, vol. 4048, pp. 212-235. International Society for Optics and Photonics, 2000.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`initsingerekf` | `singer` | `singerProcessNoise` | `singermeas` | `singermeasjac`

### Introduced in R2020b

# singermeas

Measurement function for Singer acceleration motion model

## Syntax

```
measurements = singermeas(states)
measurements = singermeas(states,frame)
measurements = singermeas(states,frame,sensorpos,sensorvel)
measurements = singermeas(states,frame,sensorpos,sensorvel,laxes)
measurements = singermeas(states,measurementParameters)
```

## Description

`measurements = singermeas(states)` returns the measurements in rectangular coordinates for the Singer motion model based on the current `states`.

`measurements = singermeas(states,frame)` specifies the measurement output coordinate system, `frame`.

`measurements = singermeas(states,frame,sensorpos,sensorvel)` also specifies the sensor position, `sensorpos`, and the sensor velocity, `sensorvel`.

`measurements = singermeas(states,frame,sensorpos,sensorvel,laxes)` specifies the local sensor axes orientation, `laxes`.

`measurements = singermeas(states,measurementParameters)` specifies the measurement parameters, `measurementParameters`.

## Examples

### Measurements for Singer Model

Define a state for a 2-D Singer acceleration motion.

```
state = [1;10;3;2;20;5];
```

Obtain the measurement in a rectangular frame.

```
measurement = singermeas(state)
measurement = 3x1
```

```
1
2
0
```

Obtain the measurement in a spherical frame.

```
measurement = singermeas(state, 'spherical')
```

```
measurement = 4x1
```

```
63.4349  
0  
2.2361  
22.3607
```

Obtain the measurement in a spherical frame relative to a stationary sensor located at [1;-2;0].

```
measurement = singermeas(state, 'spherical', [1;-2;0], [0;0;0])
```

```
measurement = 4x1
```

```
90  
0  
4  
20
```

Obtain the measurement in a spherical frame relative to a stationary sensor located at [1;-2;0] that is rotated by 90 degrees around the z axis relative to the global frame.

```
laxes = [0 -1 0; 1 0 0; 0 0 1];  
measurement = singermeas(state, 'spherical', [1;-2;0], [0;0;0], laxes)
```

```
measurement = 4x1
```

```
0  
0  
4  
20
```

Obtain measurements from multiple 2D states in a rectangular frame.

```
states = [1 2 3; 10 20 30; 2 4 5; 20 30 40; 5 6 11; 1 3 1.5];  
measurements = singermeas(states)
```

```
measurements = 3x3
```

```
1      2      3  
20     30     40  
0      0      0
```

## Input Arguments

### **states — Current states**

real-valued  $3N$ -by-1 vector | real-valued  $3N$ -by- $M$  matrix

Current states, specified as a real-valued  $3N$ -by-1 vector or a real-valued  $3N$ -by- $M$  matrix.  $N$  is the spatial degree of the state, and  $M$  is the number of states.

The state vector in each column takes different forms based on its spatial dimensions.

Spatial Degrees	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 m/s<sup>2</sup>.

Example: [5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]

**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. For more details, see "Measurement Parameters" on page 1-403.

Data Types: struct

## Output Arguments

### **measurements — Measurements**

*N*-by-1 column vector of scalar | *N*-by-*M* matrix of scalar

Measurement vector, returned as an *N*-by-1 column vector of scalars or an *N*-by-*M* matrix of scalars. The form of the 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, <code>az</code> , elevation angle, <code>el</code> , range, <code>r</code> , and range rate, <code>rr</code> of the measurements.			
<b>Spherical Measurements</b>				
<b>HasVelocity</b>			<b>HasElevation</b>	
	false	$[az; r]$	$[az; el; r]$	
	true	$[az; r; rr]$	$[az; el; r; rr]$	
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 measurements.			
<b>Rectangular Measurements</b>				
<b>HasVelocity</b>	false	$[x; y; z]$		
	true	$[x; y; z; vx; vy; vz]$		
Position units are in meters and velocity units are in m/s.				

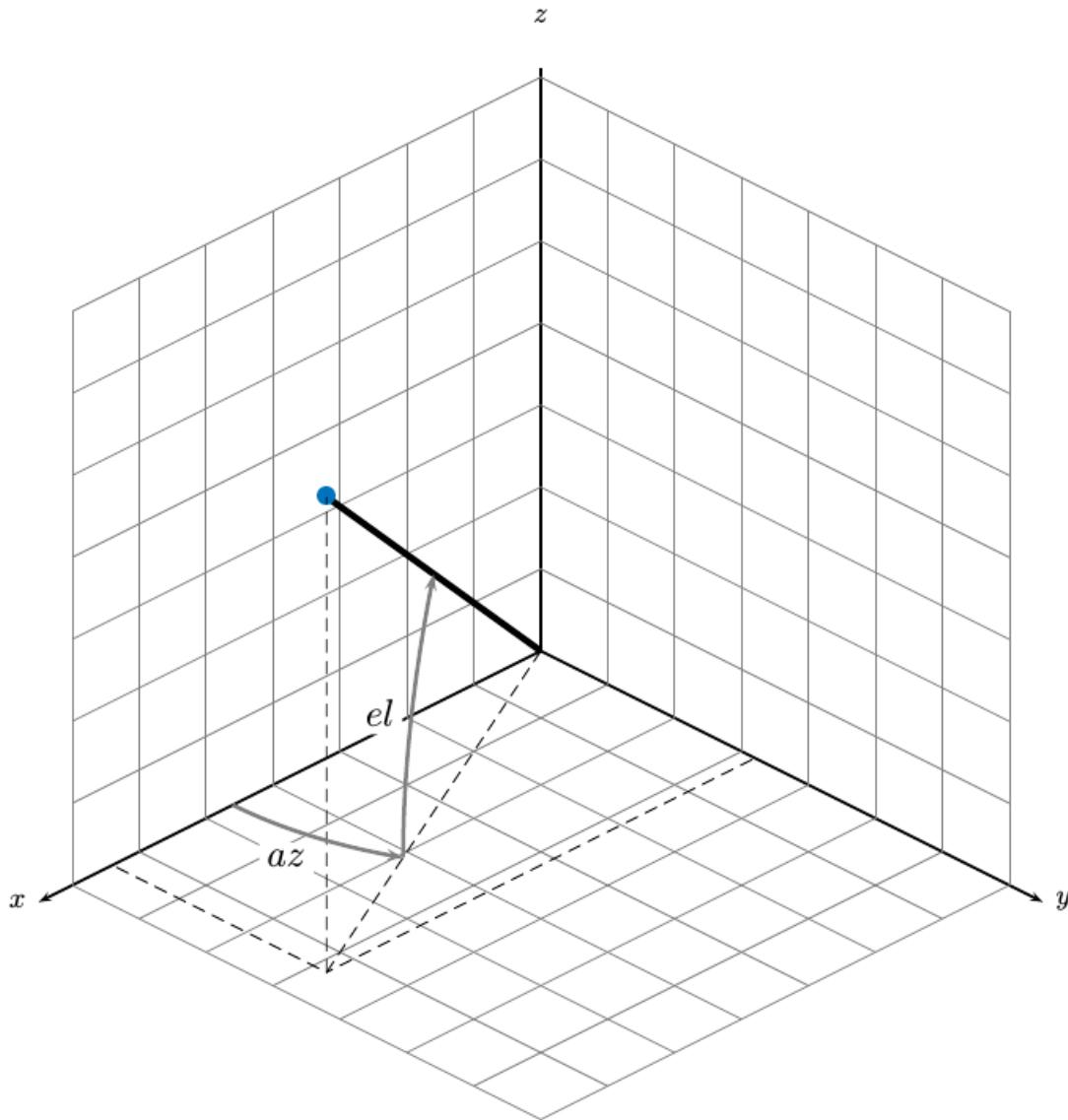
Data Types: double

## More About

### **Azimuth and Elevation Angle Definitions**

Define the azimuth and elevation angles used in Sensor Fusion and Tracking 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.



### Measurement Parameters

The **MeasurementParameters** property consists of an array of structures that describe a sequence of coordinate transformations from a child frame to a parent frame or the inverse transformations (see “Frame Rotation”). If **MeasurementParameters** only contains one structure, then it represents the rotation from one frame to the other. If **MeasurementParameters** contains an array of structures, then it represents rotations between multiple frames.

The fields of `MeasurementParameters` are shown here. Not all fields have to be present in the structure.

Field	Description
Frame	Enumerated type indicating the frame used to report measurements. When detections are reported using a rectangular coordinate system, <code>Frame</code> is set to ' <code>'rectangular'</code> '. When detections are reported in spherical coordinates, <code>Frame</code> is set ' <code>'spherical'</code> ' for the first <code>struct</code> .
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 <code>IsParentToChild</code> field.
IsParentToChild	A logical scalar indicating if <code>Orientation</code> performs a frame rotation from the parent coordinate frame to the child coordinate frame. If <code>false</code> , <code>Orientation</code> 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, and if <code>HasElevation</code> is <code>false</code> , 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 the rectangular frame, if <code>HasVelocity</code> is <code>false</code> , the measurements are reported as <code>[x y z]</code> . If <code>HasVelocity</code> is <code>true</code> , measurements are reported as <code>[x y z vx vy vz]</code> .

## References

- [1] Singer, Robert A. "Estimating optimal tracking filter performance for manned maneuvering targets." *IEEE Transactions on Aerospace and Electronic Systems* 4 (1970): 473-483.
- [2] Blackman, Samuel S., and Robert Popoli. "Design and analysis of modern tracking systems." (1999).

- [3] Li, X. Rong, and Vesselin P. Jilkov. "Survey of maneuvering target tracking: dynamic models." Signal and Data Processing of Small Targets 2000, vol. 4048, pp. 212-235. International Society for Optics and Photonics, 2000.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`initsingerekf` | `singer` | `singerProcessNoise` | `singerjac` | `singermeasjac`

**Introduced in R2020b**

# singermeasjac

Jacobian of measurement function for Singer acceleration motion model

## Syntax

```
jacobian = singermeasjac(state)
jacobian = singermeasjac(state,frame)
jacobian = singermeasjac(state,frame,sensorpos,sensorvel)
jacobian = singermeasjac(state,frame,sensorpos,sensorvel,laxes)
jacobian = singermeasjac(state,measurementParameters)
```

## Description

`jacobian = singermeasjac(state)` returns the measurement Jacobian in rectangular coordinates with respect to the state for the Singer acceleration motion model.

`jacobian = singermeasjac(state,frame)` specifies the measurement Jacobian output coordinate system, `frame`.

`jacobian = singermeasjac(state,frame,sensorpos,sensorvel)` specifies the sensor position, `sensorpos`, and the sensor velocity, `sensorvel`.

`jacobian = singermeasjac(state,frame,sensorpos,sensorvel,laxes)` specifies the local sensor axes orientation, `laxes`.

`jacobian = singermeasjac(state,measurementParameters)` specifies the measurement parameters, `measurementParameters`.

## Examples

### Obtain Measurement Jacobian Matrix of Singer Model

Define a state for 2-D Singer acceleration motion.

```
state = [1;10;0;2;20;1];
```

Obtain the measurement Jacobian in a rectangular frame.

```
jacobian = singermeasjac(state)
```

```
jacobian = 3x6
```

```
1     0     0     0     0     0
0     0     0     1     0     0
0     0     0     0     0     0
```

Obtain the measurement Jacobian in a spherical frame.

```
jacobian = singermeasjac(state, 'spherical')
```

```
jacobian = 4x6
```

```
-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
```

Obtain the measurement Jacobian in a spherical frame relative to a stationary sensor located at [1;-2;0].

```
jacobian = singermeasjac(state, 'spherical', [1;-2;0], [0;0;0])
```

```
jacobian = 4x6
```

```
-14.3239      0      0        0      0      0
      0      0      0        0      0      0
      0      0      0    1.0000      0      0
  2.5000      0      0        0    1.0000      0
```

Obtain the measurement Jacobian in a spherical frame relative to a stationary sensor located at [1;-2;0] that is rotated by 90 degrees around the z axis relative to the global frame.

```
laxes = [0 -1 0; 1 0 0; 0 0 1];
jacobian = singermeasjac(state, 'spherical', [1;-2;0], [0;0;0], laxes)
```

```
jacobian = 4x6
```

```
-14.3239      0      0        0      0      0
      0      0      0        0      0      0
      0      0      0    1.0000      0      0
  2.5000      0      0        0    1.0000      0
```

## Input Arguments

### **state — Current state**

real-valued  $3N$ -by-1 vector

Current state, specified as a real-valued  $3N$ -by-1 vector.  $N$  is the spatial degree of the state. The state vector takes the different forms based on its dimensions.

Spatial Degrees	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 m/s. Acceleration coordinates are in m/s<sup>2</sup>.

Example: [5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]

**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. For more details, see "Measurement Parameters" on page 1-410.

Data Types: struct

## Output Arguments

**jacobian — Measurement Jacobian**

real-valued 3-by- $N$  matrix | real-valued 4-by- $N$  matrix

The measurement Jacobian for a Singer model, returned as a real-valued 3-by- $N$  for a rectangular frame or 4-by- $N$  matrix for a spherical frame.  $N$  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. Coordinates are in meters.

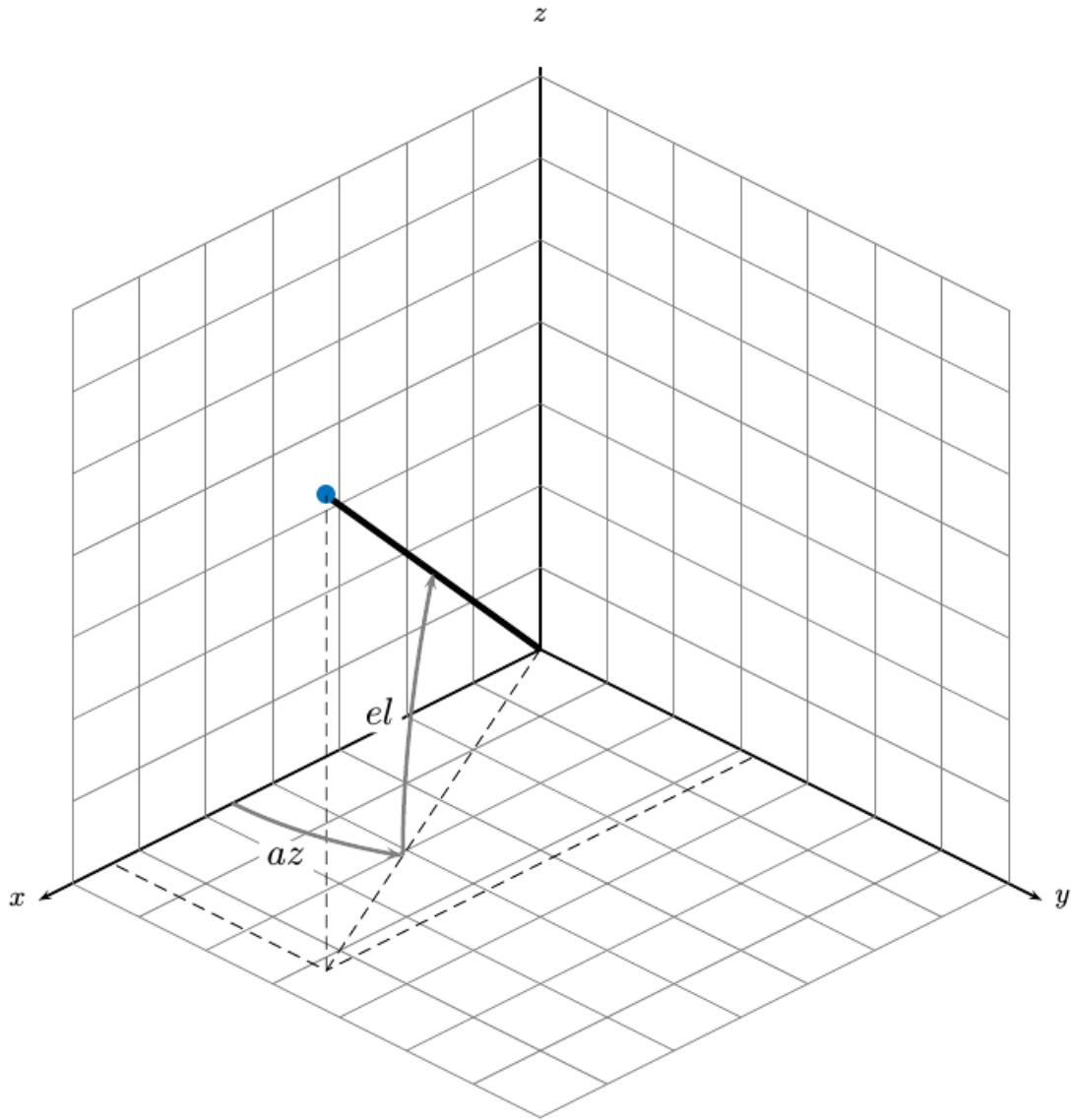
Frame	Measurement Jacobian
'spherical'	Jacobian of the measurement vector [az;el;r;rr] with respect to the state vector. Measurement vector is the azimuth angle, elevation angle, range, and range rate of the object in the local sensor coordinate system. Angle units are in degrees. Range units are in meters and range rate units are in ms/s.

## More About

### Azimuth and Elevation Angle Definitions

Define the azimuth and elevation angles used in Sensor Fusion and Tracking 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.



### Measurement Parameters

The **MeasurementParameters** property consists of an array of structures that describe a sequence of coordinate transformations from a child frame to a parent frame or the inverse transformations (see “Frame Rotation”). If **MeasurementParameters** only contains one structure, then it represents the rotation from one frame to the other. If **MeasurementParameters** contains an array of structures, then it represents rotations between multiple frames.

The fields of **MeasurementParameters** are shown here. Not all fields have to be present in the structure.

Field	Description
-------	-------------

Frame	Enumerated type indicating the frame used to report measurements. When detections are reported using a rectangular coordinate system, <b>Frame</b> is set to ' <b>rectangular</b> '. When detections are reported in spherical coordinates, <b>Frame</b> is set to ' <b>spherical</b> ' for the first struct.
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 <b>IsParentToChild</b> field.
IsParentToChild	A logical scalar indicating if <b>Orientation</b> performs a frame rotation from the parent coordinate frame to the child coordinate frame. If <b>false</b> , <b>Orientation</b> 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, and if <b>HasElevation</b> is <b>false</b> , 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 the rectangular frame, if <b>HasVelocity</b> is <b>false</b> , the measurements are reported as [x y z]. If <b>HasVelocity</b> is <b>true</b> , measurements are reported as [x y z vx vy vz].

## References

- [1] Singer, Robert A. "*Estimating optimal tracking filter performance for manned maneuvering targets.*" IEEE Transactions on Aerospace and Electronic Systems 4 (1970): 473-483.
- [2] Blackman, Samuel S., and Robert Popoli. "*Design and analysis of modern tracking systems.*" (1999).

- [3] Li, X. Rong, and Vesselin P. Jilkov. "Survey of maneuvering target tracking: dynamic models." Signal and Data Processing of Small Targets 2000, vol. 4048, pp. 212-235. International Society for Optics and Photonics, 2000.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`initsingerekf` | `singer` | `singerProcessNoise` | `singerjac` | `singermeas`

**Introduced in R2020b**

# singerProcessNoise

Process noise matrix for Singer acceleration model

## Syntax

```
processNoise = singerProcessNoise(state)
processNoise = singerProcessNoise(state,dt)
processNoise = singerProcessNoise(state,dt,tau)
processNoise = singerProcessNoise(state,dt,tau,sigma)
```

## Description

`processNoise = singerProcessNoise(state)` returns the process noise matrix for the Singer acceleration model based on the current state. For more details, see Reference [3].

`processNoise = singerProcessNoise(state,dt)` specifies the time step `dt`. The default time step is 1 second.

`processNoise = singerProcessNoise(state,dt,tau)` specifies the target maneuver time constant `tau`. The default maneuver time constant is 20 seconds.

`processNoise = singerProcessNoise(state,dt,tau,sigma)` specifies the target maneuver standard deviation `sigma`. The default maneuver standard deviation is 1 meter per second squared.

## Examples

### Process Noise Matrix for Singer Acceleration Model

Obtain the Singer process noise for a 3-D Singer state that has a default time step, a target maneuver time constant, and a standard deviation.

```
Q1 = singerProcessNoise((1:9)')
```

```
Q1 = 9x9
```

0.0049	0.0121	0.0159	0	0	0	0	0	0
0.0121	0.0321	0.0476	0	0	0	0	0	0
0.0159	0.0476	0.0952	0	0	0	0	0	0
0	0	0	0.0049	0.0121	0.0159	0	0	0
0	0	0	0.0121	0.0321	0.0476	0	0	0
0	0	0	0.0159	0.0476	0.0952	0	0	0
0	0	0	0	0	0	0.0049	0.0121	0.0159
0	0	0	0	0	0	0.0121	0.0321	0.0476
0	0	0	0	0	0	0.0159	0.0476	0.0952

Set the time step as 2 seconds. Set the target maneuver time constant as 10 seconds in x- and y- axes and as 100 seconds in z-axis. Set the target maneuver standard deviation as  $1\text{m/s}^2$  in x- and y- axes and  $0\text{ m/s}^2$  in z-axis.

```
Q2 = singerProcessNoise((1:9)', 2, [10 10 100], [1 1 0])
```

Q2 = 9×9

0.2868	0.3508	0.2188	0	0	0	0	0	0
0.3508	0.4603	0.3286	0	0	0	0	0	0
0.2188	0.3286	0.3297	0	0	0	0	0	0
0	0	0	0.2868	0.3508	0.2188	0	0	0
0	0	0	0.3508	0.4603	0.3286	0	0	0
0	0	0	0.2188	0.3286	0.3297	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

## Input Arguments

### state — Current state

real-valued  $3N$ -by-1 vector

Current state, specified as a real-valued  $3N$ -by-1 vector.  $N$  is the spatial degree of the state. The state vector takes the different forms based on its dimensions.

Spatial Degrees	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 m/s. Acceleration coordinates are in m/s<sup>2</sup>.

Example: [5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]

### dt — Time step

1.0 (default) | positive scalar

Time step, specified as a positive scalar in seconds.

Example: 0.5

### tau — Target maneuver time constant

20 (default) | positive scalar |  $N$ -element vector of scalar

Target maneuver time constant, specified as a positive scalar or an  $N$ -element vector of scalars in seconds.  $N$  is the spatial degree of the state. When specified as a vector, each element applies to the corresponding spatial dimension.

Example: 30

### sigma — Maneuver standard deviation

1 (default) | positive scalar |  $N$ -element vector of scalar

Maneuver standard deviation, specified as a positive scalar or an  $N$ -element vector of scalars in m/s<sup>2</sup>.  $N$  is the spatial degree of the state. When specified as a vector, each element applies to the corresponding spatial dimension.

Example: 3

## Output Arguments

### **processNoise — Process noise for Singer acceleration model**

$N$ -by- $N$  matrix of nonnegative scalars

Process noise for a Singer acceleration model, returned as an  $N$ -by- $N$  matrix of nonnegative scalars.  $N$  is the spatial dimension of the `state` input.

## References

- [1] Singer, Robert A. "Estimating optimal tracking filter performance for manned maneuvering targets." IEEE Transactions on Aerospace and Electronic Systems 4 (1970): 473-483.
- [2] Blackman, Samuel S., and Robert Popoli. "Design and analysis of modern tracking systems." (1999).
- [3] Li, X. Rong, and Vesselin P. Jilkov. "Survey of maneuvering target tracking: dynamic models." Signal and Data Processing of Small Targets 2000, vol. 4048, pp. 212-235. International Society for Optics and Photonics, 2000.

## Extended Capabilities

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

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

## See Also

`initsingerekf` | `singer` | `singerjac` | `singermeas` | `singermeasjac`

## Introduced in R2020b

# assignauction

Assignment using auction global nearest neighbor

## Syntax

```
[assignments,unassignedrows,unassignedcolumns] = assignauction(costmatrix,  
costofnonassignment)
```

## Description

[assignments,unassignedrows,unassignedcolumns] = assignauction(costmatrix, costofnonassignment) returns a table of assignments of detections to tracks derived based on the forward/reverse auction algorithm. The auction algorithm finds a suboptimal solution to the global nearest neighbor (GNN) assignment problem by minimizing the total cost of assignment. While suboptimal, the auction algorithm is faster than the Munkres algorithm for large GNN assignment problems, for example, when there are more than 50 rows and columns in the cost matrix.

The cost of each potential assignment is contained in the cost matrix, **costmatrix**. Each matrix entry represents the cost of a possible assignments. Matrix rows represent tracks and columns represent detections. All possible assignments are represented in the cost matrix. The lower the cost, the more likely the assignment is to be made. Each track can be assigned to at most one detection and each detection can be assigned to at most one track. If the number of rows is greater than the number of columns, some tracks are unassigned. If the number of columns is greater than the number of rows, some detections are unassigned. You can set an entry of **costmatrix** to Inf to prohibit an assignment.

**costofnonassignment** represents the cost of leaving tracks or detections unassigned. Higher values increase the likelihood that every existing object is assigned.

The function returns a list of unassigned tracks, **unassignedrows**, and a list of unassigned detections, **unassignedcolumns**.

## Examples

### Assign Detections to Tracks Using Auction Algorithm

Use **assignAuction** to assign three detections to two tracks.

Start with two predicted track locations in x-y coordinates.

```
tracks = [1,1; 2,2];
```

Assume three detections are received. At least one detection will not be assigned.

```
dets = [1.1, 1.1; 2.1, 2.1; 1.5, 3];
```

Construct a cost matrix by defining the cost of assigning a detection to a track as the Euclidean distance between them. Set the cost of non-assignment to 0.2.

```
for i = size(tracks, 1):-1:1
    delta = dets - tracks(i, :);
    costMatrix(i, :) = sqrt(sum(delta .^ 2, 2));
end
costofnonassignment = 0.2;
```

Use the Auction algorithm to assign detections to tracks.

```
[assignments, unassignedTracks, unassignedDetections] = ...
    assignauction(costMatrix,costofnonassignment);
```

Display the assignments.

```
disp(assignments)
```

```
1 1
2 2
```

Show that there are no unassigned tracks.

```
disp(unassignedTracks)
```

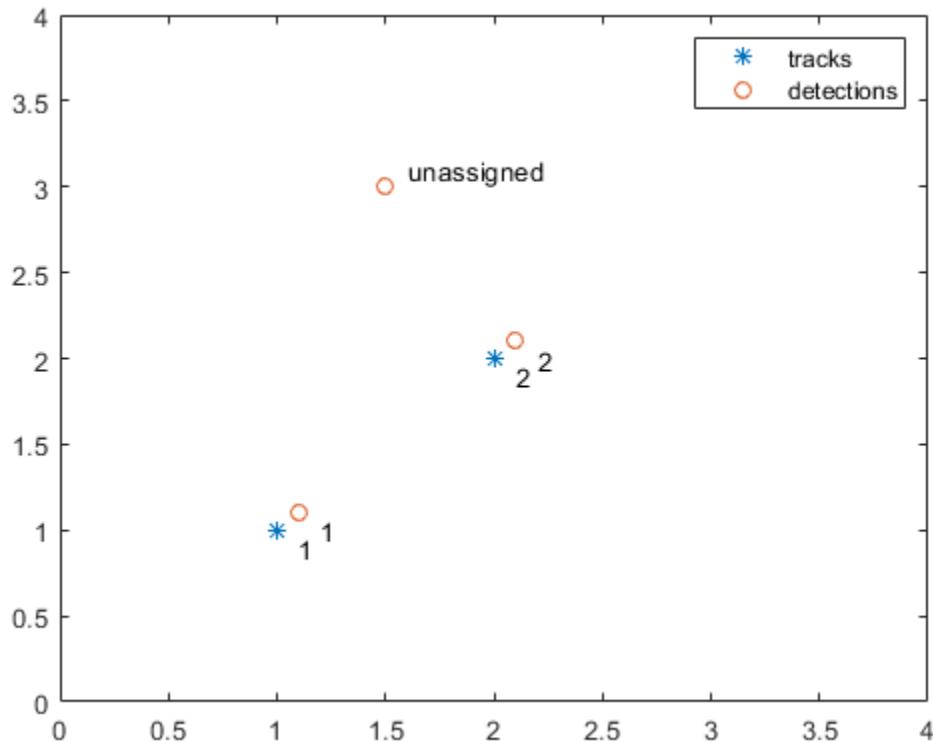
Display the unassigned detections.

```
disp(unassignedDetections)
```

```
3
```

Plot detection to track assignments.

```
plot(tracks(:, 1), tracks(:, 2), '*', dets(:, 1), dets(:, 2), 'o')
hold on
xlim([0, 4])
ylim([0, 4])
legend('tracks', 'detections')
assignStr = strsplit(num2str(1:size(assignments,1)));
text(tracks(assignments(:, 1),1) + 0.1, ...
    tracks(assignments(:, 1),2) - 0.1, assignStr);
text(dets(assignments(:, 2),1) + 0.1, ...
    dets(assignments(:, 2),2) - 0.1, assignStr);
text(dets(unassignedDetections(:,1) + 0.1, ...
    dets(unassignedDetections(:,2) + 0.1, 'unassigned');
```



The track to detection assignments are:

- 1** Detection 1 is assigned to track 1.
- 2** Detection 2 is assigned to track 2.
- 3** Detection 3 is not assigned.

## Input Arguments

### **costmatrix — Cost matrix**

real-valued  $M$ -by- $N$

Cost matrix, specified as an  $M$ -by- $N$  matrix.  $M$  is the number of tracks to be assigned and  $N$  is the number of detections to be assigned. Each entry in the cost matrix contains the cost of a track and detection assignment. The matrix may contain `Inf` entries to indicate that an assignment is prohibited. The cost matrix cannot be a sparse matrix.

Data Types: `single` | `double`

### **costofnonassignment — cost of non-assignment of tracks and detections**

scalar

Cost of non-assignment, specified as a scalar. The cost of non-assignment represents the cost of leaving tracks or detections unassigned. Higher values increase the likelihood that every object is assigned. The value cannot be set to `Inf`.

Data Types: single | double

## Output Arguments

### **assignments — Assignment of tracks to detections**

integer-valued  $L$ -by-2 matrix

Assignment of detections to track, returned as an integer-valued  $L$ -by-2 matrix where  $L$  is the number of assignments. The first column of the matrix contains the assigned track indices and the second column contains the assigned detection indices.

Data Types: uint32

### **unassignedrows — Indices of unassigned tracks**

integer-valued  $P$ -by-1 column vector

Indices of unassigned tracks, returned as an integer-valued  $P$ -by-1 column vector.

Data Types: uint32

### **unassignedcolumns — Indices of unassigned detections**

integer-valued  $Q$ -by-1 column vector

Indices of unassigned detections, returned as an integer-valued  $Q$ -by-1 column vector.

Data Types: uint32

## References

- [1] Samuel S. Blackman and Popoli, R. *Design and Analysis of Modern Tracking Systems*. Artech House: Norwood, MA. 1999.

## Extended Capabilities

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

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

## See Also

### **Functions**

`assignTOMHT` | `assignjv` | `assignkbest` | `assignkbestsd` | `assignmunkres` | `assignd` | `trackerGNN` | `trackerTOMHT`

**Introduced in R2018b**

# assignjv

Jonker-Volgenant global nearest neighbor assignment algorithm

## Syntax

```
[assignments,unassignedrows,unassignedcolumns] = assignjv(costmatrix,  
costofnonassignment)
```

## Description

[assignments,unassignedrows,unassignedcolumns] = assignjv(costmatrix, costofnonassignment) returns a table of assignments of detections to tracks using the Jonker-Volgenant algorithm. The JV algorithm finds an optimal solution to the global nearest neighbor (GNN) assignment problem by finding the set of assignments that minimize the total cost of the assignments. The Jonker-Volgenant algorithm solves the GNN assignment in two phases: begin with the auction algorithm and end with the Dijkstra shortest path algorithm.

The cost of each potential assignment is contained in the cost matrix, `costmatrix`. Each matrix entry represents the cost of a possible assignments. Matrix rows represent tracks and columns represent detections. All possible assignments are represented in the cost matrix. The lower the cost, the more likely the assignment is to be made. Each track can be assigned to at most one detection and each detection can be assigned to at most one track. If the number of rows is greater than the number of columns, some tracks are unassigned. If the number of columns is greater than the number of rows, some detections are unassigned. You can set an entry of `costmatrix` to Inf to prohibit an assignment.

`costofnonassignment` represents the cost of leaving tracks or detections unassigned. Higher values increase the likelihood that every existing object is assigned.

The function returns a list of unassigned tracks, `unassignedrows`, and a list of unassigned detections, `unassignedcolumns`.

## Examples

### Assign Detections to Tracks Using Jonker-Volgenant Algorithm

Use `assignjv` to assign three detections to two tracks.

Start with two predicted track locations in x-y coordinates.

```
tracks = [1,1; 2,2];
```

Assume three detections are received. At least one detection will not be assigned.

```
dets = [1.1, 1.1; 2.1, 2.1; 1.5, 3];
```

Construct a cost matrix by defining the cost of assigning a detection to a track as the Euclidean distance between them. Set the cost of non-assignment to 0.2.

```
for i = size(tracks,1):-1:1
    delta = dets - tracks(i,:);
    costMatrix(i,:) = sqrt(sum(delta .^ 2,2));
end
costofnonassignment = 0.2;
```

Use the Auction algorithm to assign detections to tracks.

```
[assignments, unassignedTracks, unassignedDetections] = ...
    assignjv(costMatrix,costofnonassignment);
```

Display the assignments.

```
disp(assignments)
```

```
1 1
2 2
```

Show that there are no unassigned tracks.

```
disp(unassignedTracks)
```

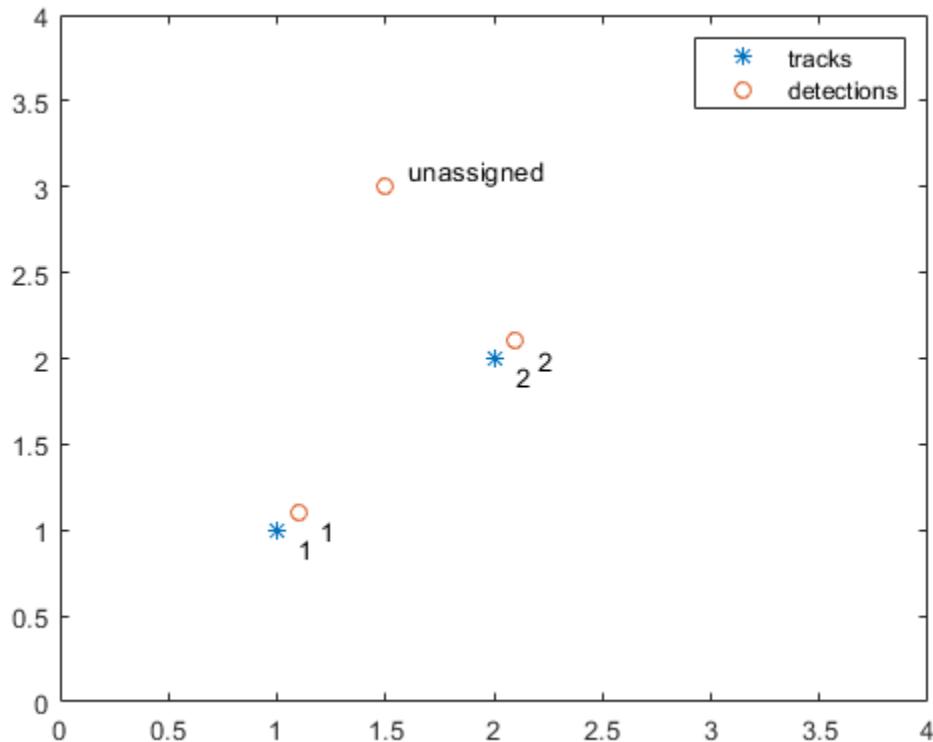
Display the unassigned detections.

```
disp(unassignedDetections)
```

```
3
```

Plot the detection to track assignments.

```
plot(tracks(:, 1), tracks(:, 2), '*', dets(:, 1), dets(:, 2), 'o')
hold on
xlim([0,4])
ylim([0,4])
legend('tracks', 'detections')
assignStr = strsplit(num2str(1:size(assignments,1)));
text(tracks(assignments(:,1),1) + 0.1, ...
    tracks(assignments(:,1),2) - 0.1, assignStr);
text(dets(assignments(:,2),1) + 0.1, ...
    dets(assignments(:,2),2) - 0.1, assignStr);
text(dets(unassignedDetections(:,1) + 0.1, ...
    dets(unassignedDetections(:,2) + 0.1, 'unassigned');
```



The track to detection assignments are:

- 1** Detection 1 is assigned to track 1.
- 2** Detection 2 is assigned to track 2.
- 3** Detection 3 is not assigned.

## Input Arguments

### **costmatrix — Cost matrix**

real-valued  $M$ -by- $N$

Cost matrix, specified as an  $M$ -by- $N$  matrix.  $M$  is the number of tracks to be assigned and  $N$  is the number of detections to be assigned. Each entry in the cost matrix contains the cost of a track and detection assignment. The matrix may contain `Inf` entries to indicate that an assignment is prohibited. The cost matrix cannot be a sparse matrix.

Data Types: `single` | `double`

### **costofnonassignment — cost of non-assignment of tracks and detections**

scalar

Cost of non-assignment, specified as a scalar. The cost of non-assignment represents the cost of leaving tracks or detections unassigned. Higher values increase the likelihood that every object is assigned. The value cannot be set to `Inf`.

Data Types: single | double

## Output Arguments

### **assignments — Assignment of tracks to detections**

integer-valued  $L$ -by-2 matrix

Assignment of detections to track, returned as an integer-valued  $L$ -by-2 matrix where  $L$  is the number of assignments. The first column of the matrix contains the assigned track indices and the second column contains the assigned detection indices.

Data Types: uint32

### **unassignedrows — Indices of unassigned tracks**

integer-valued  $P$ -by-1 column vector

Indices of unassigned tracks, returned as an integer-valued  $P$ -by-1 column vector.

Data Types: uint32

### **unassignedcolumns — Indices of unassigned detections**

integer-valued  $Q$ -by-1 column vector

Indices of unassigned detections, returned as an integer-valued  $Q$ -by-1 column vector.

Data Types: uint32

## References

- [1] Samuel S. Blackman and Popoli, R. *Design and Analysis of Modern Tracking Systems*. Artech House: Norwood, MA. 1999.

## Extended Capabilities

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

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

## See Also

### **Functions**

[assignTOMHT](#) | [assignauction](#) | [assignkbest](#) | [assignkbestsd](#) | [assignmunkres](#) | [assignsd](#) | [trackerGNN](#) | [trackerTOMHT](#)

**Introduced in R2018b**

# assignkbest

Assignment using k-best global nearest neighbor

## Syntax

```
[assignments,unassignedrows,unassignedcolumns,cost] = assignkbest(costmatrix,  
costofnonassignment)  
[assignments,unassignedrows,unassignedcolumns,cost] = assignkbest(costmatrix,  
costofnonassignment,k)  
[assignments,unassignedrows,unassignedcolumns,cost] = assignkbest(costmatrix,  
costofnonassignment,k,algorithm)
```

## Description

`[assignments,unassignedrows,unassignedcolumns,cost] = assignkbest(costmatrix, costofnonassignment)` returns a table of assignments, `assignments`, of detections to tracks using the Munkres algorithm. The algorithm finds the global nearest neighbor (GNN) solution that minimizes the total cost of the assignments.

The cost of each potential assignment is contained in the cost matrix, `costmatrix`. Each matrix entry represents the cost of a possible assignments. Matrix rows represent tracks and columns represent detections. All possible assignments are represented in the cost matrix. The lower the cost, the more likely the assignment is to be made. Each track can be assigned to at most one detection and each detection can be assigned to at most one track. If the number of rows is greater than the number of columns, some tracks are unassigned. If the number of columns is greater than the number of rows, some detections are unassigned. You can set an entry of `costmatrix` to Inf to prohibit an assignment.

`costofnonassignment` represents the cost of leaving tracks or detections unassigned. Higher values increase the likelihood that every existing object is assigned.

All inputs must all be single precision or all be double precision.

The function returns a list of unassigned tracks, `unassignedrows`, a list of unassigned detections, `unassignedcolumns`, and the cost of assignment, `cost`.

`[assignments,unassignedrows,unassignedcolumns,cost] = assignkbest(costmatrix, costofnonassignment,k)` also specifies the number, `k`, of  $k$ -best global nearest neighbor solutions that minimize the total cost of assignments. In addition to the best solution, the function uses the Murty algorithm to find the remaining  $k-1$  solutions.

`[assignments,unassignedrows,unassignedcolumns,cost] = assignkbest(costmatrix, costofnonassignment,k,algorithm)` also specifies the algorithm, `algorithm`, for finding the assignments.

## Examples

## Find Five Best Solutions Using Assignnkbest

Create a cost matrix containing prohibited assignments. Then, use the `assignnkbest` function to find the 5 best solutions.

Set up the cost matrix to contain some prohibited or invalid assignments by inserting `Inf` into the matrix.

```
costMatrix = [10 5 8 9; 7 Inf 20 Inf; Inf 21 Inf Inf; Inf 15 17 Inf; Inf inf 16 22];
costOfNonAssignment = 100;
```

Find the 5 best assignments.

```
[assignments,unassignedrows,unassignedcols,cost] = ...
assignnkbest(costMatrix,costOfNonAssignment,5)

assignments=5x1 cell array
{4x2 uint32}
{4x2 uint32}
{4x2 uint32}
{4x2 uint32}
{4x2 uint32}

unassignedrows=5x1 cell array
{[3]}
{[3]}
{[3]}
{[4]}
{[5]}

unassignedcols=5x1 cell array
{0x1 uint32}
{0x1 uint32}
{0x1 uint32}
{0x1 uint32}
{0x1 uint32}

cost = 5x1
147
151
152
153
154
```

## Input Arguments

### `costmatrix` – Cost matrix

real-valued  $M$ -by- $N$

Cost matrix, specified as an  $M$ -by- $N$  matrix.  $M$  is the number of tracks to be assigned and  $N$  is the number of detections to be assigned. Each entry in the cost matrix contains the cost of a track and

detection assignment. The matrix may contain `Inf` entries to indicate that an assignment is prohibited. The cost matrix cannot be a sparse matrix.

Data Types: `single` | `double`

**costofnonassignment — cost of non-assignment of tracks and detections**  
scalar

Cost of non-assignment, specified as a scalar. The cost of non-assignment represents the cost of leaving tracks or detections unassigned. Higher values increase the likelihood that every object is assigned. The value cannot be set to `Inf`.

Data Types: `single` | `double`

**k — Number of best solutions**  
positive integer

Number of best solutions, specified as a positive integer.

Data Types: `single` | `double`

**algorithm — Assignment algorithm**  
`'munkres'` (default) | `'jv'` | `'auction'`

Assignment algorithm, specified as `'munkres'` for the Munkres algorithm, `'jv'` for the Jonker-Volgenant algorithm, or `'auction'` for the Auction algorithm.

Example: `'jv'`

Data Types: `char` | `string`

## Output Arguments

**assignments — Assignment of tracks to detections**  
*k*-element cell array

Assignment of tracks to detections, returned as a *k*-element cell array. *k* is the number of best solutions. Each cell contains an  $L_i$ -by-2 matrix of pairs of track indices and assigned detection indices.  $L_i$  is the number of assignment pairs in the *i*<sup>th</sup> solution cell. The first column of each matrix contains the track indices and the second column contains the assigned detection indices.

**unassignedrows — Indices of unassigned tracks**  
*k*-element cell array

Indices of unassigned tracks, returned as a *k*-element cell array. Each cell is a  $P_i$  vector where  $P_i = M - L_i$  is the number of unassigned rows in the *i*<sup>th</sup> cell. Each element is the index of a row to which no columns are assigned. *k* is the number of best solutions.

Data Types: `uint32`

**unassignedcolumns — Indices of unassigned detections**  
*k*-element cell array

Indices of unassigned detections, returned as a *k*-element cell array. Each cell is a  $Q_i$  vector where  $Q_i = M - L_i$  is the number of unassigned detections in the *i*<sup>th</sup> cell. Each element is the index of a column to which no rows are assigned. *k* is the number of best solutions.

Data Types: uint32

**cost — Total cost of solutions**

$k$ -element vector (default)

Total cost of solutions, returned as a  $k$ -element vector. Each element is a scalar value summarizing the total cost of the solution to the assignment problem.

Data Types: single | double

## References

[1] Murty, Katta G. "An algorithm for ranking all the assignments in order of increasing cost." *Operations research* 16, no. 3 (1968): 682-687.

[2] Samuel Blackman and Robert Popoli. *Design and Analysis of Modern Tracking Systems*, Artech House, 1999.

## Extended Capabilities

**C/C++ Code Generation**

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

## See Also

**Functions**

[assignTOMHT](#) | [assignauction](#) | [assignjv](#) | [assignkbestsd](#) | [assignmunkres](#) | [assignd](#) | [trackerGNN](#) | [trackerTOMHT](#)

**Introduced in R2018b**

## assignkbestsd

K-best S-D solution that minimizes total cost of assignment

### Syntax

```
[assignments, cost, solutionGap] = assignkbestsd(costmatrix)
[assignments, cost, solutionGap] = assignkbestsd(costmatrix, k)
[assignments, cost, solutionGap] = assignkbestsd(costmatrix, k, desiredGap)
[assignments, cost, solutionGap] = assignkbestsd(costmatrix, k, desiredGap,
maxIterations)
[assignments, cost, solutionGap] = assignkbestsd(costmatrix, k, desiredGap,
maxIterations, algorithm)
```

### Description

[assignments, cost, solutionGap] = assignkbestsd(costmatrix) returns a table of assignments of detections to tracks by finding the best S-D solution that minimizes the total cost of the assignments. The algorithm uses Lagrangian relaxation to convert the S-D assignment problem to a corresponding 2-D assignment problem and then solves the 2-D problem. The cost of each potential assignment is contained in the cost matrix, `costmatrix`.

`costmatrix` is an n-dimensional cost matrix where `costmatrix(i, j, k, ...)` defines the cost of the n-tuple (*i, j, k, ...*) in assignment. The index '1' on all dimensions in `costmatrix` represents dummy measurement or a false track and is used to complete the assignment problem. The index 1, being a dummy, can be a part of multiple n-tuples. The index can be assigned more than once. A typical cost value for `costmatrix(1, 1, 1, 1, ...)` is 0.

The function also returns the solution gap, `solutionGap`, and the cost of assignments, `cost`.

[assignments, cost, solutionGap] = assignkbestsd(costmatrix, k) also specifies the number, *k* of *K*-best S-D solutions. The function finds *K* optimal solutions that minimize the total cost. First, the function finds the best solution. Then, the function uses the Murty algorithm to generate partitioned cost matrices. Finally, the function obtains the remaining *K* - 1 minimum cost solutions for each partitioned matrix.

[assignments, cost, solutionGap] = assignkbestsd(costmatrix, k, desiredGap) also specifies the desired maximum gap, `desiredGap`, between the dual solution and the feasible solution. The gap controls the quality of the solution. Values usually range from 0 to 1. A value of 0 means the dual and feasible solutions are the same.

[assignments, cost, solutionGap] = assignkbestsd(costmatrix, k, desiredGap, maxIterations) also specifies the maximum number of iterations allowed. The `desiredGap` and `maxIterations` arguments define the terminating conditions for the S-D algorithm.

[assignments, cost, solutionGap] = assignkbestsd(costmatrix, k, desiredGap, maxIterations, algorithm) also specifies the `algorithm` for finding the assignments.

### Examples

## Assign Detections to Tracks Using K-Best SD

Find the first 5 best assignments of the S-D assignment problem. Set the desired gap to 0.01 and the maximum number of iterations to 100.

Load the cost matrix.

```
load passiveAssociationCostMatrix.mat
```

Find the 5 best solutions.

```
[assignments,cost,solutionGap] = assignkbestsd(costMatrix,5,0.01,100)
```

```
assignments=5x1 cell array
{2x3 uint32}
{3x3 uint32}
{3x3 uint32}
{3x3 uint32}
{3x3 uint32}
```

```
cost = 5x1
```

```
-34.7000
-31.7000
-29.1000
-28.6000
-28.0000
```

```
solutionGap = 5x1
```

```
0
0.0552
0.0884
0.1075
0.1964
```

## Input Arguments

### **costmatrix — Cost matrix**

real-valued  $M$ -by- $N$

Cost matrix, specified as an  $n$ -dimensional array where `costmatrix(i,j,k, ...)` defines the cost of the  $n$ -tuple  $(i, j, k, \dots)$  in an assignment. The index '1' on all dimensions in `costmatrix` represents a dummy measurement or a false track and is used to complete the assignment problem. The index 1, being a dummy, can be a part of multiple  $n$ -tuples. The index can be assigned more than once. A typical cost value for `costmatrix(1,1,1,1, ...)` is 0.

Data Types: `single` | `double`

### **k — Number of best solutions**

1 (default) | positive integer

Number of best solutions, specified as a positive integer.

Data Types: `single` | `double`

**desiredGap — Desired maximal gap**

`0.01` (default) | nonnegative scalar

Desired maximum gap between the dual and feasible solutions, specified as a nonnegative scalar.

Example: `0.05`

Data Types: `single` | `double`

**maxIterations — Maximum number of iterations**

`100` (default) | positive integer

Maximum number of iterations, specified as a positive integer.

Example: `50`

Data Types: `single` | `double`

**algorithm — Assignment algorithm**

`'auction'` (default) | `'munkres'` | `'jv'`

Assignment algorithm for solving the 2-D assignment problem, specified as `'munkres'` for the Munkres algorithm, `'jv'` for the Jonker-Volgenant algorithm, or `'auction'` for the Auction algorithm.

Example: `'jv'`

## Output Arguments

**assignments — Assignment of tracks to detections**

*K*-element cell array

Assignments of tracks to detections, returned as a *K*-element cell array. Each cell is an *P*-by-*N* list of assignments. Assignments of the type [1 1 Q 1] from a four-dimensional cost matrix can be seen as a Q-1 entity from dimension 3 that was left unassigned. The cost value at (1,1,Q,1) defines the cost of not assigning the (Q-1)<sup>th</sup> entity from dimension 3.

**cost — Total cost of solutions**

*K*-element array

Total cost of solutions, returned as a *K*-element vector where *K* is the number of best solutions. Each element is a scalar value summarizing the total cost of the solution to the assignment problem.

Data Types: `single` | `double`

**solutionGap — Solution gap**

real-valued *K*-element array

Solution gap, returned as a positive-valued *K*-element array where *K* is the number of best solutions. Each element is the duality gap achieved between the feasible and dual solution. A gap value near zero indicates the quality of solution.

Data Types: `single` | `double`

## Algorithms

All numeric inputs can be single or double precision, but they all must have the same precision.

## References

- [1] Popp, R.L., Pattipati, K., and Bar Shalom, Y. "M-best S=D Assignment Algorithm with Application to Multitarget Tracking". IEEE Transactions on Aerospace and Electronic Systems, 37(1), 22-39. 2001.
- [2] Deb, S., Yeddanapudi, M., Pattipati, K., & Bar-Shalom, Y. (1997). "A generalized SD assignment algorithm for multisensor-multitarget state estimation". IEEE Transactions on Aerospace and Electronic Systems, 33(2), 523-538.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

[assignTOMHT](#) | [assignauction](#) | [assignjv](#) | [assignkbest](#) | [assignmunkres](#) | [assignd](#)

### System Objects

[trackerGNN](#) | [trackerTOMHT](#)

## Introduced in R2018b

# assignmunkres

Munkres global nearest neighbor assignment algorithm

## Syntax

```
[assignments,unassignedrows,unassignedcolumns] = assignmunkres(costmatrix,  
costofnonassignment)
```

## Description

[assignments,unassignedrows,unassignedcolumns] = assignmunkres(costmatrix, costofnonassignment) returns a table of assignments of detections to tracks using the Munkres algorithm. The Munkres algorithm obtains an optimal solution to the global nearest neighbor (GNN) assignment problem. An optimal solution minimizes the total cost of the assignments.

The cost of each potential assignment is contained in the cost matrix, `costmatrix`. Each matrix entry represents the cost of a possible assignments. Matrix rows represent tracks and columns represent detections. All possible assignments are represented in the cost matrix. The lower the cost, the more likely the assignment is to be made. Each track can be assigned to at most one detection and each detection can be assigned to at most one track. If the number of rows is greater than the number of columns, some tracks are unassigned. If the number of columns is greater than the number of rows, some detections are unassigned. You can set an entry of `costmatrix` to Inf to prohibit an assignment.

`costofnonassignment` represents the cost of leaving tracks or detections unassigned. Higher values increase the likelihood that every existing object is assigned.

The function returns a list of unassigned tracks, `unassignedrows`, and a list of unassigned detections, `unassignedcolumns`

## Examples

### Assign Detections to Tracks Using Munkres Algorithm

Use `assignMunkres` to assign three detections to two tracks.

Start with two predicted track locations in x-y coordinates.

```
tracks = [1,1; 2,2];
```

Assume three detections are received. At least one detection will not be assigned.

```
dets = [1.1, 1.1; 2.1, 2.1; 1.5, 3];
```

Construct a cost matrix by defining the cost of assigning a detection to a track as the Euclidean distance between them. Set the cost of non-assignment to 0.2.

```
for i = size(tracks, 1):-1:1  
    delta = dets - tracks(i, :);  
    costMatrix(i, :) = sqrt(sum(delta .^ 2, 2));
```

```
end
costofnonassignment = 0.2;
```

Use the Auction algorithm to assign detections to tracks.

```
[assignments, unassignedTracks, unassignedDetections] = ...
    assignmunkres(costMatrix,costofnonassignment);
```

Display the assignments.

```
disp(assignments)
```

```
1 1
2 2
```

Show that there are no unassigned tracks.

```
disp(unassignedTracks)
```

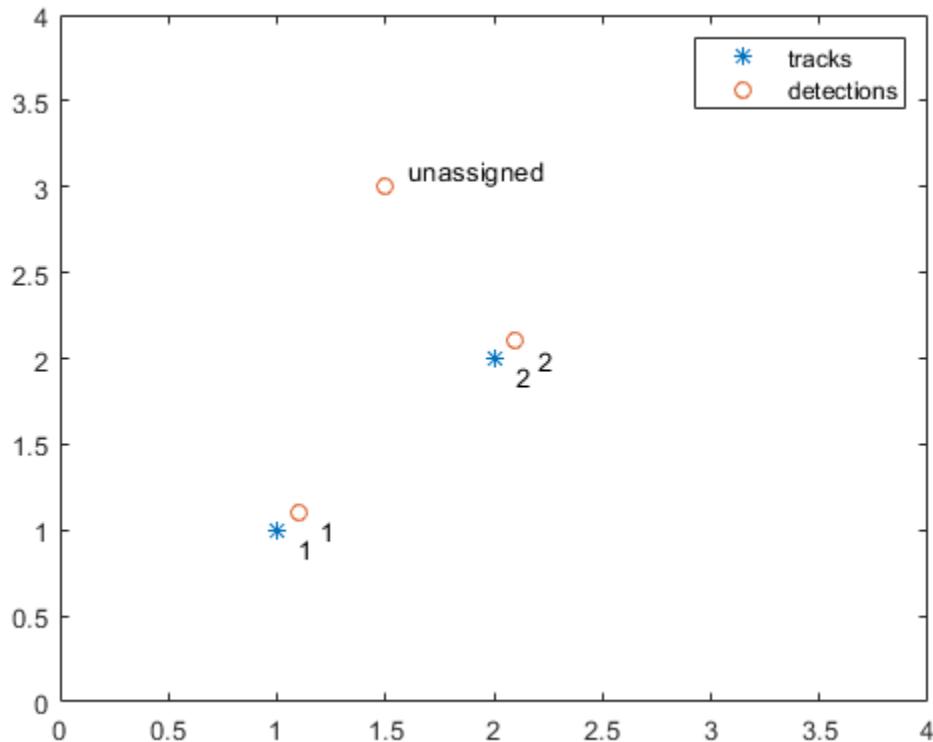
Display the unassigned detections.

```
disp(unassignedDetections)
```

```
3
```

Plot detection to track assignments.

```
plot(tracks(:, 1), tracks(:, 2), '*', dets(:, 1), dets(:, 2), 'o')
hold on
xlim([0, 4])
ylim([0, 4])
legend('tracks', 'detections')
assignStr = strsplit(num2str(1:size(assignments,1)));
text(tracks(assignments(:, 1),1) + 0.1, ...
    tracks(assignments(:, 1),2) - 0.1, assignStr);
text(dets(assignments(:, 2),1) + 0.1, ...
    dets(assignments(:, 2),2) - 0.1, assignStr);
text(dets(unassignedDetections(:,1) + 0.1, ...
    dets(unassignedDetections(:,2) + 0.1, 'unassigned');
```



The track to detection assignments are:

- 1** Detection 1 is assigned to track 1.
- 2** Detection 2 is assigned to track 2.
- 3** Detection 3 is not assigned.

## Input Arguments

### **costmatrix — Cost matrix**

real-valued  $M$ -by- $N$

Cost matrix, specified as an  $M$ -by- $N$  matrix.  $M$  is the number of tracks to be assigned and  $N$  is the number of detections to be assigned. Each entry in the cost matrix contains the cost of a track and detection assignment. The matrix may contain `Inf` entries to indicate that an assignment is prohibited. The cost matrix cannot be a sparse matrix.

Data Types: `single` | `double`

### **costofnonassignment — cost of non-assignment of tracks and detections**

scalar

Cost of non-assignment, specified as a scalar. The cost of non-assignment represents the cost of leaving tracks or detections unassigned. Higher values increase the likelihood that every object is assigned. The value cannot be set to `Inf`.

Data Types: single | double

## Output Arguments

### **assignments — Assignment of tracks to detections**

integer-valued  $L$ -by-2 matrix

Assignment of detections to track, returned as an integer-valued  $L$ -by-2 matrix where  $L$  is the number of assignments. The first column of the matrix contains the assigned track indices and the second column contains the assigned detection indices.

Data Types: uint32

### **unassignedrows — Indices of unassigned tracks**

integer-valued  $P$ -by-1 column vector

Indices of unassigned tracks, returned as an integer-valued  $P$ -by-1 column vector.

Data Types: uint32

### **unassignedcolumns — Indices of unassigned detections**

integer-valued  $Q$ -by-1 column vector

Indices of unassigned detections, returned as an integer-valued  $Q$ -by-1 column vector.

Data Types: uint32

## References

- [1] Samuel S. Blackman and Popoli, R. *Design and Analysis of Modern Tracking Systems*. Artech House: Norwood, MA. 1999.

## Extended Capabilities

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

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

## See Also

### **Functions**

[assignTOMHT](#) | [assignauction](#) | [assignjv](#) | [assignkbest](#) | [assignkbestsd](#) | [assignd](#) | [trackerGNN](#) | [trackerTOMHT](#)

**Introduced in R2018b**

# assignsd

S-D assignment using Lagrangian relaxation

## Syntax

```
[assignments, cost, solutionGap] = assignsd(costmatrix)
[assignments, cost, solutionGap] = assignsd(costmatrix, desiredGap)
[assignments, cost, solutionGap] = assignsd(costmatrix, desiredGap,
maxIterations)
[assignments, cost, solutionGap] = assignsd(costmatrix, desiredGap,
maxIterations, algorithm)
```

## Description

`[assignments, cost, solutionGap] = assignsd(costmatrix)` returns a table of assignments, `assignments`, of detections to tracks by finding a suboptimal solution to the S-D assignment problem using Lagrangian relaxation. The cost of each potential assignment is contained in the cost matrix, `costmatrix`. The algorithm terminates when the gap reaches below 0.01 (1 percent) or if the number of iterations reaches 100.

`costmatrix` is an n-dimensional cost matrix where `costmatrix(i, j, k, ...)` defines the cost of the n-tuple (`i, j, k, ...`) in assignment. The index '1' on all dimensions in `costmatrix` represents dummy measurement or a false track and is used to complete the assignment problem. The index 1, being a dummy, can be a part of multiple n-tuples. The index can be assigned more than once. A typical cost value for `costmatrix(1, 1, 1, 1, ...)` is 0.

All inputs can be single or double precision, but they all must be of the same precision.

The function also returns the solution gap, `solutionGap`, and the total cost of assignments, `cost`.

`[assignments, cost, solutionGap] = assignsd(costmatrix, desiredGap)` also specifies the desired maximum gap, `desiredGap`, between the dual and the feasible solutions as a scalar. The gap controls the quality of the solution. Values usually range from 0 to 1. A value of 0 means the dual and feasible solutions are the same.

`[assignments, cost, solutionGap] = assignsd(costmatrix, desiredGap, maxIterations)` also specifies the maximum number of iterations, `maxIterations`.

`[assignments, cost, solutionGap] = assignsd(costmatrix, desiredGap, maxIterations, algorithm)` also specifies the assignment algorithm, `algorithm`.

## Examples

### Assign Detections to Tracks Using `assignsd` Algorithm

Use `assignsd` to perform strict assignment without index 1.

Not having dummy index means that no entity is left unassigned. Therefore, define the cost matrix to be equi-dimensional.

```
costMatrix = rand(6,6,6);
```

Initialize the `fullmatrix` to all Inf. The `fullmatrix` is one size larger than the cost matrix in all dimensions.

```
fullMatrix = inf(7,7,7);
```

Set the inner matrix to `costMatrix` to force the assignments involving index 1 to have infinite cost.

```
fullMatrix(2:end,2:end,2:end) = costMatrix;
fullMatrix(1,1,1) = 0;
[assignments,cost,gapAchieved] = assignd(fullMatrix,0.01,100);
```

Restore the actual indices.

```
assignments = assignments - 1
assignments = 6x3 uint32 matrix
```

1	6	6
2	4	3
3	3	4
4	1	2
5	2	1
6	5	5

## Input Arguments

### **costmatrix — Cost matrix**

real-valued  $M$ -by- $N$

Cost matrix, specified as an  $n$ -dimensional array where `costmatrix(i,j,k, ...)` defines the cost of the  $n$ -tuple  $(i, j, k, \dots)$  in an assignment. The index '1' on all dimensions in `costmatrix` represents a dummy measurement or a false track and is used to complete the assignment problem. The index 1, being a dummy, can be a part of multiple  $n$ -tuples. The index can be assigned more than once. A typical cost value for `costmatrix(1,1,1,1, ...)` is 0.

Data Types: `single` | `double`

### **desiredGap — Desired maximal gap**

0.01 (default) | nonnegative scalar

Desired maximum gap between the dual and feasible solutions, specified as a nonnegative scalar.

Example: 0.05

Data Types: `single` | `double`

### **maxIterations — Maximum number of iterations**

100 (default) | positive integer

Maximum number of iterations, specified as a positive integer.

Example: 50

Data Types: `single` | `double`

**algorithm — Assignment algorithm**  
'auction' (default) | 'munkres' | 'jv'

Assignment algorithm for solving the 2-D assignment problem, specified as 'munkres' for the Munkres algorithm, 'jv' for the Jonker-Volgenant algorithm, or 'auction' for the Auction algorithm.

Example: 'jv'

## Output Arguments

**assignments — Assignment of tracks to detections**  
*P*-by-*N* matrix

Assignments of tracks to detections, returned as a *P*-by-*N* list of assignments. Assignments of the type [1 1 Q 1] from a four-dimensional cost matrix can be seen as a Q-1 entity from dimension 3 that was left unassigned. The cost value at (1, 1, Q, 1) defines the cost of not assigning the (Q-1)<sup>th</sup> entity from dimension 3.

**cost — Total cost of assignment solution**  
positive scalar

Total cost of solutions, returned as a *K*-element vector where *K* is the number of best solutions. Each element is a scalar value summarizing the total cost of the solution to the assignment problem.

Data Types: single | double

**solutionGap — Solution gap**  
positive scalar (default)

Solution gap, returned as a positive scalar. The solution gap is the duality gap achieved between the feasible and dual solution. A gap value near zero indicates the quality of solution.

Data Types: single | double

## Algorithms

- The Lagrangian relaxation method computes a suboptimal solution to the S-D assignment problem. The method relaxes the S-D assignment problem to a 2-D assignment problem using a set of Lagrangian multipliers. The relaxed 2-D assignment problem is commonly known as the dual problem, which can be solved optimally using algorithms like the Munkres algorithm. Constraints are then enforced on the dual solution by solving multiple 2-D assignment problems to obtain a feasible solution to the original problem. The cost of the dual solution and the feasible solution serves as lower and upper bounds on the optimal cost, respectively. The algorithm iteratively tries to minimize the gap between the dual and feasible solutions, commonly known as the dual gap. The iteration stops when the dual gap is below a desired gap or the maximum number of iterations have reached.
- When using the auction algorithm, the `assignsd` function uses the Heuristic Price Update algorithm to update the Lagrangian multipliers. When using the Munkres and JV algorithms, the function uses the Accelerated Subgradient Update algorithm.
- For cost matrices with well-defined solutions, such as passive association with high-precision sensors, the solution gap converges to within 0.05 (5 percent) in approximately 100 iterations.

- As the optimal solution is unknown, the solution gap can be non-zero even when the returned solution is optimal.

## References

- [1] Deb, S., Yeddanapudi, M., Pattipati, K., and Bar-Shalom, Y. (1997). *A generalized SD assignment algorithm for multisensor-multitarget state estimation*. IEEE Transactions on Aerospace and Electronic Systems, 33(2), 523-538.
- [2] Blackman, Samuel, and Robert Popoli. *Design and analysis of modern tracking systems*. Norwood, MA: Artech House, 1999. (1999)

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

`assignTOMHT` | `assignauction` | `assignjv` | `assignkbest` | `assignkbestsd` | `assignmunkres` | `trackerGNN` | `trackerTOMHT`

### Introduced in R2018b

# assignTOMHT

Track-oriented multi-hypotheses tracking assignment

## Syntax

```
[assignments,unassignedrows,unassignedcolumns] = assignTOMHT(costmatrix,  
costThreshold)
```

## Description

[assignments,unassignedrows,unassignedcolumns] = assignTOMHT(costmatrix, costThreshold) returns a table of assignments, `assignments`, of detections to tracks using a track-oriented multi-hypothesis algorithm (TOMHT).

The cost of each potential assignment is contained in the cost matrix, `costmatrix`. Each matrix entry represents the cost of a possible assignments. Matrix rows represent tracks and columns represent detections. All possible assignments are represented in the cost matrix. The lower the cost, the more likely the assignment is to be made. Each track can be assigned to at most one detection and each detection can be assigned to at most one track. If the number of rows is greater than the number of columns, some tracks are unassigned. If the number of columns is greater than the number of rows, some detections are unassigned. You can set an entry of `costmatrix` to Inf to prohibit an assignment.

`costThreshold` represents the set of three gates used for assigning detections to tracks.

The function returns a list of unassigned tracks, `unassignedrows`, and a list of unassigned detections, `unassignedcolumns`.

## Examples

### Assignment Using AssignTOMHT

Find the assignments from a cost matrix using `assignTOMHT` with a nonzero C1 gate and a nonzero C2 gate.

Create a cost matrix that assigns:

- Track 1 to detection 1 within the C1 gate and detection 2 within the C2 gate.
- Track 2 to detection 2 within the C2 gate and detection 3 within the C3 gate.
- Track 3 is unassigned.
- Detection 4 is unassigned.

```
costMatrix = [4 9 200 Inf; 300 12 28 Inf; 32 100 210 1000];  
costThresh = [5 10 30];
```

Calculate the assignments.

```
[assignments, unassignedTracks, unassignedDets] = assignTOMHT(costMatrix,costThresh)
```

```

assignments = 4x2 uint32 matrix

1 1
1 2
2 2
2 3

unassignedTracks = 2x1 uint32 column vector

2
3

unassignedDets = 2x1 uint32 column vector

3
4

```

Tracks that are assigned detections within the C1 gate are not considered as unassigned. For example, track 1. Detections that are assigned to tracks within the C2 gate are not considered as unassigned. For example, detections 1 and 2.

## Input Arguments

### **costmatrix — Cost matrix**

real-valued  $M$ -by- $N$

Cost matrix, specified as an  $M$ -by- $N$  matrix.  $M$  is the number of tracks to be assigned and  $N$  is the number of detections to be assigned. Each entry in the cost matrix contains the cost of a track and detection assignment. The matrix may contain `Inf` entries to indicate that an assignment is prohibited. The cost matrix cannot be a sparse matrix.

Data Types: `single` | `double`

### **costThreshold — Assignment gates**

positive, real-valued 3-element vector

Assignment gates, specified as a positive, real-valued three-element vector `[c1gate, c2gate, c3gate]` where `c1gate <= c2gate <= c3gate`.

Example: `[0.1, 0.3, 0.5]`

Data Types: `single` | `double`

## Output Arguments

### **assignments — Assignment of tracks to detections**

integer-valued  $L$ -by-2 matrix

Assignment of detections to track, returned as an integer-valued  $L$ -by-2 matrix where  $L$  is the number of assignments. The first column of the matrix contains the assigned track indices and the second column contains the assigned detection indices.

Data Types: `uint32`

**unassignedrows — Indices of unassigned tracks**

integer-valued  $P$ -by-1 column vector

Indices of unassigned tracks, returned as an integer-valued  $P$ -by-1 column vector.

Data Types: uint32

**unassignedcolumns — Indices of unassigned detections**

integer-valued  $Q$ -by-1 column vector

Indices of unassigned detections, returned as an integer-valued  $Q$ -by-1 column vector.

Data Types: uint32

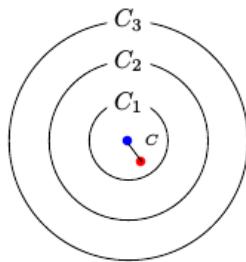
## Algorithms

### Assignment Thresholds for Multi-Hypothesis Tracker

Three assignment thresholds,  $C_1$ ,  $C_2$ , and  $C_3$ , control (1) the assignment of a detection to a track, (2) the creation of a new branch from a detection, and (3) the creation of a new branch from an unassigned track. The threshold values must satisfy:  $C_1 \leq C_2 \leq C_3$ .

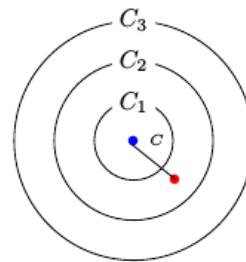
If the cost of an assignment is  $C = \text{costmatrix}(i, j)$ , the following hypotheses are created based on comparing the cost to the values of the assignment thresholds. Below each comparison, there is a list of the possible hypotheses.

● Track  
● Detection



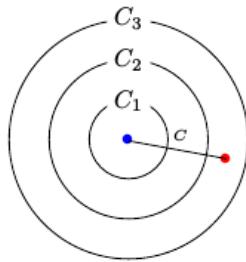
$C \leq C_1$   
Single Hypothesis

- (1) Detection is assigned to track. A branch is created updating the track with this detection.



$C_1 < C \leq C_2$   
Two Hypotheses

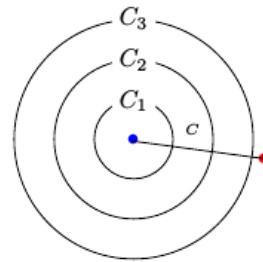
- (1) Detection is assigned to track. A branch is created updating the track with this detection.  
(2) Track is not assigned to detection and is coasted.



$C_2 < C \leq C_3$

Three Hypotheses

- (1) Detection is assigned to track. A branch is created updating the track with this detection.  
(2) Track is not assigned to detection and is coasted.  
(3) Detection is not assigned and creates a new track (branch).



$C_3 < C$

Single Hypothesis

- (1) Detection is not assigned and creates a new track (branch).

Tips:

- Increase the value of  $C_3$  if there are detections that should be assigned to tracks but are not. Decrease the value if there are detections that are assigned to tracks they should not be assigned to (too far away).
- Increasing the values  $C_1$  and  $C_2$  helps control the number of track branches that are created. However, doing so reduces the number of branches (hypotheses) each track has.
- To allow each track to be unassigned, set  $C_1 = 0$ .
- To allow each detection to be unassigned, set  $C_2 = 0$ .

## References

- [1] Werthmann, John R. "Step-by-step description of a computationally efficient version of multiple hypothesis tracking." In Signal and Data Processing of Small Targets 1992, vol. 1698, pp. 288-300. International Society for Optics and Photonics, 1992.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

`assignauction | assignjv | assignkbest | assignkbestsd | assignmunkres | assignsd | trackerGNN | trackerTOMHT`

**Introduced in R2018b**

# fusecovint

Covariance fusion using covariance intersection

## Syntax

```
[fusedState,fusedCov] = fusecovint(trackState,trackCov)
[fusedState,fusedCov] = fusecovint(trackState,trackCov,minProp)
```

## Description

[fusedState,fusedCov] = `fusecovint(trackState,trackCov)` fuses the track states in `trackState` and their corresponding covariance matrices `trackCov`. The function computes the fused state and covariance as an intersection of the individual covariances. It creates a convex combination of the covariances and finds weights that minimize the determinant of the fused covariance matrix.

[fusedState,fusedCov] = `fusecovint(trackState,trackCov,minProp)` estimates the fused covariance by minimizing `minProp`, which can be either the determinant or the trace of the fused covariance matrix.

## Examples

### Covariance Intersection Fusion Using Default Values

Define a state vector of tracks.

```
x(:,1) = [1;2;0];
x(:,2) = [2;2;0];
x(:,3) = [2;3;0];
```

Define the covariance matrices of the tracks.

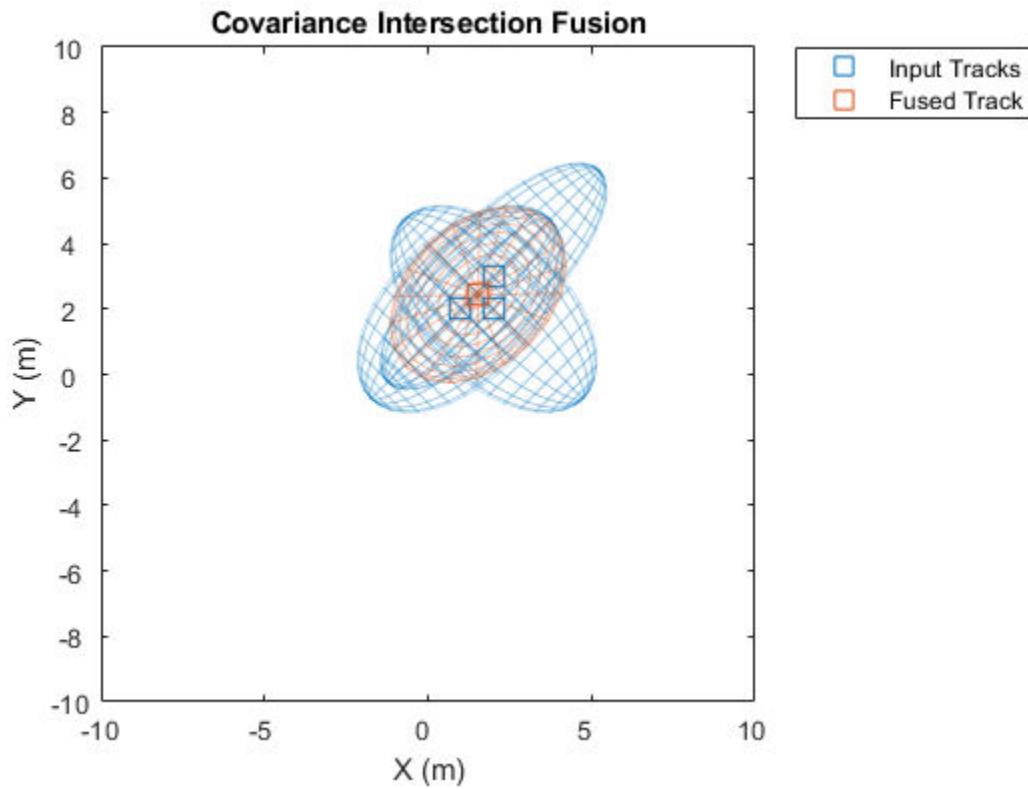
```
p(:,:1) = [10 5 0; 5 10 0; 0 0 1];
p(:,:2) = [10 -5 0; -5 10 0; 0 0 1];
p(:,:3) = [12 9 0; 9 12 0; 0 0 1];
```

Estimate the fused state vector and its covariance.

```
[fusedState,fusedCov] = fusecovint(x,p);
```

Use `trackPlotter` to plot the results.

```
tPlotter = theaterPlot('XLim',[-10 10], 'YLim', [-10 10], 'ZLim', [-10 10]);
tPlotter1 = trackPlotter(tPlotter, ...
    'DisplayName','Input Tracks','MarkerEdgeColor',[0.000 0.447 0.741]);
tPlotter2 = trackPlotter(tPlotter,'DisplayName', ...
    'Fused Track','MarkerEdgeColor',[0.850 0.325 0.098]);
plotTrack(tPlotter1,x',p)
plotTrack(tPlotter2,fusedState',fusedCov)
title('Covariance Intersection Fusion')
```



### Covariance Intersection Fusion Using Trace Minimization

Define a state vector of tracks.

```
x(:,1) = [1;2;0];
x(:,2) = [2;2;0];
x(:,3) = [2;3;0];
```

Define the covariance matrices of the tracks.

```
p(:,:1) = [10 5 0; 5 10 0; 0 0 1];
p(:,:2) = [10 -5 0; -5 10 0; 0 0 1];
p(:,:3) = [12 9 0; 9 12 0; 0 0 1];
```

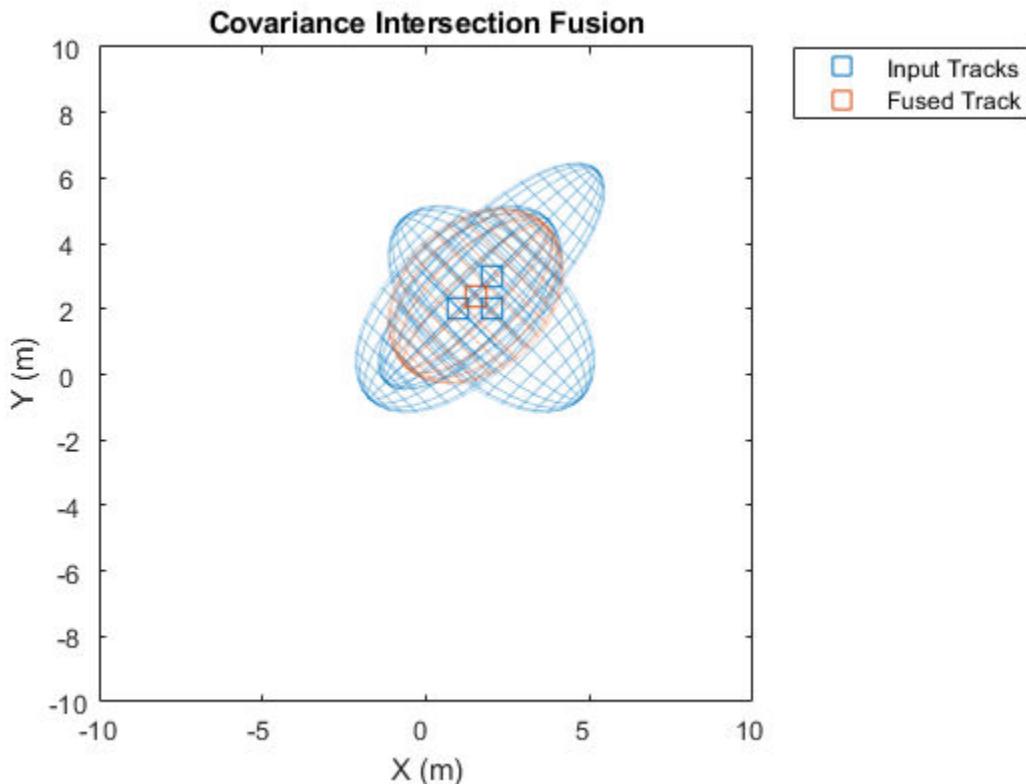
Estimate the fused state vector and its covariance. Combine the original covariances so that the trace of the fused covariance matrix is minimized.

```
[fusedState,fusedCov] = fusecovint(x,p,'trace');
```

Use `trackPlotter` to plot the results.

```
tPlotter = theaterPlot('XLim',[-10 10],'YLim',[-10 10],'ZLim',[-10 10]);
tPlotter1 = trackPlotter(tPlotter, ...
    'DisplayName','Input Tracks','MarkerEdgeColor',[0.000 0.447 0.741]);
tPlotter2 = trackPlotter(tPlotter, ...
```

```
'DisplayName','Fused Track','MarkerEdgeColor',[0.850 0.325 0.098]);
plotTrack(tPlotter1,x',p)
plotTrack(tPlotter2,fusedState',fusedCov)
title('Covariance Intersection Fusion')
```



## Input Arguments

### **trackState — Track states**

*N*-by-*M* matrix

Track states, specified as an *N*-by-*M* matrix, where *N* is the dimension of the state and *M* is the number of tracks.

Data Types: single | double

### **trackCov — Track covariance matrices**

*N*-by-*N*-by-*M* array

Track covariance matrices, specified as an *N*-by-*N*-by-*M* array, where *N* is the dimension of the state and *M* is the number of tracks.

Data Types: single | double

### **minProp — Property to minimize**

'det' (default) | 'trace'

Property to minimize when estimating the fused covariance, specified as 'det' or 'trace'.

Data Types: char | string

## Output Arguments

### fusedState — Fused state

*N*-by-1 vector

Fused state, returned as an *N*-by-1 vector, where *N* is the dimension of the state.

### fusedCov — Fused covariance matrix

*N*-by-*N* matrix

Fused covariance matrix, returned as an *N*-by-*N* matrix, where *N* is the dimension of the state.

## References

- [1] Matzka, Stephan, and Richard Altendorfer. "A comparison of track-to-track fusion algorithms for automotive sensor fusion." In Multisensor Fusion and Integration for Intelligent Systems, pp. 69-81. Springer, Berlin, Heidelberg, 2009.
- [2] Julier, Simon, and Jeffrey K. Uhlmann. "General decentralized data fusion with covariance intersection." In Handbook of multisensor data fusion, pp. 339-364. CRC Press, 2017.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

[fusecovunion](#) | [fusexcov](#)

## Introduced in R2018b

# fusecovunion

Covariance fusion using covariance union

## Syntax

```
[fusedState,fusedCov] = fusecovunion(trackState,trackCov)
```

## Description

[fusedState,fusedCov] = fusecovunion(trackState,trackCov) fuses the track states in `trackState` and their corresponding covariance matrices `trackCov`. The function estimates the fused state and covariance in a way that maintains consistency. For more details, see “Consistent Estimator” on page 1-451.

## Examples

### Covariance Union Fusion

Define a state vector of tracks.

```
x(:,1) = [1;2;0];
x(:,2) = [2;2;0];
x(:,3) = [2;3;0];
```

Define the covariance matrices of the tracks.

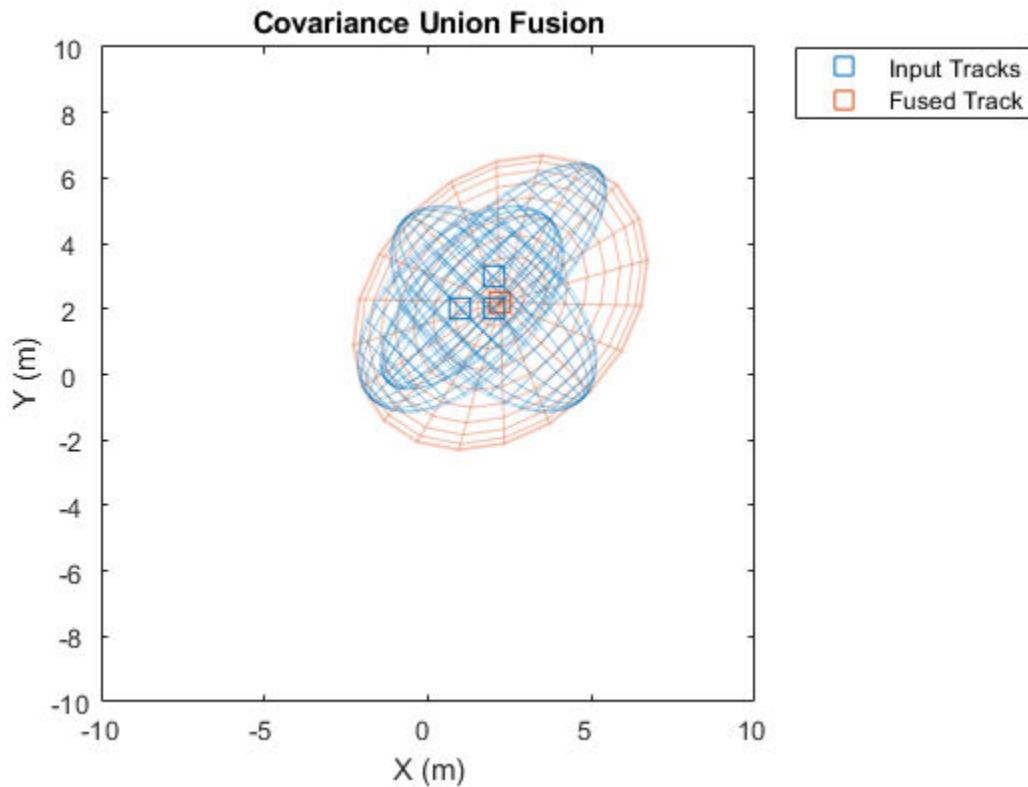
```
p(:,:,:,1) = [10 5 0; 5 10 0; 0 0 1];
p(:,:,:,2) = [10 -5 0; -5 10 0; 0 0 1];
p(:,:,:,3) = [12 9 0; 9 12 0; 0 0 1];
```

Estimate the fused state vector and its covariance.

```
[fusedState,fusedCov] = fusecovunion(x,p);
```

Use `trackPlotter` to plot the results.

```
tPlotter = theaterPlot('XLim',[-10 10], 'YLim',[-10 10], 'ZLim',[-10 10]);
tPlotter1 = trackPlotter(tPlotter, ...
    'DisplayName','Input Tracks', 'MarkerEdgeColor',[0.000 0.447 0.741]);
tPlotter2 = trackPlotter(tPlotter, ...
    'DisplayName','Fused Track', 'MarkerEdgeColor',[0.850 0.325 0.098]);
plotTrack(tPlotter1,x',p)
plotTrack(tPlotter2, fusedState', fusedCov)
title('Covariance Union Fusion')
```



## Input Arguments

### **trackState — Track states**

*N*-by-*M* matrix

Track states, specified as an *N*-by-*M* matrix, where *N* is the dimension of the state and *M* is the number of tracks.

Data Types: `single` | `double`

### **trackCov — Track covariance matrices**

*N*-by-*N*-by-*M* array

Track covariance matrices, specified as an *N*-by-*N*-by-*M* array, where *N* is the dimension of the state and *M* is the number of tracks.

Data Types: `single` | `double`

## Output Arguments

### **fusedState — Fused state**

*N*-by-1 vector

Fused state, returned as an *N*-by-1 vector, where *N* is the dimension of the state.

**fusedCov — Fused covariance matrix**

*N*-by-*N* matrix

Fused covariance matrix, returned as an *N*-by-*N* matrix, where *N* is the dimension of the state.

## More About

### Consistent Estimator

A *consistent estimator* is an estimator that converges in probability to the quantity being estimated as the sample size grows. In the case of tracking, a position estimate is consistent if its covariance (error) matrix is not smaller than the covariance of the actual distribution of the true state about the estimate. The covariance union method guarantees consistency by ensuring that all the individual means and covariances are bounded by the fused mean and covariance.

## References

- [1] Reece, Steven, and Stephen Rogers. "Generalised Covariance Union: A Unified Approach to Hypothesis Merging in Tracking." *IEEE® Transactions on Aerospace and Electronic Systems*. Vol. 46, No. 1, Jan. 2010, pp. 207-221.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[fusecovint](#) | [fusexcov](#)

### Introduced in R2018b

## fusexcov

Covariance fusion using cross-covariance

### Syntax

```
[fusedState,fusedCov] = fusexcov(trackState,trackCov)
[fusedState,fusedCov] = fusexcov(trackState,trackCov,crossCovFactor)
```

### Description

[fusedState,fusedCov] = fusexcov(trackState,trackCov) fuses the track states in trackState and their corresponding covariance matrices trackCov. The function estimates the fused state and covariance within a Bayesian framework in which the cross-correlation between tracks is unknown.

[fusedState,fusedCov] = fusexcov(trackState,trackCov,crossCovFactor) specifies a cross-covariance factor for the effective correlation coefficient when computing the cross-covariance.

### Examples

#### Cross-Covariance Fusion Using Default Values

Define a state vector of tracks.

```
x(:,1) = [1;2;0];
x(:,2) = [2;2;0];
x(:,3) = [2;3;0];
```

Define the covariance matrices of the tracks.

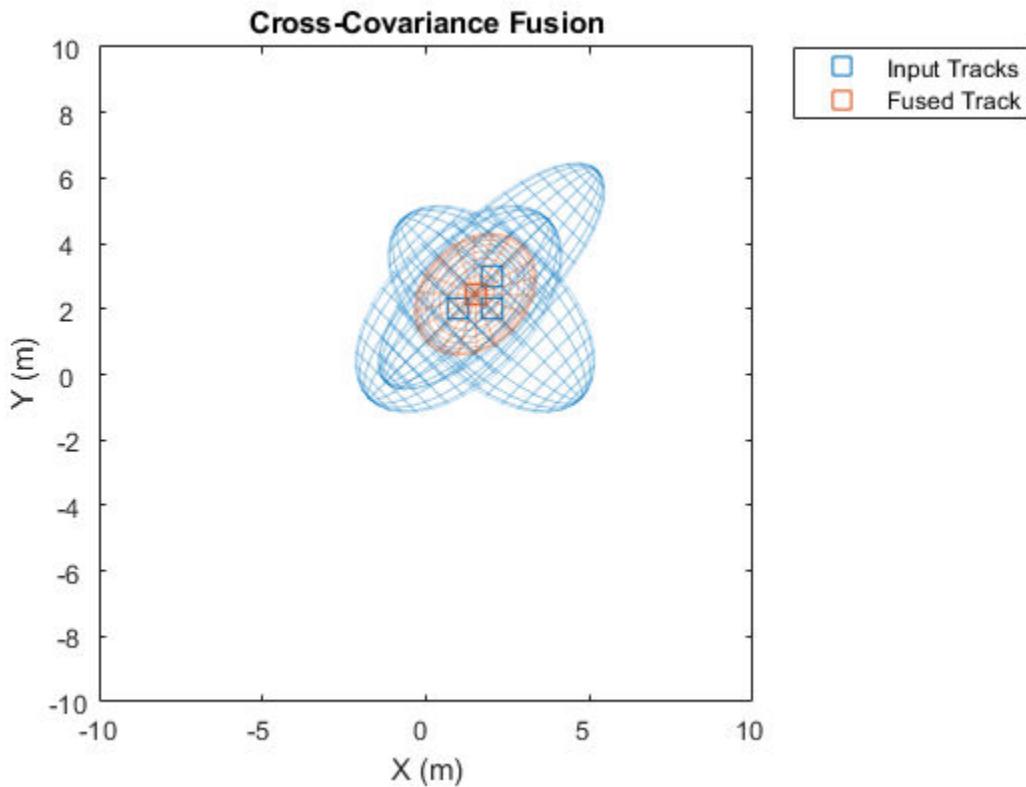
```
p(:,:1) = [10 5 0; 5 10 0; 0 0 1];
p(:,:2) = [10 -5 0; -5 10 0; 0 0 1];
p(:,:3) = [12 9 0; 9 12 0; 0 0 1];
```

Estimate the fused state vector and its covariance.

```
[fusedState,fusedCov] = fusexcov(x,p);
```

Use `trackPlotter` to plot the results.

```
tPlotter = theaterPlot('XLim',[-10 10], 'YLim', [-10 10], 'ZLim', [-10 10]);
tPlotter1 = trackPlotter(tPlotter, ...
    'DisplayName', 'Input Tracks', 'MarkerEdgeColor', [0.000 0.447 0.741]);
tPlotter2 = trackPlotter(tPlotter, ...
    'DisplayName', 'Fused Track', 'MarkerEdgeColor', [0.850 0.325 0.098]);
plotTrack(tPlotter1,x',p)
plotTrack(tPlotter2, fusedState', fusedCov)
title('Cross-Covariance Fusion')
```



### Cross-Covariance Fusion Using Cross-Covariance Factor

Define a state vector of tracks.

```
x(:,1) = [1;2;0];
x(:,2) = [2;2;0];
x(:,3) = [2;3;0];
```

Define the covariance matrices of the tracks.

```
p(:,:1) = [10 5 0; 5 10 0; 0 0 1];
p(:,:2) = [10 -5 0; -5 10 0; 0 0 1];
p(:,:3) = [12 9 0; 9 12 0; 0 0 1];
```

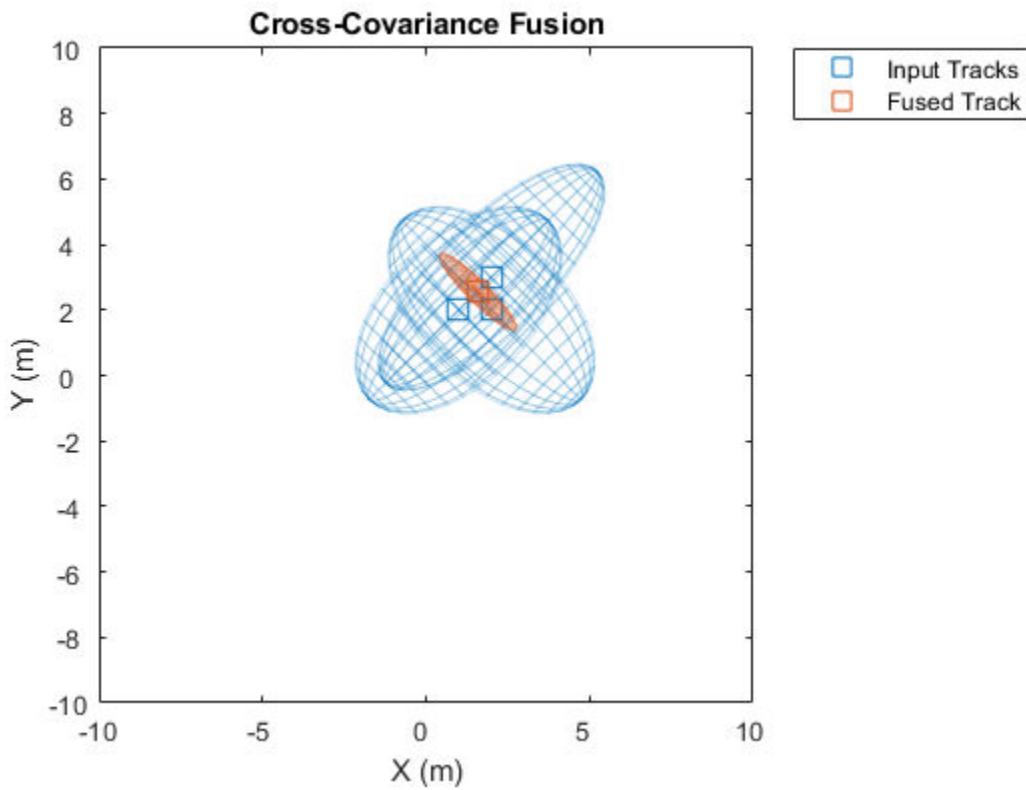
Estimate the fused state vector and its covariance. Specify a cross-covariance factor of 0.5.

```
[fusedState,fusedCov] = fusexcov(x,p,0.5);
```

Use `trackPlotter` to plot the results.

```
tPlotter = theaterPlot('XLim',[-10 10], 'YLim', [-10 10], 'ZLim', [-10 10]);
tPlotter1 = trackPlotter(tPlotter, ...
    'DisplayName','Input Tracks','MarkerEdgeColor',[0.000 0.447 0.741]);
tPlotter2 = trackPlotter(tPlotter, ...
    'DisplayName','Fused Track','MarkerEdgeColor',[0.850 0.325 0.098]);
```

```
plotTrack(tPlotter1,x',p)
plotTrack(tPlotter2, fusedState', fusedCov)
title('Cross-Covariance Fusion')
```



## Input Arguments

### **trackState — Track states**

*N*-by-*M* matrix

Track states, specified as an *N*-by-*M* matrix, where *N* is the dimension of the state and *M* is the number of tracks.

Data Types: single | double

### **trackCov — Track covariance matrices**

*N*-by-*N*-by-*M* array

Track covariance matrices, specified as an *N*-by-*N*-by-*M* array, where *N* is the dimension of the state and *M* is the number of tracks.

Data Types: single | double

### **crossCovFactor — Cross-covariance factor**

0 .4 (default) | scalar

Cross-covariance factor, specified as a scalar.

Data Types: `single` | `double`

## Output Arguments

### **fusedState — Fused state**

$N$ -by-1 vector

Fused state, returned as an  $N$ -by-1 vector, where  $N$  is the dimension of the state.

### **fusedCov — Fused covariance matrix**

$N$ -by- $N$  matrix

Fused covariance matrix, returned as an  $N$ -by- $N$  matrix, where  $N$  is the dimension of the state.

## References

- [1] Bar-Shalom, Yaakov, and Xiao-Rong Li. *Multitarget-multisensor tracking: principles and techniques*. Vol. 19. Storrs, CT: YBs, 1995.
- [2] Weng, Zhiyuan, and Petar M. Djurić. "A bayesian approach to covariance estimation and data fusion." In 2012 Proceedings of the 20th European Signal Processing Conference , pp. 2352-2356. IEEE, 2012.
- [3] Matzka, Stephan, and Richard Altendorfer. "A comparison of track-to-track fusion algorithms for automotive sensor fusion." In Multisensor Fusion and Integration for Intelligent Systems, pp. 69-81. Springer, Berlin, Heidelberg, 2009.

## Extended Capabilities

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

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

## See Also

`fusecovint` | `fusecovunion`

## Introduced in R2018b

# clusterTrackBranches

Cluster track-oriented multi-hypothesis history

## Syntax

```
[clusters,incompatibleBranches] = clusterTrackBranches(branchHistory)
[clusters,incompatibleBranches] = clusterTrackBranches(
branchHistory,'OutputForm',out)
```

## Description

`[clusters,incompatibleBranches] = clusterTrackBranches(branchHistory)` computes the clusters and incompatibility matrix for a set of branches.

Branches  $i$ ,  $j$ , and  $k$  belong to the same cluster if branches  $i$  and  $j$  are pairwise-incompatible and branches  $j$  and  $k$  are pairwise-incompatible. Two branches are pairwise-incompatible if they share a track ID (the first column of `branchHistory`) or if they share detections that fall in their gates during the number of recent scans as specified by the history depth.

`[clusters,incompatibleBranches] = clusterTrackBranches(
branchHistory,'OutputForm',out)` returns the clusters in the format specified by `out`.

## Examples

### Compute Clusters of Branches

Create a branch history matrix for 12 branches. For this example, the branch history matrix has 11 columns that represent the history of 2 sensors with a history depth of 4.

```
branchHistory = uint32([
 4    9    9    0    0    1    0    0    0    0    0
 5   10   10    0    0    0    2    0    0    0    0
 6   11   11    0    0    3    0    0    0    0    0
 1   12   12    0    0    1    0    1    0    0    0
 1   13   13    0    0    0    2    1    0    0    0
 1   14   14    0    0    1    2    1    0    0    0
 2   15   15    0    0    3    0    3    0    0    0
 3   16   16    0    0    0    4    0    4    0    0
 7    0   17    1    0    0    0    0    0    0    0
 1    5   18    1    0    0    0    0    2    0    0
 1    5   19    0    2    0    0    0    2    0    0
 1    5   20    1    2    0    0    0    2    0    0]);
```

Get the list of clusters and the list of incompatible branches. The `clusters` matrix has three columns, therefore there are three clusters.

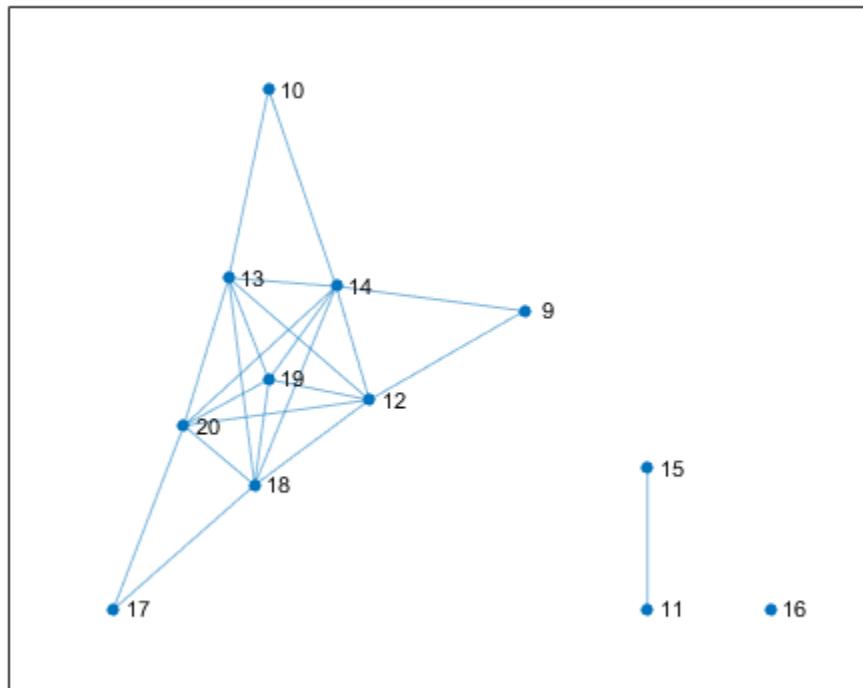
```
[clusters,incompBranches] = clusterTrackBranches(branchHistory);
size(clusters)

ans = 1x2
```

12 3

Show the incompatible branches as a graph. The numeric branch IDs are in the third column of `branchHistory`. To display the IDs of the branches on the graph, convert the IDs to character vectors. You can see the three distinct clusters.

```
branchIDs = cellstr(num2str(branchHistory(:,3)));
g = graph(incompBranches,branchIDs,'omitselfloops');
plot(g)
```



## Input Arguments

### **branchHistory — Branch history**

matrix of integers

Branch history, specified as a matrix of integers. Each row of `branchHistory` represents a unique track branch. `branchHistory` must have  $3+(D \times S)$  columns, where  $D$  is the number of maintained scans (the history depth) and  $S$  is the maximum number of maintained sensors. For more information, see the `history` output of the `trackBranchHistory` system object.

### **out — Output form**

'logical' (default) | 'vector' | 'cell'

Output form of the returned `clusters`, specified as 'logical', 'vector', or 'cell'.

## Output Arguments

### **clusters — Clusters**

*M*-by-*P* logical matrix | *M*-element numeric vector | cell array

Clusters, returned as one of the following. The format of **clusters** is specified by **out**.

- An *M*-by-*P* logical matrix. *M* is the number of branches (rows) in **branchHistory** and *P* is the number of clusters. The  $(i,j)$  element is **true** if branch *j* is contained in cluster *i*. The value of *P* is less than or equal to *M*.
- A vector of length *M*, where the *i*-th element gives the index of the cluster that contains branch *i*.
- A cell array **c**, where **c**{*j*} contains the IDs of all the branches in cluster *j*.

Data Types: logical

### **incompatibleBranches — Incompatible branches**

*M*-by-*M* symmetric logical matrix

Incompatible branches, returned as an *M*-by-*M* symmetric logical matrix. The  $(i,j)$  element is **true** if branches *i* and *j* are pairwise-incompatible.

Data Types: logical

## References

- [1] Werthmann, John R. "A Step-by-Step Description of a Computationally Efficient Version of Multiple Hypothesis Tracking." In *Proceedings of SPIE Vol. 1698, Signal and Processing of Small Targets*. 1992, pp. 288-300. doi: 10.1117/12.139379.

## Extended Capabilities

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

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

Usage notes and limitations:

- Code generation only supports the 'logical' value of output form **out**.

## See Also

[compatibleTrackBranches](#) | [pruneTrackBranches](#) | [trackBranchHistory](#) | [trackerTOMHT](#)

## Introduced in R2018b

# compatibleTrackBranches

Formulate global hypotheses from clusters

## Syntax

```
[hypotheses,hypScores] = compatibleTrackBranches(clusters,  
incompatibleBranches,scores,maxNumHypotheses)
```

## Description

[*hypotheses*,*hypScores*] = compatibleTrackBranches(*clusters*,  
*incompatibleBranches*,*scores*,*maxNumHypotheses*) returns the list of hypotheses  
*hypotheses* and their scores *hypScores* from information about clusters of branches and  
incompatibility of branches.

Hypotheses are sets of compatible track branches, which are branches that do not belong to the same track or share a detection in their history. The score of each hypothesis is the sum of scores of all branches included in the hypothesis.

## Examples

### Get Hypotheses of Branches

Create a branch history matrix for 12 branches. For this example, the branch history matrix has 11 columns that represent the history of 2 sensors with a history depth of 4.

```
branchHistory = uint32([  
    4 9 9 0 0 1 0 0 0 0 0  
    5 10 10 0 0 0 2 0 0 0 0  
    6 11 11 0 0 3 0 0 0 0 0  
    1 12 12 0 0 1 0 1 0 0 0  
    1 13 13 0 0 0 2 1 0 0 0  
    1 14 14 0 0 1 2 1 0 0 0  
    2 15 15 0 0 3 0 3 0 0 0  
    3 16 16 0 0 0 4 0 4 0 0  
    7 0 17 1 0 0 0 0 0 0 0  
    1 5 18 1 0 0 0 0 2 0 0  
    1 5 19 0 2 0 0 0 2 0 0  
    1 5 20 1 2 0 0 0 2 0 0]);
```

Get the list of clusters and the list of incompatible branches. The *clusters* matrix has three columns, therefore there are three clusters.

```
[clusters,incompBranches] = clusterTrackBranches(branchHistory);
```

Specify a 12-by-1 column vector containing the branch scores.

```
scores = [81.4; 90.5; 12.7; 91.3; 63.2; 9.7; 27.8; 54.6; 95.7; 96.4; 15.7; 97.1];
```

Specify the number of global hypotheses.

```
numHypotheses = 6;

Get a matrix of hypotheses and the score of each hypothesis.

[hyps,hypScores] = compatibleTrackBranches(clusters,incompBranches,scores,numHypotheses)

hyp = 12x6 logical array

1 0 1 1 1 0
1 1 1 1 1 1
0 0 0 0 1 1
0 1 0 0 0 1
0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 0 0
1 1 1 1 1 1
1 1 0 0 1 1
0 0 0 1 0 0

hypScores = 1x6
365.7000 359.9000 351.4000 350.7000 350.6000 344.8000
```

## Input Arguments

### **clusters — Clusters**

*M*-by-*P* logical matrix | *M*-element numeric vector | cell array

Clusters, specified as one of the following.

- An *M*-by-*P* logical matrix. *M* is the number of branches and *P* is the number of clusters. The  $(i,j)$  element is true if branch *j* is contained in cluster *i*. The value of *P* is less than or equal to *M*.
- A vector of length *M*, where the *i*-th element gives the index of the cluster that contains branch *i*.
- A cell array *c*, where *c*{*j*} contains the IDs of all the branches in cluster *j*.

You can use `clusterTrackBranches` to compute the clusters from a branch history matrix.

Data Types: logical

### **incompatibleBranches — Incompatible branches**

*M*-by-*M* symmetric logical matrix

Incompatible branches, specified as an *M*-by-*M* symmetric logical matrix. The  $(i,j)$  element is true if branches *i* and *j* are pairwise-incompatible.

You can use `clusterTrackBranches` to compute incompatible branches from a branch history matrix.

Data Types: logical

### **scores — Branch scores**

*M*-by-1 numeric vector | *M*-by-2 numeric matrix

Branch scores, specified as an  $M$ -by-1 numeric vector or an  $M$ -by-2 numeric matrix.

---

**Note** If you specify `scores` as an  $M$ -by-2 numeric matrix, then the first column specifies the current score of each branch and the second column specifies the maximum score. `compatibleTrackBranches` ignores the second column.

---

Data Types: `single` | `double`

#### **maxNumHypotheses — Maximum number of hypotheses**

positive integer

Maximum number of hypotheses, specified as a positive integer.

## Output Arguments

#### **hypotheses — Hypotheses**

$M$ -by- $H$  logical matrix

Hypotheses, returned as an  $M$ -by- $H$  logical matrix, where  $M$  is the number of branches and  $H$  is the value of `maxNumHypotheses`.

#### **hypScores — Hypotheses score**

1-by- $H$  numeric vector

Hypotheses score, returned as a 1-by- $H$  numeric vector.

## References

- [1] Werthmann, John R. "A Step-by-Step Description of a Computationally Efficient Version of Multiple Hypothesis Tracking." In *Proceedings of SPIE Vol. 1698, Signal and Processing of Small Targets*. 1992, pp. 288–300. doi: 10.1117/12.139379.

## Extended Capabilities

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

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

Usage notes and limitations:

- Code generation only supports `clusters` specified as an  $M$ -by- $P$  logical matrix.

## See Also

`clusterTrackBranches` | `pruneTrackBranches` | `trackBranchHistory` | `trackerTOMHT`

## Introduced in R2018b

# pruneTrackBranches

Prune track branches with low likelihood

## Syntax

```
[toPrune,globalProbability] = pruneTrackBranches(branchHistory,scores,  
hypotheses)  
[toPrune,globalProbability] = pruneTrackBranches(branchHistory,scores,  
hypotheses,Name,Value)  
[toPrune,globalProbability,info] = pruneTrackBranches(____)
```

## Description

[toPrune,globalProbability] = pruneTrackBranches(branchHistory,scores,hypotheses) returns a logical flag, toPrune, that indicates which branches should be pruned based on the branch history, branch scores, and hypotheses. pruneTrackBranches also returns the global branch probabilities, globalProbability.

[toPrune,globalProbability] = pruneTrackBranches(branchHistory,scores,hypotheses,Name,Value) uses name-value pairs to modify how branches are pruned.

[toPrune,globalProbability,info] = pruneTrackBranches(\_\_\_\_) returns additional information, info, about the pruned branches.

## Examples

### Prune Branches For Single Sensor Using N-Scan Pruning

Create a branch history matrix for a single sensor with 20 branches. For this example, the history depth is 4 therefore the matrix has 7 columns.

```
history = [  
    8    14    14    0    0    2    0  
    1    23    23    0    0    2    1  
    2    24    24    0    0    1    2  
    9    25    25    0    1    0    0  
   10    26    26    0    2    0    0  
    1    28    28    0    1    0    1  
    4    33    33    0    1    2    1  
    1    34    34    0    1    2    1  
    2    35    35    0    2    1    2  
   11    0    36    1    0    0    0  
   12    0    37    2    0    0    0  
    8    14    38    2    0    2    0  
    1    23    39    2    0    2    1  
    2    24    40    1    0    1    2  
    9    25    41    2    1    0    0  
   10    26    42    1    2    0    0  
    1    28    43    2    1    0    1  
    4    33    44    2    1    2    1
```

```
1    34    45    2    1    2    1
2    35    46    1    2    1    2];
```

Get the list of clusters and the list of incompatible branches. The `clusters` matrix has two columns, therefore there are two clusters.

```
[clusters,incompBranches] = clusterTrackBranches(history);
```

Specify a 20-by-1 column vector containing branch scores.

```
scores = [4.5 44.9 47.4 6.8 6.8 43.5 50.5 61.9 64.7 9.1 9.1 19 61.7 ...
63.5 21.2 20.5 60.7 67.3 79.2 81.5]';
```

Get a matrix of hypothesis.

```
hypotheses = compatibleTrackBranches(clusters,incompBranches,scores,10);
```

Prune the track branches, using name-value pair arguments to specify a single sensor and the 'Hypothesis' method of N-scan pruning. Return the pruning flag, global probability, and pruning information about each branch. To make the information easier to compare, convert the information from a struct to a table.

The *i*-th value of `toPrune` is `true` if any of 'PrunedByProbability', 'PrunedByNScan', or 'PrunedByNumBranches' are `true` in the *i*-th row of the information table.

```
[toPrune,probs,info] = pruneTrackBranches(history,scores,hypotheses, ...
'NumSensors',1,'NScanPruning','Hypothesis');
infoTable = struct2table(info)
```

BranchID	PriorProbability	GlobalProbability	PrunedByProbability	PrunedByNScan
14	0.98901	0.098901	false	false
23	1	0.1	false	false
24	1	0.1	false	false
25	0.99889	0.099889	false	false
26	0.99889	0.099889	false	false
28	1	0	true	true
33	1	0	true	false
34	1	0.2	false	false
35	1	0.2	false	false
36	0.99989	0.19998	false	false
37	0.99989	0.19998	false	false
38	1	0	true	false
39	1	0.1	false	false
40	1	0.1	false	false
41	1	0.1	false	false
42	1	0.1	false	false
:				

## Input Arguments

### branchHistory — Branch history

matrix of integers

Branch history, specified as a matrix of integers. Each row of `branchHistory` represents a unique track branch. `branchHistory` must have  $3+(D \times S)$  columns, where  $D$  is the number of maintained scans (the history depth) and  $S$  is the maximum number of maintained sensors. For more information, see the `history` output of the `trackBranchHistory` system object.

**scores — Branch scores**

$M$ -by-1 numeric vector |  $M$ -by-2 numeric matrix

Branch scores, specified as an  $M$ -by-1 numeric vector or an  $M$ -by-2 numeric matrix.

---

**Note** If you specify `scores` as an  $M$ -by-2 numeric matrix, then the first column specifies the current score of each branch and the second column specifies the maximum score. `pruneTrackBranches` ignores the second column.

---

Data Types: `single` | `double`

**hypotheses — Hypotheses**

$M$ -by- $H$  logical matrix

Hypotheses, returned as an  $M$ -by- $H$  logical matrix, where  $M$  is the number of branches and  $H$  is the number of global hypotheses. You can use `clusterTrackBranches` to compute the clusters from a branch history matrix, then use `compatibleTrackBranches` to compute the hypotheses from the clusters.

**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: `[toPrune, probs] = pruneTrackBranches(branchHistory, scores, hypotheses, 'MinBranchProbability', 2e-3);`

**MinBranchProbability — Minimum branch probability**

$1e-3$  (default) | number in the range [0,1)

Minimum branch probability threshold, specified as the comma-separated pair consisting of '`MinBranchProbability`' and a number in the range [0,1). Typical values are between  $1e-3$  and  $5e-3$ . The `pruneTrackBranches` function prunes branches with global probability less than the threshold.

**MaxNumTrackBranches — Maximum number of branches**

3 (default) | positive integer

Maximum number of branches to keep per track, specified as the comma-separated pair consisting of '`MaxNumTrackBranches`' and a positive integer. Typical values are between 2 and 6. If a track has more than this number of branches, then `pruneTrackBranches` prunes branches with the lowest initial score.

**NScanPruning — N-scan pruning method**

'None' (default) | 'Hypothesis'

N-scan pruning method, specified as the comma-separated pair consisting of 'NScanPruning' and 'None' or 'Hypothesis'. If you specify 'Hypothesis', then `pruneTrackBranches` prunes branches that are incompatible with the current most likely branch in the most recent  $N$  scans. By default, `pruneTrackBranches` does not use N-scan pruning.

#### **NumSensors — Number of sensors**

20 (default) | positive integer

Number of sensors in history, specified as the comma-separated pair consisting of 'NumSensors' and a positive integer.

## Output Arguments

#### **toPrune — Branches to prune**

M-by-1 logical vector

Branches to prune, returned as an  $M$ -by-1 logical vector. A value of `true` indicates that the branch should be pruned.

Data Types: logical

#### **globalProbability — Global branch probabilities**

M-by-1 numeric vector

Global branch probabilities, returned as an  $M$ -by-1 numeric vector.

#### **info — Pruning information**

struct

Pruning information about each branch, returned as a struct with the following fields.

- **BranchID** — An  $M$ -by-1 numeric vector. Each value specifies the ID of a track branch. The IDs come from the third column of `branchHistory`.
- **PriorProbability** — An  $M$ -by-1 numeric vector. Each value specifies the branch prior probability from the branch score.
- **GlobalProbability** — An  $M$ -by-1 numeric vector. Each value specifies the branch global probability, which considers the hypotheses that contain the branch and their scores.
- **PrunedByProbability** — An  $M$ -by-1 logical vector. A value of `true` indicates that the branch is pruned by `MinBranchProbability`.
- **PrunedByNScan** — An  $M$ -by-1 logical vector. A value of `true` indicates that the branch is pruned by `NScanPruning`.
- **PrunedByNumBranches** — An  $M$ -by-1 logical vector. A value of `true` indicates that the branch is pruned by `MaxNumTrackBranches`.

## References

- [1] Werthmann, John R. "A Step-by-Step Description of a Computationally Efficient Version of Multiple Hypothesis Tracking." In *Proceedings of SPIE Vol. 1698, Signal and Processing of Small Targets*. 1992, pp. 288–300. doi: 10.1117/12.139379.
- [2] Blackman, Samuel, and Robert Popoli. "Design and Analysis of Modern Tracking Systems." Artech House, 1999.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[clusterTrackBranches](#) | [compatibleTrackBranches](#) | [trackBranchHistory](#) | [trackerTOMHT](#)

### Introduced in R2018b

# triangulateLOS

Triangulate multiple line-of-sight detections

## Syntax

```
estPos = triangulateLOS(detections)
[estPos,estCov] = triangulateLOS(detections)
```

## Description

`estPos = triangulateLOS(detections)` estimates the position of a target in a global Cartesian coordinate frame by triangulating a set of angle-only detections. Angle-only detections are also known as line-of-sight (LOS) detections. For more details, see “Algorithms” on page 1-470.

`[estPos,estCov] = triangulateLOS(detections)` also returns `estCov`, the covariance of the error in target position. The function uses a Taylor-series approximation to estimate the error covariance.

## Examples

### Triangulate Line-of-Sight Measurements from Three Sensors

Load a MAT-file containing a set of line-of-sight detections stored in the variable `detectionSet`.

```
load angleOnlyDetectionFusion.mat
```

Plot the angle-only detections and the sensor positions. Specify a range of 5 km for plotting the direction vector. To specify the position of the origin, use the second measurement parameter because the sensor is located at the center of the platform. Convert the azimuth and elevation readings to Cartesian coordinates.

```
rPlot = 5000;

for i = 1:numel(detectionSet)
    originPos = detectionSet{i}.MeasurementParameters(2).OriginPosition;

    az = detectionSet{i}.Measurement(1);
    el = detectionSet{i}.Measurement(2);
    [xt,yt,zt] = sph2cart(deg2rad(az),deg2rad(el),rPlot);

    positionData(:,i) = originPos;
    plotData(:,3*i+(-2:0)) = [xt yt zt]'*[1 0 NaN]+originPos;
end

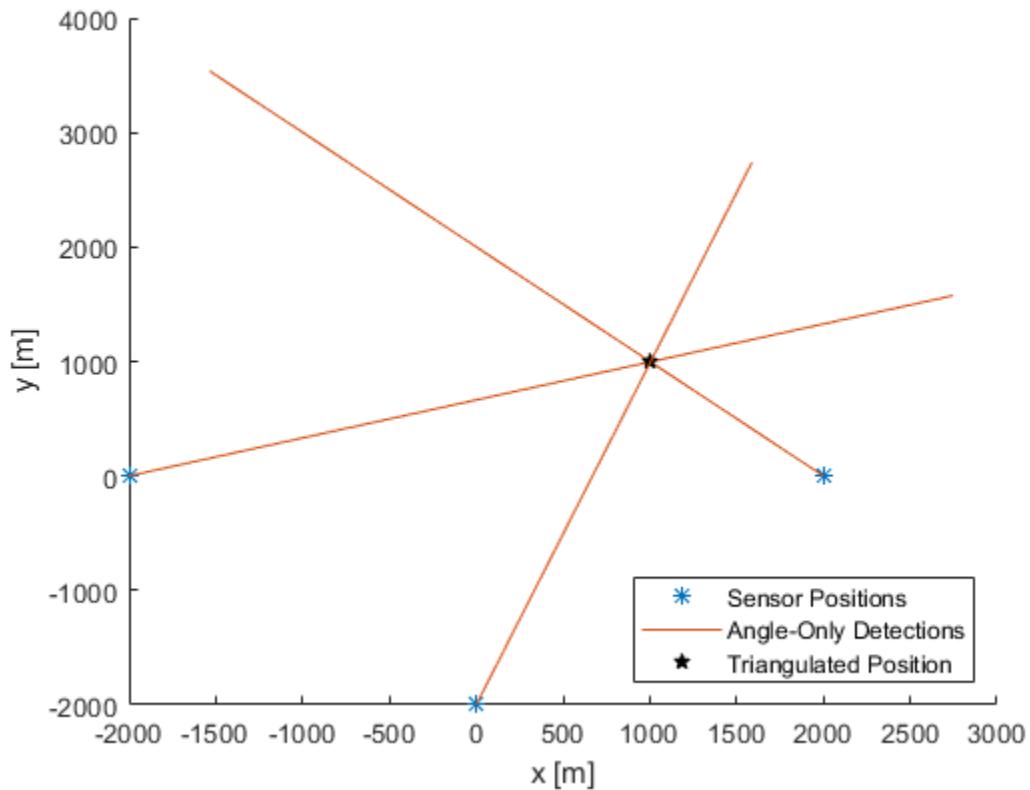
plot3(positionData(1,:),positionData(2,:),positionData(3,:),'*')
hold on
plot3(plotData(1,:),plotData(2,:),plotData(3,:))
```

Triangulate the detections by using `triangulateLOS`. Plot the triangulated position.

```
[estPos,estCov] = triangulateLOS(detectionSet);

plot3(estPos(1),estPos(2),estPos(3), 'pk', 'MarkerFaceColor', 'k')
hold off

legend('Sensor Positions', 'Angle-Only Detections', 'Triangulated Position', ...
    'location', 'southeast')
xlabel('x [m]')
ylabel('y [m]')
view(2)
```



## Input Arguments

### **detections — Line-of-sight measurements**

cell array of `objectDetection` objects

Line-of-sight measurements, specified as a cell array of `objectDetection` objects. Each object has the properties listed in the table.

Property	Definition
Time	Measurement time
Measurement	Object measurements
MeasurementNoise	Measurement noise covariance matrix

Property	Definition
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

Each detection must specify the **MeasurementParameters** property as a structure with the fields described in the table.

Parameter	Definition
Frame	Frame used to report measurements. Specify <b>Frame</b> as ' <b>spherical</b> ' for the first structure.
OriginPosition	Position offset of the origin of the frame relative to the parent frame, represented as a 3-by-1 real vector.
OriginVelocity	Velocity offset of the origin of the frame relative to the parent frame, represented as a 3-by-1 real vector.
Orientation	A 3-by-3 real-valued orthonormal frame orientation matrix.
IsParentToChild	A logical scalar that indicates if <b>Orientation</b> is given as a frame rotation from the parent coordinate frame to the child coordinate frame. If <b>false</b> , then <b>Orientation</b> is given as a frame rotation from the child coordinate frame to the parent coordinate frame.
HasElevation	A logical scalar that indicates if elevation is included in the measurements. This parameter is <b>true</b> by default.
HasAzimuth	A logical scalar that indicates if azimuth is included in the measurements. This parameter is <b>true</b> by default. If specified as a field, it must be set to <b>true</b> .
HasRange	A logical scalar that indicates if range is included in the measurements. This parameter must be specified as a field and set to <b>false</b> .
HasVelocity	A logical scalar that indicates if velocity is included in the measurements. This parameter is <b>false</b> by default. If specified as a field, it must be set to <b>false</b> .

The function provides default values for fields left unspecified.

## Output Arguments

### **estPos — Estimated position**

3-by-1 vector

Estimated position of the target, returned as a 3-by-1 vector.

### **estCov — Estimated error covariance**

3-by-3 matrix

Estimated error covariance of the target position, returned as a 3-by-3 matrix.

## Algorithms

Multiple angle-only or line-of-sight measurements result in lines in space. These lines might or might not intersect because of measurement noise. `triangulateLOS` uses a suboptimal linear least-squares method to minimize the distance of miss between multiple detections. The formulation makes these assumptions:

- All detections report measurements with approximately the same accuracy in azimuth and elevation (if measured).
- The distances from the different sensors to the triangulated target are all of the same order.

## References

- [1] Blackman, Samuel, and Robert Popoli. "Design and analysis of modern tracking systems." Norwood, MA: Artech House, 1999. (1999).

## Extended Capabilities

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

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

## See Also

### **Objects**

`objectDetection`

### **System Objects**

`staticDetectionFuser`

**Introduced in R2018b**

# 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: 3000000000
        Bandwidth: 3000000
        WaveformType: 0
        ProcessingGain: 0
        PropagationRange: 0
        PropagationRangeRate: 0
        EIRP: 0
        RCS: 0
```

## Reflect Radar Emission From Platform within Tracking Scenario

Reflect a radar emission from a platform defined within a trackingScenario.

Create a tracking scenario object.

```
scenario = trackingScenario;
```

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, Platform, 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 quaternion(1,0,0,0).
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 array {rcsSignature,irSignature , tsSignature}

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` | `radarSensor`

**Introduced in R2018b**

# underwaterChannel

Propagated and reflected sonar signals

## Syntax

```
sonarsigout = underwaterChannel(sonarsigin,platforms)
```

## Description

`sonarsigout = underwaterChannel(sonarsigin,platforms)` returns sonar signals, `sonarsigout`, as combinations of signals, `sonarsigin`, reflected from platforms, `platforms`.

## Examples

### Reflect Sonar Emission From Platform

Create a sonar emission and a platform and reflect the emission from the platform.

Create a sonar emission object.

```
sonarSig = sonarEmission('PlatformID',1,'EmitterIndex',1,'OriginPosition',[0 0 0]);
```

Create a platform structure.

```
platfm = struct('PlatformID',2,'Position',[10 0 0],'Signatures',tsSignature());
```

Reflect the emission from the platform.

```
sigs = underwaterChannel(sonarSig,platfm)
```

```
sigs =  
2x1 sonarEmission array with properties:
```

```
SourceLevel  
TargetStrength  
PlatformID  
EmitterIndex  
OriginPosition  
OriginVelocity  
Orientation  
FieldOfView  
CenterFrequency  
Bandwidth  
WaveformType  
ProcessingGain  
PropagationRange  
PropagationRangeRate
```

## Reflect Sonar Emission from Platform within Tracking Scenario

Reflect a sonar emission from a platform defined within a trackingScenario.

Create a tracking scenario object.

```
scenario = trackingScenario;
```

Create an sonarEmitter.

```
emitter = sonarEmitter(1);
```

Mount the emitter on a platform within the scenario.

```
plat = platform(scenario,'Emitters',emitter);
```

Add another platform to reflect the emitted signal.

```
tgt = platform(scenario);
tgt.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 sonarEmission}
```

Reflect the signal from the platforms in the scenario.

```
sigs = underwaterChannel(txSigs, scenario.Platforms)

sigs = 1x1 cell array
    {1x1 sonarEmission}
```

## Input Arguments

### **sonarsigin — Input sonar signals**

array of sonarEmission objects

Input sonar signals, specified as an array of sonarEmission objects.

### **platforms — Reflector platform**

cell array of Platform objects | array of Platform structures

Reflector platforms, specified as a cell array of Platform objects, Platform, 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.

Field	Description
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 quaternion(1,0,0,0).
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 array {rcsSignature,irSignature , tsSignature}

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.

## Output Arguments

### **sonarsigout — Reflected sonar signals**

array of `sonarEmission` objects

Reflected sonar signals, specified as an array of `sonarEmission` objects.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`sonarEmission` | `sonarEmitter` | `sonarSensor`

**Introduced in R2018b**

# 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:

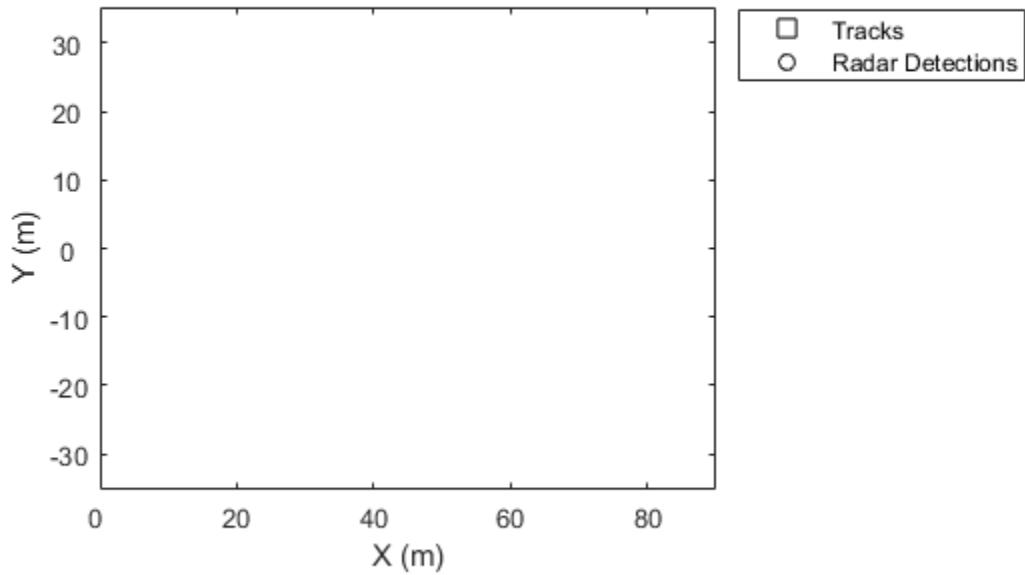
- `detectionPlotter`
- `orientationPlotter`
- `platformPlotter`
- `trackPlotter`
- `trajectoryPlotter`

## Examples

### Clear Specific Plotter Data

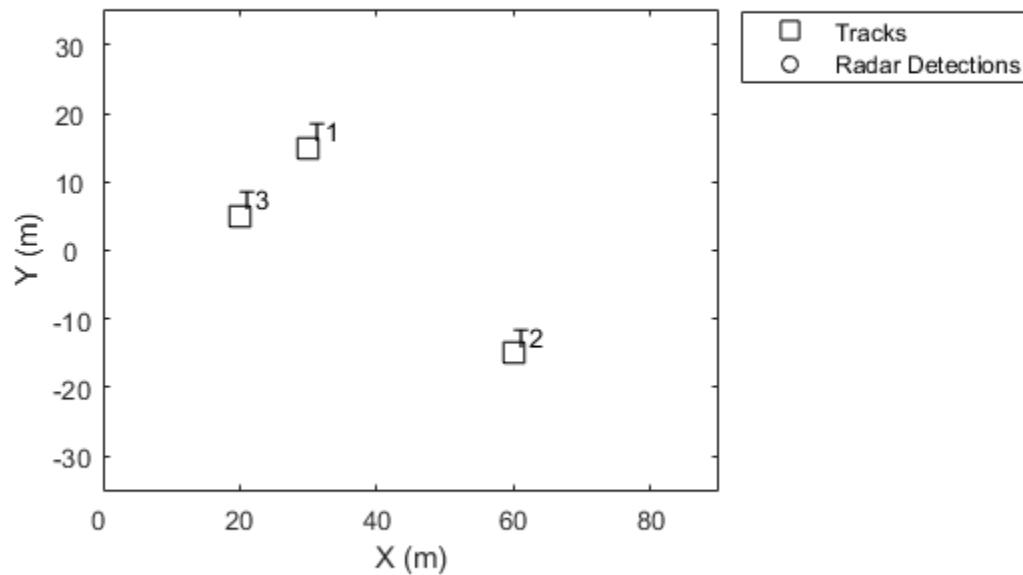
Create a theater plot. Add a track plotter and detection plotter to the theater plot.

```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35]);
tPlotter = trackPlotter(tp,'DisplayName','Tracks');
radarPlotter = detectionPlotter(tp,'DisplayName','Radar Detections');
```



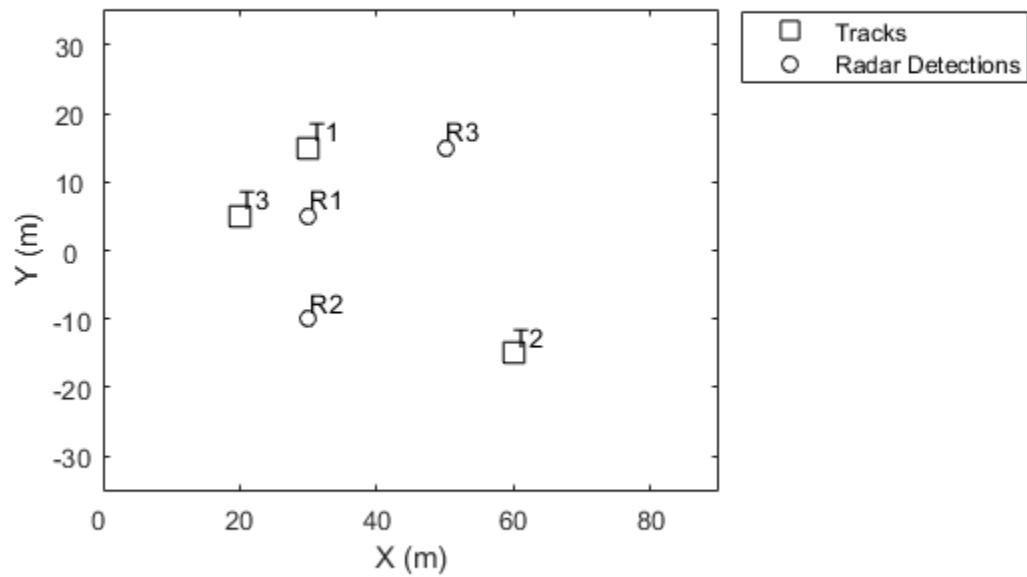
Plot a set of tracks in the track plotter.

```
trackPos = [30, 15, 1; 60, -15, 1; 20, 5, 1];
trackLabels = {'T1', 'T2', 'T3'};
plotTrack(tPlotter, trackPos, trackLabels)
```



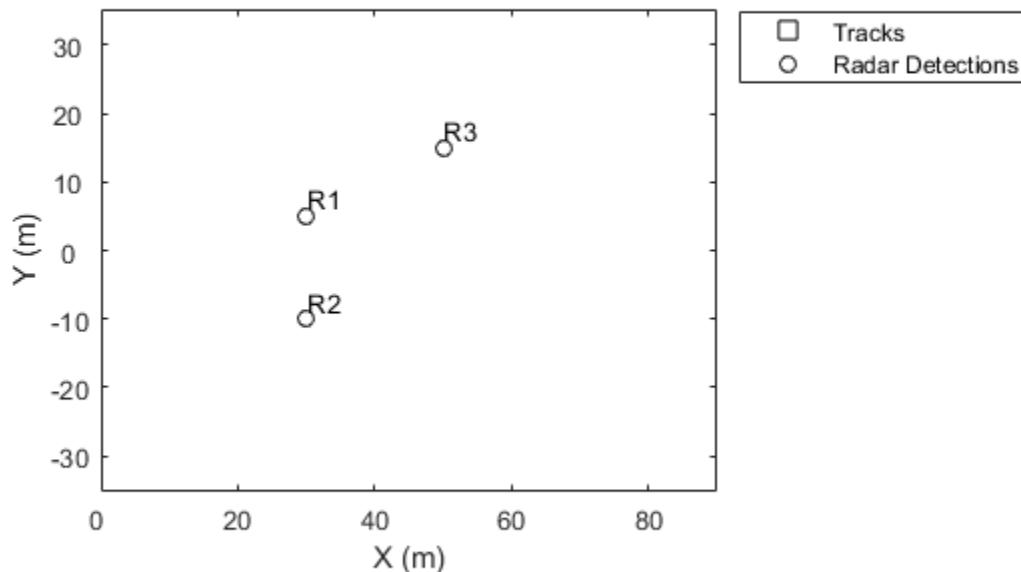
Plot a set of detections in the detection plotter.

```
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

**pl — 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` | `findPlotter` | `theaterPlot`

**Introduced in R2018b**

# emissionsInBody

Transform emissions to body frame of platform

## Syntax

```
embody = emissionsInBody(emscene,bodyframe)
```

## Description

`embody = emissionsInBody(emscene,bodyframe)` converts emissions, `emscene`, referenced to scenario coordinates into emissions, `embody`, referenced to platform body coordinates. `bodyframe` specifies the position, velocity, and orientation of the platform body.

## Examples

### Convert Radar Emission to Body Frame

Convert a radar emission from scenario coordinates to body frame.

Define a radar emission with respect to the scenario frame.

```
emScene = radarEmission('PlatformID',1,'EmitterIndex',1, ...
    'OriginPosition',[0 0 0])
```

```
emScene =
    radarEmission with properties:
```

```
    PlatformID: 1
    EmitterIndex: 1
    OriginPosition: [0 0 0]
    OriginVelocity: [0 0 0]
    Orientation: [1x1 quaternion]
    FieldOfView: [180 180]
    CenterFrequency: 3000000000
    Bandwidth: 3000000
    WaveformType: 0
    ProcessingGain: 0
    PropagationRange: 0
    PropagationRangeRate: 0
    EIRP: 0
    RCS: 0
```

Define the position, velocity, and orientation, of the body relative to the scenario frame.

```
bodyFrame = struct( ...
    'Position',[10 0 0], ...
    'Velocity',[5 5 0], ...
    'Orientation',quaternion([45 0 0],'eulerd','zyx','frame'));
```

Convert the emission into the body frame.

```

emBody = emissionsInBody(emScene,bodyFrame)

emBody =
    radarEmission with properties:

        PlatformID: 1
        EmitterIndex: 1
        OriginPosition: [-7.0711 7.0711 0]
        OriginVelocity: [-7.0711 4.4409e-16 0]
        Orientation: [1x1 quaternion]
        FieldOfView: [180 180]
        CenterFrequency: 300000000
        Bandwidth: 3000000
        WaveformType: 0
        ProcessingGain: 0
        PropagationRange: 0
        PropagationRangeRate: 0
        EIRP: 0
        RCS: 0

```

## Convert Sonar Emission into Body Frame

Convert a sonar emission from scenario coordinates into body coordinates. Use `trackingScenario` to define the motion of the body and use `sonarEmitter` to create the emission.

Set up a tracking scenario.

```
scene = trackingScenario;
```

Create a sonar emitter to mount on a platform.

```
emitter = sonarEmitter(1,'No scanning');
```

Mount the emitter on a platform in the scenario 100 meters below sea-level.

```
platTx = platform(scene,'Emitters',emitter);
platTx.Trajectory.Position = [10 0 100];
```

Create another platform in the scenario.

```
platRx = platform(scene);
platRx.Trajectory.Position = [100 0 100];
platRx.Trajectory.Orientation = quaternion([45 0 0],'eulerd',...
    'zyx','frame');
```

Emit a signal. The emitted signal is in the scenario frame.

```
emScene = emit(platTx,scene.SimulationTime)

emScene = 1x1 cell array
    {1x1 sonarEmission}
```

Propagate the emission through an underwater channel.

```
emPropScene = underwaterChannel(emScene,scene.Platforms)
```

```
emPropScene=2x1 cell array
{1x1 sonarEmission}
{1x1 sonarEmission}
```

Convert the emission to the body frame of the second platform.

```
emBodyRx = emissionsInBody(emPropScene, platRx);
disp(emBodyRx(1))

{1x1 sonarEmission}
```

## Input Arguments

### **emscene — Emissions in scenario coordinates**

emission object

Emissions in scenario coordinates, specified as a cell array of `radarEmission` or `sonarEmission` emission objects.

### **bodyframe — Body frame**

structure | Platform object

Body frame, specified as a structure or `Platform` object. You can use a `Platform` object because it contains the necessary information. The body frame structure must contain at least these fields:

Field	Description
Position	Position of body in scenario coordinates, specified as a real-valued 1-by-3 vector. This field is required. There is no default value. Units are in meters.
Velocity	Velocity of body in scenario coordinates, specified as a real-valued 1-by-3 vector. Units are in meters per second. The default is [0 0 0].
Orientation	Orientation of body with respect to the scenario coordinate frame, specified as a scalar quaternion or a 3-by-3 rotation matrix. Orientation defines the frame rotation from the scenario coordinate system to the body coordinate system. Units are dimensionless. The default is <code>quaternion(1,0,0,0)</code> or, equivalently, <code>eye(3)</code> .

Because the fields in the body frame structure are a subset of the fields in a platform structure, you can use the platform structure output from the `platformPoses` method of `trackingScenario` as the input `bodyframe`.

## Output Arguments

### **embody — Emissions in body coordinates**

emission object

Emissions in body coordinates, returned as a cell array of `radarEmission` and `sonarEmission` emission objects.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

`radarChannel` | `underwaterChannel`

### Objects

`Platform` | `radarEmission` | `sonarEmission` | `trackingScenario`

### System Objects

`radarEmitter` | `sonarEmitter`

## Introduced in R2018b



# Classes

---

# ahrs10filter

Height and orientation from MARG and altimeter readings

## Description

The `ahrs10filter` object fuses MARG and altimeter sensor data to estimate device height and orientation. MARG (magnetic, angular rate, gravity) data is typically derived from magnetometer, gyroscope, and accelerometer sensors. The filter uses an 18-element state vector to track the orientation quaternion, vertical velocity, vertical position, MARG sensor biases, and geomagnetic vector. The `ahrs10filter` object uses an extended Kalman filter to estimate these quantities.

## Creation

### Syntax

```
FUSE = ahrs10filter  
FUSE = ahrs10filter('ReferenceFrame',RF)  
FUSE = ahrs10filter(___,Name,Value)
```

### Description

`FUSE = ahrs10filter` returns an extended Kalman filter object, `FUSE`, for sensor fusion of MARG and altimeter readings to estimate device height and orientation.

`FUSE = ahrs10filter('ReferenceFrame',RF)` returns an extended Kalman filter object that estimates device height and orientation relative to the reference frame `RF`. Specify `RF` as '`NED`' (North-East-Down) or '`ENU`' (East-North-Up). The default value is '`NED`'.

`FUSE = ahrs10filter(___,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

## Properties

### **IMUSampleRate — Sample rate of the IMU (Hz)**

`100` (default) | positive scalar

Sample rate of the IMU in Hz, specified as a positive scalar.

Data Types: `single` | `double`

### **GyroscopeNoise — Multiplicative process noise variance from gyroscope ((rad/s)<sup>2</sup>)**

`[1e-9,1e-9,1e-9]` (default) | scalar | three-element row vector

Multiplicative process noise variance from the gyroscope in  $(\text{rad}/\text{s})^2$ , specified as positive real finite numbers.

Data Types: `single` | `double`

**AccelerometerNoise — Multiplicative process noise variance from accelerometer ( $(m/s^2)^2$ )**  
 $[1e-4, 1e-4, 1e-4]$  (default) | scalar | three-element row vector

Multiplicative process noise variance from the accelerometer in  $(m/s^2)^2$ , specified as positive real finite numbers.

Data Types: single | double

**GyroscopeBiasNoise — Multiplicative process noise variance from gyroscope bias ( $(rad/s^2)^2$ )**  
 $[1e-10, 1e-10, 1e-10]$  (default) | scalar | three-element row vector

Multiplicative process noise variance from the gyroscope bias in  $(rad/s^2)^2$ , specified as positive real finite numbers.

Data Types: single | double

**AccelerometerBiasNoise — Multiplicative process noise variance from accelerometer bias ( $(m/s^2)^2$ )**  
 $[1e-4, 1e-4, 1e-4]$  (default) | scalar | three-element row vector

Multiplicative process noise variance from the accelerometer bias in  $(m/s^2)^2$ , specified as positive real finite numbers.

Data Types: single | double

**GeomagneticVectorNoise — Additive process noise for geomagnetic vector ( $\mu T^2$ )**  
 $[1e-6, 1e-6, 1e-6]$  (default) | scalar | three-element row vector

Additive process noise for geomagnetic vector in  $\mu T^2$ , specified as positive real finite numbers.

Data Types: single | double

**MagnetometerBiasNoise — Additive process noise for magnetometer bias ( $\mu T^2$ )**  
 $[0.1, 0.1, 0.1]$  (default) | scalar | three-element row vector

Additive process noise for magnetometer bias in  $\mu T^2$ , specified as positive real finite numbers.

Data Types: single | double

## **State — State vector of extended Kalman filter**

18-element column vector

State vector of the extended Kalman filter. The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Altitude (NED or ENU)	m	5
Vertical Velocity (NED or ENU)	m/s	6
Delta Angle Bias (XYZ)	rad/s	7:9
Delta Velocity Bias (XYZ)	m/s	10:12
Geomagnetic Field Vector (NED or ENU)	$\mu T$	13:15
Magnetometer Bias (XYZ)	$\mu T$	16:18

The default initial state corresponds to an object at rest located at [0 0 0] in geodetic LLA coordinates.

Data Types: single | double

#### **StateCovariance — State error covariance for extended Kalman filter**

`eye(18)*1e-6` (default) | 18-by-18 matrix

State error covariance for the Kalman filter, specified as an 18-by-18-element matrix of real numbers.

Data Types: single | double

## **Object Functions**

<code>predict</code>	Update states using accelerometer and gyroscope data for <code>ahrs10filter</code>
<code>fusemag</code>	Correct states using magnetometer data for <code>ahrs10filter</code>
<code>fusealtimeter</code>	Correct states using altimeter data for <code>ahrs10filter</code>
<code>correct</code>	Correct states using direct state measurements for <code>ahrs10filter</code>
<code>residual</code>	Residuals and residual covariances from direct state measurements for <code>ahrs10filter</code>
<code>residualmag</code>	Residuals and residual covariance from magnetometer measurements for <code>ahrs10filter</code>
<code>residualaltimeter</code>	Residuals and residual covariance from altimeter measurements for <code>ahrs10filter</code>
<code>pose</code>	Current orientation and position estimate for <code>ahrs10filter</code>
<code>reset</code>	Reset internal states for <code>ahrs10filter</code>
<code>stateinfo</code>	Display state vector information for <code>ahrs10filter</code>
<code>copy</code>	Create copy of <code>ahrs10filter</code>

## **Examples**

### **Estimate Pose of UAV**

Load logged sensor data, ground truth pose, and initial state and initial state covariance. Calculate the number of IMU samples per altimeter sample and the number of IMU samples per magnetometer sample.

```
load('fusel0exampledata.mat', ...
    'imuFs', 'accelData', 'gyroData', ...
    'magnetometerFs', 'magData', ...
    'altimeterFs', 'altData', ...
    'expectedHeight', 'expectedOrient', ...
    'initstate', 'initcov');

imuSamplesPerAlt = fix(imuFs/altimeterFs);
imuSamplesPerMag = fix(imuFs/magnetometerFs);
```

Create an AHRS filter that fuses MARG and altimeter readings to estimate height and orientation. Set the sampling rate and measurement noises of the sensors. The values were determined from datasheets and experimentation.

```
filt = ahrs10filter('IMUSampleRate', imuFs, ...
    'AccelerometerNoise', 0.1, ...
    'State', initstate, ...
    'StateCovariance', initcov);
```

```
Ralt = 0.24;
Rmag = 0.9;
```

Preallocate variables to log height and orientation.

```
numIMUSamples = size(accelData,1);
estHeight = zeros(numIMUSamples,1);
est0orient = zeros(numIMUSamples,1,'quaternion');
```

Fuse accelerometer, gyroscope, magnetometer and altimeter data. The outer loop predicts the filter forward at the fastest sample rate (the IMU sample rate).

```
for ii = 1:numIMUSamples

    % Use predict to estimate the filter state based on the accelerometer and
    % gyroscope data.
    predict(filt,accelData(ii,:),gyroData(ii,:));

    % Magnetometer data is collected at a lower rate than IMU data. Fuse
    % magnetometer data at the lower rate.
    if ~mod(ii,imuSamplesPerMag)
        fusemag(filt,magData(ii,:),Rmag);
    end

    % Altimeter data is collected at a lower rate than IMU data. Fuse
    % altimeter data at the lower rate.
    if ~mod(ii, imuSamplesPerAlt)
        fusealtimeter(filt,altData(ii),Ralt);
    end

    % Log the current height and orientation estimate.
    [estHeight(ii),est0orient(ii)] = pose(filt);
end
```

Calculate the RMS errors between the known true height and orientation and the output from the AHRS filter.

```
pErr = expectedHeight - estHeight;
qErr = rad2deg(dist(expected0orient,est0orient));

pRMS = sqrt(mean(pErr.^2));
qRMS = sqrt(mean(qErr.^2));

fprintf('Altitude RMS Error\n');

Altitude RMS Error

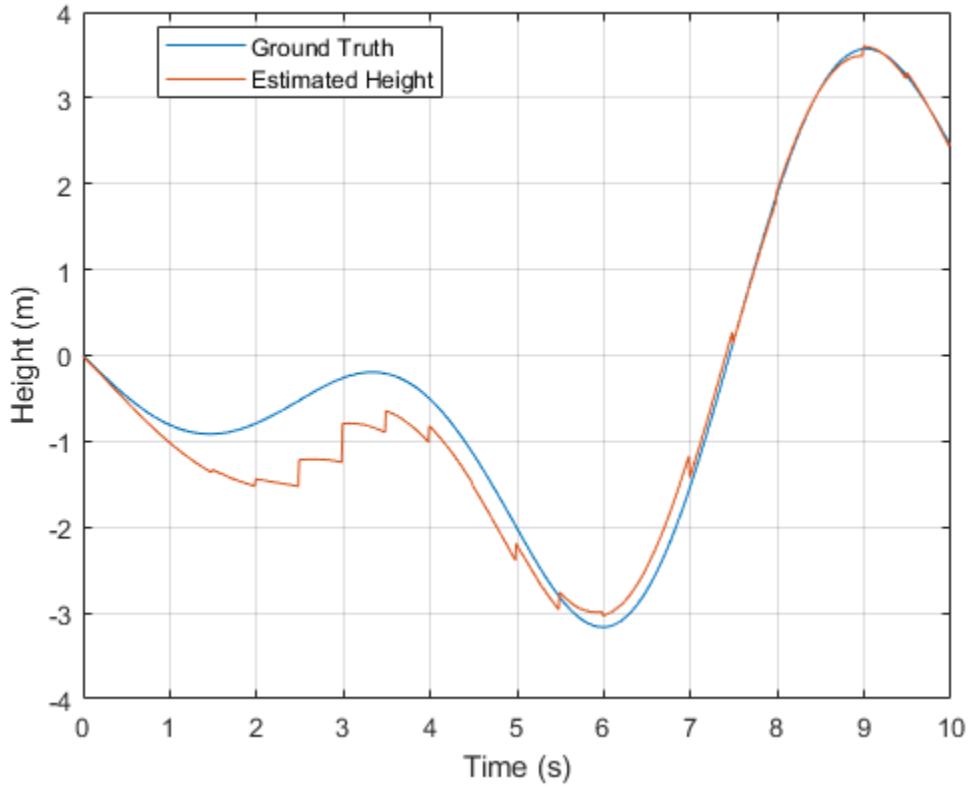
fprintf('\t%.2f (meters)\n\n',pRMS);

0.38 (meters)
```

Visualize the true and estimated height over time.

```
t = (0:(numIMUSamples-1))/imuFs;
plot(t,expectedHeight);hold on
plot(t,estHeight);hold off
legend('Ground Truth','Estimated Height','location','best')
ylabel('Height (m)')
```

```
xlabel('Time (s)')  
grid on
```



```
fprintf('Quaternion Distance RMS Error\n');  
Quaternion Distance RMS Error  
fprintf('\t%.2f (degrees)\n\n',qRMS);  
2.93 (degrees)
```

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[ahrsfilter](#) | [insfilter](#)

**Introduced in R2019a**

# copy

Create copy of ahrs10filter

## Syntax

```
newFilter = copy(filter)
```

## Description

`newFilter = copy(filter)` returns a copy of the `ahrs10filter`, `filter`, with the exactly same property values.

## Input Arguments

### **filter — Filter to be copied**

`ahrs10filter`

Filter to be copied, specified as an `ahrs10filter` object.

## Output Arguments

### **newFilter — New copied filter**

`ahrs10filter`

New copied filter, returned as an `ahrs10filter` object.

## See Also

`ahrs10filter`

## Introduced in R2020b

## reset

Reset internal states for `ahrs10filter`

### Syntax

```
reset(FUSE)
```

### Description

`reset(FUSE)` resets the `State`, `StateCovariance`, and internal integrators to their default values.

### Input Arguments

**FUSE — `ahrs10filter` object**  
object

Object of `ahrs10filter`.

### Extended Capabilities

**C/C++ Code Generation**  
Generate C and C++ code using MATLAB® Coder™.

### See Also

`ahrs10filter` | `insfilter`

**Introduced in R2019a**

# predict

Update states using accelerometer and gyroscope data for `ahrs10filter`

## Syntax

```
predict(FUSE,accelReadings,gyroReadings)
```

## Description

`predict(FUSE,accelReadings,gyroReadings)` fuses accelerometer and gyroscope data to update the state estimate.

## Input Arguments

**FUSE — ahrs10Filter object**  
object

Object of `ahrs10filter`.

**accelReadings — Accelerometer readings in the sensor body coordinate system (m/s<sup>2</sup>)**  
*N*-by-3 matrix

Accelerometer readings in local sensor body coordinate system in m/s<sup>2</sup>, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `accelReadings` represent the [x y z] measurements. Accelerometer readings are assumed to correspond to the sample rate specified by the `IMUSampleRate` property.

Data Types: `single` | `double`

**gyroReadings — Gyroscope readings in the sensor body coordinate system (rad/s)**  
*N*-by-3 matrix

Gyroscope readings in the sensor body coordinate system in rad/s, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `gyroReadings` represent the [x y z] measurements. Gyroscope readings are assumed to correspond to the sample rate specified by the `IMUSampleRate` property.

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

`ahrs10filter` | `insfilter`

## Introduced in R2019a

## pose

Current orientation and position estimate for `ahrs10filter`

### Syntax

```
[position, orientation, velocity] = pose(FUSE)
[position, orientation, velocity] = pose(FUSE,format)
```

### Description

`[position, orientation, velocity] = pose(FUSE)` returns the current estimate of the pose.

`[position, orientation, velocity] = pose(FUSE,format)` returns the current estimate of the pose with orientation in the specified orientation format.

### Input Arguments

**FUSE — ahrs10filter object**  
object

Object of `ahrs10filter`.

**format — Output orientation format**  
`'quaternion'` (default) | `'rotmat'`

Output orientation format, specified as either `'quaternion'` for a quaternion or `'rotmat'` for a rotation matrix.

Data Types: `char` | `string`

### Output Arguments

**position — Vertical position estimate in the local NED coordinate system (m)**  
scalar

Vertical position estimate in the local NED coordinate system in meters, returned as a scalar.

Data Types: `single` | `double`

**orientation — Orientation estimate in the local NED coordinate system**  
`quaternion` (default) | 3-by-3 rotation matrix

Orientation estimate in the local NED coordinate system, returned as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix represents a frame rotation from the local NED reference frame to the body reference frame.

Data Types: `single` | `double` | `quaternion`

**velocity — Vertical velocity estimate in the local NED coordinate system (m/s)**  
scalar

---

Vertical velocity estimate in the local NED coordinate system in m/s, returned as a scalar.

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`ahrs10filter` | `insfilter`

**Introduced in R2019a**

## fusemag

Correct states using magnetometer data for `ahrs10filter`

### Syntax

```
[res, resCov] = fusemag(FUSE, magReadings, magReadingsCovariance)
```

### Description

`[res, resCov] = fusemag(FUSE, magReadings, magReadingsCovariance)` fuses magnetometer data to correct the state estimate.

### Input Arguments

**FUSE — ahrs10filter object**  
object

Object of `ahrs10filter`.

**magReadings — Magnetometer readings ( $\mu\text{T}$ )**  
3-element row vector

Magnetometer readings in  $\mu\text{T}$ , specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

**magReadingsCovariance — Magnetometer readings error covariance ( $\mu\text{T}^2$ )**  
scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in  $\mu\text{T}^2$ , specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

**res — Residual**  
1-by-3 vector of real values

Residual, returned as a 1-by-3 vector of real values in  $\mu\text{T}$ .

Data Types: `single` | `double`

**resCov — Residual covariance**  
3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in  $(\mu\text{T})^2$ .

### Extended Capabilities

#### C/C++ Code Generation

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

**See Also**

[ahrs10filter](#) | [insfilter](#)

**Introduced in R2019a**

# residualmag

Residuals and residual covariance from magnetometer measurements for `ahrs10filter`

## Syntax

```
[res, resCov] = residualmag(FUSE, magReadings, magReadingsCovariance)
```

## Description

`[res, resCov] = residualmag(FUSE, magReadings, magReadingsCovariance)` computes the residual, residual, and the residual covariance, `resCov`, based on the magnetometer readings and the corresponding covariance.

## Input Arguments

**FUSE — ahrs10filter**  
ahrs10filter | object

`ahrs10filter`, specified as an object.

**magReadings — Magnetometer readings ( $\mu\text{T}$ )**  
3-element row vector

Magnetometer readings in  $\mu\text{T}$ , specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

**magReadingsCovariance — Magnetometer readings error covariance ( $\mu\text{T}^2$ )**  
scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in  $\mu\text{T}^2$ , specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

**res — Residual**  
1-by-3 vector of real values

Residual, returned as a 1-by-3 vector of real values in  $\mu\text{T}$ .

Data Types: `single` | `double`

**resCov — Residual covariance**  
3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in  $(\mu\text{T})^2$ .

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`ahrs10filter`

Introduced in R2020a

## fusealtimeter

Correct states using altimeter data for `ahrs10filter`

### Syntax

```
[res,resCov] = fusealtimeter(FUSE,altimeterReadings,  
altimeterReadingsCovariance)
```

### Description

```
[res,resCov] = fusealtimeter(FUSE,altimeterReadings,  
altimeterReadingsCovariance) fuses altimeter data to correct the state estimate.
```

### Input Arguments

**FUSE — ahrs10filter object**  
object

Object of `ahrs10filter`.

**altimeterReadings — Altimeter readings (m)**  
real scalar

Altimeter readings in meters, specified as a real scalar.

Data Types: `single` | `double`

**altimeterReadingsCovariance — Altimeter readings error covariance (m<sup>2</sup>)**  
real scalar

Altimeter readings error covariance in m<sup>2</sup>, specified as a real scalar.

Data Types: `single` | `double`

### Output Arguments

**res — Measurement residual**  
scalar

Measurement residual, returned as a scalar in meters.

**resCov — Residual covariance**  
nonnegative scalar

Residual covariance, returned as a nonnegative scalar in m<sup>2</sup>.

### Extended Capabilities

#### C/C++ Code Generation

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

**See Also**

[ahrs10filter](#) | [insfilter](#)

**Introduced in R2019a**

# residualaltimeter

Residuals and residual covariance from altimeter measurements for `ahrs10filter`

## Syntax

```
[res,resCov] = residualaltimeter(FUSE,altimeterReadings,  
altimeterReadingsCovariance)
```

## Description

`[res,resCov] = residualaltimeter(FUSE,altimeterReadings,  
altimeterReadingsCovariance)` computes the residual, `res`, and the innovation covariance, `resCov`, based on the magnetometer readings and the corresponding covariance.

## Input Arguments

### FUSE — ahrs10filter

`ahrs10filter` | object

`ahrs10filter`, specified as an object.

### altimeterReadings — Altimeter readings (m)

real scalar

Altimeter readings in meters, specified as a real scalar.

Data Types: `single` | `double`

### altimeterReadingsCovariance — Altimeter readings error covariance (m<sup>2</sup>)

real scalar

Altimeter readings error covariance in m<sup>2</sup>, specified as a real scalar.

Data Types: `single` | `double`

## Output Arguments

### res — Measurement residual

scalar

Measurement residual, returned as a scalar in meters.

### resCov — Residual covariance

nonnegative scalar

Residual covariance, returned as a nonnegative scalar in m<sup>2</sup>.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`ahrs10filter` | `insfilter`

Introduced in R2020a

## correct

Correct states using direct state measurements for `ahrs10filter`

### Syntax

```
correct(FUSE,idx,measurement,measurementCovariance)
```

### Description

`correct(FUSE,idx,measurement,measurementCovariance)` corrects the state and state estimation error covariance based on the measurement and measurement covariance. The measurement maps directly to the state specified by the indices `idx`.

### Input Arguments

**FUSE — ahrs10filter object**  
object

Object of `ahrs10filter`.

**idx — State vector index of measurement to correct**  
*N*-element vector of increasing integers in the range [1,18]

State vector index of measurement to correct, specified as an *N*-element vector of increasing integers in the range [1,18].

The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Altitude (NED)	m	5
Vertical Velocity (NED)	m/s	6
Delta Angle Bias (XYZ)	rad/s	7:9
Delta Velocity Bias (XYZ)	m/s	10:12
Geomagnetic Field Vector (NED)	$\mu$ T	13:15
Magnetometer Bias (XYZ)	$\mu$ T	16:18

Data Types: `single` | `double`

**measurement — Direct measurement of state**  
*N*-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

**measurementCovariance — Covariance of measurement**scalar |  $N$ -element vector |  $N$ -by- $N$  matrix

Covariance of measurement, specified as a scalar,  $N$ -element vector, or  $N$ -by- $N$  matrix.  $N$  is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`ahrs10filter` | `insfilter`

**Introduced in R2019a**

# stateinfo

Display state vector information for `ahrs10filter`

## Syntax

```
stateinfo(FUSE)
info = stateinfo(FUSE)
```

## Description

`stateinfo(FUSE)` displays the meaning of each index of the `State` property and the associated units.

`info = stateinfo(FUSE)` returns a structure with fields containing descriptions of the elements of the state vector of the filter, `FUSE`.

## Examples

### State information of `ahrs10filter`

Create an `ahrs10filter` object.

```
filter = ahrs10filter;
```

Display the state information of the created filter.

```
stateinfo(filter)
```

States	Units	Index
Orientation (quaternion parts)		1:4
Altitude (NAV)	m	5
Vertical Velocity (NAV)	m/s	6
Delta Angle Bias (XYZ)	rad	7:9
Delta Velocity Bias (XYZ)	m/s	10:12
Geomagnetic Field Vector (NAV)	µT	13:15
Magnetometer Bias (XYZ)	µT	16:18

Output the state information of the filter as a structure.

```
info = stateinfo(filter)

info = struct with fields:
    Orientation: [1 2 3 4]
        Altitude: 5
    VerticalVelocity: 6
        DeltaAngleBias: [7 8 9]
    DeltaVelocityBias: [10 11 12]
    GeomagneticFieldVector: [13 14 15]
        MagnetometerBias: [16 17 18]
```

## Input Arguments

**FUSE — ahrs10filter object**  
object

Object of `ahrs10filter`.

## Output Arguments

**info — State information**  
structure

State information, returned as a structure with fields containing descriptions of the elements of the state vector of the filter. The values of each field are the corresponding indices of the state vector.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

`ahrs10filter` | `insfilter`

**Introduced in R2019a**

# residual

Residuals and residual covariances from direct state measurements for `ahrs10filter`

## Syntax

```
[res, resCov] = residual(FUSE, idx, measurement, measurementCovariance)
```

## Description

`[res, resCov] = residual(FUSE, idx, measurement, measurementCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the direct state measurement and measurement covariance. The `measurement` maps directly to the states specified by indices, `idx`.

## Input Arguments

### FUSE — ahrs10filter

`ahrs10filter` | object

`ahrs10filter`, specified as an object.

### idx — State vector index of measurement to correct

`N`-element vector of increasing integers in the range [1,18]

State vector index of measurement to correct, specified as an `N`-element vector of increasing integers in the range [1,18].

The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Altitude (NED)	m	5
Vertical Velocity (NED)	m/s	6
Delta Angle Bias (XYZ)	rad/s	7:9
Delta Velocity Bias (XYZ)	m/s	10:12
Geomagnetic Field Vector (NED)	$\mu$ T	13:15
Magnetometer Bias (XYZ)	$\mu$ T	16:18

### measurement — Direct measurement of state

`N`-element vector

Direct measurement of state, specified as a `N`-element vector. `N` is the number of elements of the index argument, `idx`.

### measurementCovariance — Covariance of measurement

`N`-by-`N` matrix

Covariance of measurement, specified as an `N`-by-`N` matrix. `N` is the number of elements of the index argument, `idx`.

## Output Arguments

### **res — Measurement residual**

1-by- $N$  vector of real values

Measurement residual, returned as a 1-by- $N$  vector of real values.

### **resCov — Residual covariance**

$N$ -by- $N$  matrix of real values

Residual covariance, returned as a  $N$ -by- $N$  matrix of real values.

## Extended Capabilities

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

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

## See Also

`ahrs10filter`

**Introduced in R2020a**

# trackingSensorConfiguration

Represent sensor configuration for tracking

## Description

The `trackingSensorConfiguration` object creates the configuration for a sensor used with a `trackerPHD` System object™. It allows you to specify the sensor parameters such as clutter density, sensor limits, sensor resolution. You can also specify how a tracker perceives the detections from the sensor using properties such as `FilterInitializationFcn`, `SensorTransformFcn`, and `SensorTransformParameters`. See “Create a Tracking Sensor Configuration” on page 2-32 for more details. The `trackingSensorConfiguration` object enables the tracker to perform three main routine operations:

- Evaluate the probability of detection at points in state-space.
- Initiate components in the probability hypothesis density.
- Obtain the clutter density of the sensor.

## Creation

### Syntax

```
config = trackingSensorConfiguration(SensorIndex)
config = trackingSensorConfiguration(SensorIndex,Name,Value)
```

#### Description

`config = trackingSensorConfiguration(SensorIndex)` creates a `trackingSensorConfiguration` object with a specified sensor index, `SensorIndex`, and default property values.

`config = trackingSensorConfiguration(SensorIndex,Name,Value)` allows you to set properties using one or more name-value pairs.

### Properties

#### **SensorIndex — Unique sensor identifier**

positive integer

Unique sensor identifier, specified as a positive integer. This property distinguishes detections that come from different sensors in a multi-sensor system. When creating a `trackingSensorConfiguration` object, you must specify the `SensorIndex` as the first input argument in the creation syntax.

Example: 2

Data Types: double

**IsValidTime — Indicate detection reporting status**`false` (default) | `true`

Indicate the detection reporting status of the sensor, specified as `false` or `true`. Set this property to `true` when the sensor must report detections within its sensor limits to the tracker. If a track or target was supposed to be detected by a sensor but the sensor reported no detections, then this information is used to count against the probability of existence of the track when the `isValidTime` property is set to `true`.

Data Types: logical

**FilterInitializationFcn — Filter initialization function**`@initcvrggiwphd` (default) | function handle | character vector

Filter initialization function, specified as a function handle or as a character vector containing the name of a valid filter initialization function. The function initializes the PHD filter used by `trackerPHD`. The function must support the following syntaxes:

```
filter = filterInitializationFcn()
filter = filterInitializationFcn(detections)
```

`filter` is a valid PHD filter with components for new-born targets, and `detections` is a cell array of `objectDetection` objects. The first syntax allows you to specify the predictive birth density in the PHD filter without using detections. The second syntax allows the filter to initialize the adaptive birth density using detection information. See the “BirthRate” on page 3-0 property of `trackerPHD` for more details. If you create your own `FilterInitializationFcn`, you must also provide a transform function using the `SensorTransformFcn` property. Other than the default filter initialization function `initcvrggiwphd`, Sensor Fusion and Tracking Toolbox also provides other initialization functions, such as `initctrectgmphd`, `initctgmphd`, `initcvgmphd`, `initcagmhd`, `initctggiwphd` and `initcaggiwphd`.

Data Types: function\_handle | char

**SensorTransformFcn — Sensor transform function**`@cvmeas` | function handle | character vector

Sensor transform function, specified as a function handle or as a character vector containing the name of a valid sensor transform function. The function transforms a track's state into the sensor's detection state. For example, the function transforms the track's state in the scenario Cartesian frame to the sensor's spherical frame. You can create your own sensor transform function, but it must support the following syntax:

```
detStates = SensorTransformFcn(trackStates,params)
```

`params` are the parameters stored in the `SensorTransformParameters` property. Notice that the signature of the function is similar to a measurement function. Therefore, you can use a measurement function (such as `cvmeas`, `ctmeas`, or `cameas`) as the `SensorTransformFcn`.

Depending on the filter type and the target type, the output, `detStates`, needs to return differently.

- When used with `gmphd` for non-extended targets or with `ggiwphd`, `detStates` is a  $N$ -by- $M$  matrix, where  $N$  is the number of rows in the `SensorLimits` property and  $M$  is the number of input states in `trackStates`. For `gmphd`, non-extended targets refer to point targets and extended targets whose `MeasurementOrigin` is 'center'.
- When used with `gmphd` for extended targets, the `SensorTransformFcn` allows you to specify multiple `detStates` per `trackState`. In this case, `detStates` is a  $N$ -by- $M$ -by- $S$  matrix, where  $S$

is the number of detectable sources on the extended target. For example, if the target is described by a rectangular state, the detectable sources can be the corners of the rectangle.

If any of the source falls inside the `SensorLimits`, the target is declared detectable. The functions uses the spread (maximum coordinate – minimum coordinate) of each `detStates` and the ratio between the spread and sensor resolution on each sensor limit to calculate the expected number of detections from each extended target. You can override this default setting by providing an optional output in the `SensorTransformFcn` as:

```
[..., Nexp] = SensorTransformFcn(trackStates, params)
```

where `Nexp` is the expected number of detections from each extended track state.

Note that the default `SensorTransformFcn` is the sensor transform function of the filter returned by `FilterInitializationFcn`. For example, the `initicvggiwphd` function returns the default `cvmeas`, whereas `initictggiwphd` and `initicaggiwphd` functions return `ctmeas` and `cameas`, respectively.

Data Types: `function_handle | char`

#### **SensorTransformParameters — Parameters for sensor transform function**

structure | array of structures

Parameters for the sensor transform function, returned as a structure or an array of structures. If you only need to transform the state once, specify it as a structure. If you need to transform the state  $n$  times, specify it as an  $n$ -by-1 array of structures. For example, to transform a state from the scenario frame to the sensor frame, you usually need to first transform the state from the scenario rectangular frame to the platform rectangular frame, and then transform the state from the platform rectangular frame to the sensor spherical frame. The fields of the structure are:

Field	Description
<code>Frame</code>	Child coordinate frame type, specified as ' <code>Rectangular</code> ' or ' <code>Spherical</code> '.
<code>OriginPosition</code>	Child frame origin position expressed in the Parent frame, specified as a 3-by-1 vector.
<code>OriginVelocity</code>	Child frame origin velocity expressed in the parent frame, specified as a 3-by-1 vector.
<code>Orientation</code>	Relative orientation between frames, specified as a 3-by-3 rotation matrix. If the <code>IsParentToChild</code> property is set to <code>false</code> , then specify <code>Orientation</code> as the rotation from the child frame to the parent frame. If the <code>IsParentToChild</code> property is set to <code>true</code> , then specify <code>Orientation</code> as the rotation from the parent frame to the child frame.
<code>IsParentToChild</code>	Flag to indicate the direction of rotation between parent and child frame, specified as <code>true</code> or <code>false</code> . The default is <code>false</code> . See description of the <code>Orientation</code> field for details.
<code>HasAzimuth</code>	Indicates whether outputs contain azimuth components, specified as <code>true</code> or <code>false</code> .

<b>HasElevation</b>	Indicates whether outputs contain elevation components, specified as <code>true</code> or <code>false</code> .
<b>HasRange</b>	Indicates whether outputs contain range components, specified as <code>true</code> or <code>false</code> .
<b>HasVelocity</b>	Indicates whether outputs contains velocity components, specified as <code>true</code> or <code>false</code> .

Note that here the scenario frame is the parent frame of the platform frame, and the platform frame is the parent frame of the sensor frame.

The default values for `SensorTransformParameters` are a 2-by-1 array of structures as:

<b>Fields</b>	<b>Struct 1</b>	<b>Struct 2</b>
Frame	'Spherical'	'Rectangular'
OriginPosition	[0;0;0]	[0;0;0]
OriginVelocity	[0;0;0]	[0;0;0]
Orientation	<code>eye(3)</code>	<code>eye(3)</code>
IsParentToChild	<code>false</code>	<code>false</code>
HasAzimuth	<code>true</code>	<code>true</code>
HasElevation	<code>true</code>	<code>true</code>
HasRange	<code>true</code>	<code>true</code>
HasVelocity	<code>false</code>	<code>true</code>

In this table, Struct 2 accounts for the transformation from the scenario rectangular frame to the platform rectangular frame, and Struct 1 accounts for the transformation from the platform rectangular frame to the sensor spherical frame, given the `isParentToChild` property is set to `false`.

Data Types: `struct`

#### **SensorLimits — Sensor's detection limits**

3-by-2 matrix (default) |  $N$ -by-2 matrix

Sensor's detection limits, specified as an  $N$ -by-2 matrix, where  $N$  is the output dimension of the sensor transform function. The matrix must describe the lower and upper detection limits of the sensor in the same order as the outputs of the sensor transform function.

If you use `cvmeas`, `cameas`, or `ctmeas` as the sensor transform function, then you need to provide the sensor limits in order as:

$$\text{SensorLimits} = \begin{bmatrix} \text{minAz} & \text{maxAz} \\ \text{minEl} & \text{maxEl} \\ \text{minRng} & \text{maxRng} \\ \text{minRr} & \text{maxRr} \end{bmatrix}$$

The description of these limits and their default values are given in the following table. Note that the default values for `SensorLimits` are a 3-by-2 matrix including the top six elements in the table. Moreover, if you use these three functions, you can specify the matrix to be in other sizes (1-by-2, 2-by-2, or 3-by-4), but you have to specify these limits in the sequence shown in the `SensorLimits` matrix.

Limits	Description	Default values
<code>minAz</code>	Minimum detectable azimuth in degrees.	-10
<code>maxAz</code>	Maximum detectable azimuth in degrees.	10
<code>minEl</code>	Minimum detectable elevation in degrees.	-2.5
<code>maxEl</code>	Maximum detectable elevation in degrees.	2.5
<code>minRng</code>	Minimum detectable range in meters.	0
<code>maxRng</code>	Maximum detectable range in meters.	1000
<code>minRr</code>	Minimum detectable range rate in meters per second.	N/A
<code>maxRr</code>	Maximum detectable range rate in meters per second.	N/A

Data Types: double

#### **SensorResolution — Resolution of sensor**

[4;2;10] (default) | *N*-element positive-valued vector

Resolution of a sensor, specified as a *N*-element positive-valued vector, where *N* is the number of parameters specified in the `SensorLimits` property. If you want to assign only one resolution cell for a parameter, simply specify its resolution as the difference between the maximum limit and the minimum limit of the parameter.

Data Types: double

#### **MaxNumDetsPerObject — Maximum number of detections per object**

Inf (default) | positive integer

Maximum number of detections the sensor can report per object, specified as a positive integer.

Example: 3

Data Types: double

#### **ClutterDensity — Expected number of false alarms per unit volume**

1e-3 (default) | positive scalar

Expected number of false alarms per unit volume from the sensor, specified as a positive scalar.

Example: 2e-3

Data Types: double

#### **MinDetectionProbability — Probability of detecting track estimated to be outside of sensor limits**

0.05 (default) | positive scalar

Probability of detecting a target estimated to be outside of the sensor limits, specified as a positive scalar. This property allows a `trackerPHD` object to consider that the estimated target, which is outside the sensor limits, may be detectable.

Example: 0.03

Data Types: double

## **Examples**

### **Create Radar Sensor Configuration**

Consider a radar with the following sensor limits and sensor resolution.

```
azLimits = [-10 10];
elLimits = [-2.5 2.5];
rangeLimits = [0 500];
rangeRateLimits = [-50 50];
sensorLimits = [azLimits;elLimits;rangeLimits;rangeRateLimits];
sensorResolution = [5 2 10 3];
```

Specifying the sensor transform function that transforms the Cartesian coordinates [x;y;vx;vy] in the scenario frame to the spherical coordinates [az;el;range;rr] in the sensor's frame. You can use the measurement function `cvmeas` as the sensor transform function.

```
transformFcn = @cvmeas;
```

To specify the parameters required for `cvmeas`, use the `SensorTransformParameters` property. Here, you assume the sensor is mounted at the center of the platform and the platform located at [100;30;20] is moving with a velocity of [-5;4;2] units per second in the scenario frame.

The first structure defines the sensor's location, velocity, and orientation in the platform frame.

```
params(1) = struct('Frame','Spherical','OriginPosition',[0;0;0],...
    'OriginVelocity',[0;0;0],'Orientation',eye(3),'HasRange',true, ...
    'HasVelocity',true);
```

The second structure defines the platform's location, velocity, and orientation in the scenario frame.

```
params(2) = struct('Frame','Rectangular','OriginPosition',[100;30;20],...
    'OriginVelocity',[-5;4;2],'Orientation',eye(3),'HasRange',true, ...
    'HasVelocity',true);
```

Create the configuration.

```
config = trackingSensorConfiguration('SensorIndex',3,'SensorLimits',sensorLimits, ...
    'SensorResolution',sensorResolution, ...
    'SensorTransformParameters',params, ...
    'SensorTransformFcn',@cvmeas, ...
    'FilterInitializationFcn',@initcvrggiwphd)

config =
    trackingSensorConfiguration with properties:

        SensorIndex: 3
        IsValidTime: 0

        SensorLimits: [4x2 double]
        SensorResolution: [4x1 double]
        SensorTransformFcn: @cvmeas
        SensorTransformParameters: [1x2 struct]

        FilterInitializationFcn: @initcvrggiwphd
        MaxNumDetsPerObject: Inf

        ClutterDensity: 1.0000e-03
        DetectionProbability: 0.9000
        MinDetectionProbability: 0.0500
```

## More About

### Create a Tracking Sensor Configuration

To create the configuration for a sensor, you first need to specify the sensor transform function, which is usually given as:

$$Y = g(x, p)$$

where  $x$  denotes the tracking state,  $Y$  denotes detection states, and  $p$  denotes the required parameters. For object tracking applications, you mainly focus on obtaining an object's tracking state. For example, a radar sensor can measure an object's azimuth, elevation, range, and possibly range-rate. Using a `trackingSensorConfiguration` object, you can specify a radar's transform function using the `SensorTransformFcn` property and specify the radar's mounting location, orientation, and velocity using corresponding fields in the `SensorTransformParameters` property. If the object is moving at a constant velocity, constant acceleration, or constant turning, you can use the built-in measurement function - `cvmeas`, `cameas`, or `ctmeas`, respectively - as the `SensorTransformFcn`. To set up the exact outputs of these three functions, specify the `hasAzimuth`, `hasElevation`,

`hasRange`, and `hasVelocity` fields as `true` or `false` in the `SensorTransformParameters` property.

To set up the configuration of a sensor, you also need to specify the sensor's detection ability. Primarily, you need to specify the sensor's detection limits. For all the outputs of the sensor transform function, you need to provide the detection limits in the same order of these outputs using the `SensorLimits` property. For example, for a radar sensor, you might need to provide its azimuth, elevation, range, and range-rate limits. You can also specify the radar's `SensorResolution` and `MaxNumDetsPerObject` properties if you want to consider extended object detection. You might also want to specify other properties, such as `ClutterDensity`, `IsValidTime`, and `MinDetectionProbability` to further clarify the sensor's detection ability.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`cameas` | `ctmeas` | `cvmeas` | `ggiwphd` | `trackerPHD`

**Introduced in R2019a**

## pose

Current orientation and position estimate for `insfilterErrorState`

### Syntax

```
[position,orientation,velocity] = pose(FUSE)
[position,orientation,velocity] = pose(FUSE,format)
```

### Description

`[position,orientation,velocity] = pose(FUSE)` returns the current estimate of the pose of the object tracked by FUSE, an `insfilterErrorState` object.

`[position,orientation,velocity] = pose(FUSE,format)` returns the current estimate of the pose with orientation in the specified orientation format.

### Input Arguments

#### FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

#### format — Output orientation format

'quaternion' (default) | 'rotmat'

Output orientation format, specified as either 'quaternion' for a quaternion or 'rotmat' for a rotation matrix.

Data Types: `char` | `string`

### Output Arguments

#### position — Position estimate in local NED coordinate system (m)

3-element row vector

Position estimate in the local NED coordinate system in meters, returned as a 3-element row vector.

Data Types: `single` | `double`

#### orientation — Orientation estimate in local NED coordinate system

quaternion (default) | 3-by-3 rotation matrix

Orientation estimate in the local NED coordinate system, returned as a scalar quaternion or 3-by-3 rotation matrix, depending on the specified orientation `format`. The quaternion or rotation matrix represents a frame rotation from the local NED reference frame to the body reference frame.

Data Types: `single` | `double` | `quaternion`

#### velocity — Velocity estimate in local NED coordinate system (m/s)

3-element row vector

---

Velocity estimate in the local NED coordinate system in m/s, returned as a 3-element row vector.

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`insfilter` | `insfilterErrorState`

**Introduced in R2019a**

## copy

Create copy of insfilterErrorState

### Syntax

```
newFilter = copy(filter)
```

### Description

`newFilter = copy(filter)` returns a copy of the `insfilterErrorState`, `filter`, with the exactly same property values.

### Input Arguments

#### **filter — Filter to be copied**

`insfilterErrorState`

Filter to be copied, specified as an `insfilterErrorState` object.

### Output Arguments

#### **newFilter — New copied filter**

`insfilterErrorState`

New copied filter, returned as an `insfilterErrorState` object.

### See Also

`insfilterErrorState`

### Introduced in R2020b

# stateinfo

Display state vector information for `insfilterErrorState`

## Syntax

```
stateinfo(FUSE)
info = stateinfo(FUSE)
```

## Description

`stateinfo(FUSE)` displays the meaning of each index of the `State` property of `FUSE`, an `insfilterErrorState` object, and the associated units.

`info = stateinfo(FUSE)` returns a structure with fields containing descriptions of the elements of the state vector of the filter, `FUSE`.

## Examples

### State information of `insfilterErrorState`

Create an `insfilterErrorState` object.

```
filter = insfilterErrorState;
```

Display the state information of the created filter.

```
stateinfo(filter)
```

States	Units	Index
Orientation (quaternion parts)		1:4
Position (NAV)	m	5:7
Velocity (NAV)	m/s	8:10
Gyroscope Bias (XYZ)	rad/s	11:13
Accelerometer Bias (XYZ)	m/s <sup>2</sup>	14:16
Visual Odometry Scale		17

Output the state information of the filter as a structure.

```
info = stateinfo(filter)

info = struct with fields:
    Orientation: [1 2 3 4]
    Position: [5 6 7]
    Velocity: [8 9 10]
    GyroscopeBias: [11 12 13]
    AccelerometerBias: [14 15 16]
    VisualOdometryScale: 17
```

## Input Arguments

### FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

## Output Arguments

### info — State information

structure

State information, returned as a structure with fields containing descriptions of the elements of the state vector of the filter. The values of each field are the corresponding indices of the state vector.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

`insfilter` | `insfilterErrorState`

**Introduced in R2019a**

# reset

Reset internal states for `insfilterErrorState`

## Syntax

```
reset(FUSE)
```

## Description

`reset(FUSE)` resets the `State`, `StateCovariance`, and internal integrators of `FUSE`, an `insfilterErrorState` object, to their default values.

## Input Arguments

### **FUSE — INS filter object**

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

## Extended Capabilities

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

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

## See Also

`insfilter` | `insfilterErrorState`

**Introduced in R2019a**

# **predict**

Update states using accelerometer and gyroscope data for `insfilterErrorState`

## **Syntax**

```
predict(FUSE,accelReadings,gyroReadings)
```

## **Description**

`predict(FUSE,accelReadings,gyroReadings)` fuses accelerometer and gyroscope data to update the state estimate.

## **Input Arguments**

### **FUSE — INS filter object**

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

### **accelReadings — Accelerometer readings in local sensor body coordinate system (m/s<sup>2</sup>)**

3-element row vector

Accelerometer readings in m/s<sup>2</sup>, specified as a 3-element row vector.

Data Types: `single` | `double`

### **gyroReadings — Gyroscope readings in local sensor body coordinate system (rad/s)**

3-element row vector

Gyroscope readings in rad/s, specified as a 3-element row vector.

Data Types: `single` | `double`

## **Extended Capabilities**

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

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

## **See Also**

`insfilter` | `insfilterErrorState`

## **Introduced in R2019a**

# fusemvo

Correct states using monocular visual odometry for `insfilterErrorState`

## Syntax

```
[pResidual,oResidual,resCov] = fusemvo(FUSE,position,positionCovariance,ornt,  
orntCovariance)
```

## Description

`[pResidual,oResidual,resCov] = fusemvo(FUSE,position,positionCovariance,ornt,  
orntCovariance)` fuses position and orientation data from monocular visual odometry (MVO)  
measurements to correct the state and state estimation error covariance.

## Input Arguments

### FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

### position — Position of camera in local NED coordinate system (m)

3-element row vector

Position of camera in the local NED coordinate system in meters, specified as a real finite 3-element row vector.

Data Types: `single` | `double`

### positionCovariance — Position measurement covariance of MVO (m<sup>2</sup>)

scalar | 3-element vector | 3-by-3 matrix

Position measurement covariance of MVO in m<sup>2</sup>, specified as a scalar, 3-element vector, or 3-by-3 matrix.

Data Types: `single` | `double`

### ornt — Orientation of camera with respect to local NED coordinate system

scalar quaternion | rotation matrix

Orientation of the camera with respect to the local NED coordinate system, specified as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix is a frame rotation from the NED coordinate system to the current camera coordinate system.

Data Types: `quaternion` | `single` | `double`

### orntCovariance — Orientation measurement covariance of monocular visual odometry (rad<sup>2</sup>)

scalar | 3-element vector | 3-by-3 matrix

Orientation measurement covariance of monocular visual odometry in rad<sup>2</sup>, specified as a scalar, 3-element vector, or 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **pResidual — Position residual**

1-by-3 vector of real values

Position residual, returned as a 1-by-3 vector of real values in m.

### **oResidual — Rotation vector residual**

1-by-3 vector of real values

Rotation vector residual, returned as a 1-by-3 vector of real values in radians.

### **resCov — Residual covariance**

6-by-6 matrix of real values

Residual covariance, returned as a 6-by-6 matrix of real values.

## Extended Capabilities

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

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

### See Also

`insfilter` | `insfilterErrorState`

**Introduced in R2019a**

# residualmvo

Residuals and residual covariance from monocular visual odometry measurements for `insfilterErrorState`

## Syntax

```
[pResidual,oResidual,resCov] = residualmvo(FUSE,position,positionCovariance,  
ornt,orntCovariance)
```

## Description

`[pResidual,oResidual,resCov] = residualmvo(FUSE,position,positionCovariance,  
ornt,orntCovariance)` computes the residual information based on the monocular visual odometry measurements and covariance.

## Input Arguments

### FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

### position — Position of camera in local NED coordinate system (m)

3-element row vector

Position of camera in the local NED coordinate system in meters, specified as a real finite 3-element row vector.

Data Types: `single` | `double`

### positionCovariance — Position measurement covariance of MVO (m<sup>2</sup>)

scalar | 3-element vector | 3-by-3 matrix

Position measurement covariance of MVO in m<sup>2</sup>, specified as a scalar, 3-element vector, or 3-by-3 matrix.

Data Types: `single` | `double`

### ornt — Orientation of camera with respect to local NED coordinate system

scalar quaternion | rotation matrix

Orientation of the camera with respect to the local NED coordinate system, specified as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix is a frame rotation from the NED coordinate system to the current camera coordinate system.

Data Types: `quaternion` | `single` | `double`

### orntCovariance — Orientation measurement covariance of monocular visual odometry (rad<sup>2</sup>)

scalar | 3-element vector | 3-by-3 matrix

Orientation measurement covariance of monocular visual odometry in rad<sup>2</sup>, specified as a scalar, 3-element vector, or 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **pResidual — Position residual**

1-by-3 vector of real values

Position residual, returned as a 1-by-3 vector of real values in meters.

### **oResidual — Rotation vector residual**

1-by-3 vector of real values

Rotation vector residual, returned a 1-by-3 vector of real values in radians.

### **resCov — Residual covariance**

6-by-6 matrix of real values

Residual covariance, returned as a 6-by-6 matrix of real values.

## Extended Capabilities

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

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

### See Also

`insfilter` | `insfilterErrorState`

### Introduced in R2020a

# fusegps

Correct states using GPS data for `insfilterErrorState`

## Syntax

```
[res,resCov] = fusegps(FUSE,position,positionCovariance)
[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

## Description

`[res,resCov] = fusegps(FUSE,position,positionCovariance)` fuses GPS position data to correct the state estimate.

`[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)` fuses GPS position and velocity data to correct the state estimate.

## Input Arguments

### FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

### position — Position of GPS receiver (LLA)

3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

### positionCovariance — Position measurement covariance of GPS receiver ( $\text{m}^2$ )

scalar | 3-element row vector | 3-by-3 matrix

Position measurement covariance of GPS receiver in  $\text{m}^2$ , specified as a 3-by-3 matrix.

Data Types: `single` | `double`

### velocity — Velocity of GPS receiver in local NED coordinate system (m/s)

3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

### velocityCovariance — Velocity measurement covariance of GPS receiver ( $\text{m}/\text{s}^2$ )

scalar | 3-element row vector | 3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in (m/s)<sup>2</sup>, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **res — Position and velocity residual**

1-by-6 vector of real values

Position and velocity residual, returned as a 1-by-6 vector of real values in m and m/s, respectively.

### **resCov — Innovation residual**

6-by-6 matrix of real values

Innovation residual, returned as a 6-by-6 matrix of real values.

## Extended Capabilities

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

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

## See Also

`insfilter` | `insfilterErrorState`

## Introduced in R2019a

# correct

Correct states using direct state measurements for `insfilterErrorState`

## Syntax

```
correct(FUSE,idx,measurement,measurementCovariance)
```

## Description

`correct(FUSE,idx,measurement,measurementCovariance)` corrects the state and state estimation error covariance of `FUSE`, an `insfilterErrorState` object, based on the measurement and measurement covariance. The measurement maps directly to the state specified by the indices `idx`.

## Input Arguments

### **FUSE — INS filter object**

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

### **idx — State vector index of measurements to correct**

`N`-element vector of increasing integers in the range [1, 17]

State vector index of measurements to correct, specified as an `N`-element vector of increasing integers in the range [1, 17].

The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Position (NED)	m	5:7
Velocity (NED)	m/s	8:10
Gyroscope Bias (XYZ)	rad/s	11:13
Accelerometer Bias (XYZ)	m/s <sup>2</sup>	14:16
Visual Odometry Scale (XYZ)	N/A	17

Data Types: `single` | `double`

### **measurement — Direct measurement of state**

`N`-element vector

Direct measurement of state, specified as a `N`-element vector. `N` is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

### **measurementCovariance — Covariance of measurement**

scalar | `M`-element vector | `M`-by-`M` matrix

Covariance of measurement, specified as a scalar,  $M$ -element vector, or  $M$ -by- $M$  matrix. If you correct orientation (state indices 1–4), then  $M = \text{numel}(\text{idx}) - 1$ . If you do not correct orientation, then  $M = \text{numel}(\text{idx})$ .

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`insfilter` | `insfilterErrorState`

**Introduced in R2019a**

# residual

Residuals and residual covariances from direct state measurements for `insfilterErrorState`

## Syntax

```
[res, resCov] = residual(FUSE, idx, measurement, measurementCovariance)
```

## Description

`[res, resCov] = residual(FUSE, idx, measurement, measurementCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the direct state measurement and measurement covariance. The measurement maps directly to the states specified by indices, `idx`.

## Input Arguments

### FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

### idx — State vector index of measurements to correct

`N`-element vector of increasing integers in the range [1, 17]

State vector index of measurements to correct, specified as an `N`-element vector of increasing integers in the range [1, 17].

The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Position (NED)	m	5:7
Velocity (NED)	m/s	8:10
Gyroscope Bias (XYZ)	rad/s	11:13
Accelerometer Bias (XYZ)	m/s <sup>2</sup>	14:16
Visual Odometry Scale (XYZ)	N/A	17

Data Types: `single` | `double`

### measurement — Direct measurement of state

`N`-element vector

Direct measurement of state, specified as a `N`-element vector. `N` is the number of elements of the index argument, `idx`.

### measurementCovariance — Covariance of measurement

`N`-by-`N` matrix

Covariance of measurement, specified as an  $N$ -by- $N$  matrix.  $N$  is the number of elements of the index argument, `idx`.

## Output Arguments

### **res — Measurement residual**

1-by- $N$  vector of real values

Measurement residual, returned as a 1-by- $N$  vector of real values.

### **resCov — Residual covariance**

$N$ -by- $N$  matrix of real values

Residual covariance, returned as a  $N$ -by- $N$  matrix of real values.

## Extended Capabilities

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

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

### See Also

`insfilterErrorState`

**Introduced in R2020a**

# residualgps

Residuals and residual covariance from GPS measurements for `insfilterErrorState`

## Syntax

```
[res,resCov] = residualgps(FUSE,position,positionCovariance)
[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

## Description

`[res,resCov] = residualgps(FUSE,position,positionCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

`[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,
velocityCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

## Input Arguments

### FUSE — INS filter object

`insfilterErrorState`

`insfilterErrorState`, specified as an object.

### position — Position of GPS receiver (LLA)

3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

### positionCovariance — Position measurement covariance of GPS receiver ( $\text{m}^2$ )

3-by-3 matrix

Position measurement covariance of GPS receiver in  $\text{m}^2$ , specified as a 3-by-3 matrix.

Data Types: `single` | `double`

### velocity — Velocity of GPS receiver in local NED coordinate system (m/s)

3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

### velocityCovariance — Velocity measurement covariance of GPS receiver ( $\text{m}/\text{s}^2$ )

3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s<sup>2</sup>, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **res — Position and velocity residual**

1-by-3 vector of real values | 1-by-6 vector of real values

Position and velocity residual, returned as a 1-by-3 vector of real values if the inputs only contain position information, and returned as 1-by-6 vector of real values if the inputs also contain velocity information.

### **resCov — Residual covariance**

3-by-3 matrix of real values | 6-by-6 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values if the inputs only contain position information, and a 6-by-6 matrix of real values if the inputs also contain velocity information.

## Extended Capabilities

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

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

### **See Also**

`insfilterErrorState`

**Introduced in R2020a**

# insfilterErrorState

Estimate pose from IMU, GPS, and monocular visual odometry (MVO) data

## Description

The `insfilterErrorState` object implements sensor fusion of IMU, GPS, and monocular visual odometry (MVO) data to estimate pose in the NED (or ENU) reference frame. The filter uses a 17-element state vector to track the orientation quaternion, velocity, position, IMU sensor biases, and the MVO scaling factor. The `insfilterErrorState` object uses an error-state Kalman filter to estimate these quantities.

## Creation

### Syntax

```
filter = insfilterErrorState
filter = insfilterErrorState('ReferenceFrame',RF)
filter = insfilterErrorState(___,Name,Value)
```

### Description

`filter = insfilterErrorState` creates an `insfilterErrorState` object with default property values.

`filter = insfilterErrorState('ReferenceFrame',RF)` allows you to specify the reference frame, RF, of the filter. Specify RF as 'NED' (North-East-Down) or 'ENU' (East-North-Up). The default value is 'NED'.

`filter = insfilterErrorState(___,Name,Value)` also allows you set properties of the created filter using one or more name-value pairs. Enclose each property name in single quotes.

## Properties

### IMUSampleRate — Sample rate of IMU (Hz)

100 (default) | positive scalar

Sample rate of the inertial measurement unit (IMU) in Hz, specified as a positive scalar.

Data Types: `single` | `double`

### ReferenceLocation — Reference location (deg, deg, meters)

[0 0 0] (default) | 3-element positive row vector

Reference location, specified as a 3-element row vector in geodetic coordinates (latitude, longitude, and altitude). Altitude is the height above the reference ellipsoid model, WGS84. The reference location units are [degrees degrees meters].

Data Types: `single` | `double`

**GyroscopeNoise — Multiplicative process noise variance from gyroscope ((rad/s)<sup>2</sup>)**

[1e-6 1e-6 1e-6] (default) | scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope in (rad/s)<sup>2</sup>, specified as a scalar or 3-element row vector of positive real finite numbers.

- If GyroscopeNoise is specified as a row vector, the elements correspond to the noise in the x, y, and z axes of the gyroscope, respectively.
- If GyroscopeNoise is specified as a scalar, the single element is applied to each axis.

Data Types: single | double

**GyroscopeBiasNoise — Additive process noise variance from gyroscope bias ((rad/s)<sup>2</sup>)**

[1e-9 1e-9 1e-9] (default) | scalar | 3-element row vector

Additive process noise variance from the gyroscope bias in (rad/s)<sup>2</sup>, specified as a scalar or 3-element row vector of positive real finite numbers.

- If GyroscopeBiasNoise is specified as a row vector, the elements correspond to the noise in the x, y, and z axes of the gyroscope, respectively.
- If GyroscopeBiasNoise is specified as a scalar, the single element is applied to each axis

Data Types: single | double

**AccelerometerNoise — Multiplicative process noise variance from accelerometer ((m/s<sup>2</sup>)<sup>2</sup>)**

[1e-4 1e-4 1e-4] (default) | scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer in (m/s<sup>2</sup>)<sup>2</sup>, specified as a scalar or 3-element row vector of positive real finite numbers.

- If AccelerometerNoise is specified as a row vector, the elements correspond to the noise in the x, y, and z axes of the accelerometer, respectively.
- If AccelerometerNoise is specified as a scalar, the single element is applied to each axis.

Data Types: single | double

**AccelerometerBiasNoise — Additive process noise variance from accelerometer bias ((m/s<sup>2</sup>)<sup>2</sup>)**

[1e-4 1e-4 1e-4] (default) | scalar | 3-element row vector

Additive process noise variance from accelerometer bias in (m/s<sup>2</sup>)<sup>2</sup>, specified as a scalar or 3-element row vector of positive real numbers.

- If AccelerometerBiasNoise is specified as a row vector, the elements correspond to the noise in the x, y, and z axes of the accelerometer, respectively.
- If AccelerometerBiasNoise is specified as a scalar, the single element is applied to each axis.

**State — State vector of Kalman filter**

[1;zeros(15,1);1] (default) | 17-element column vector

State vector of the extended Kalman filter, specified as a 17-element column vector. The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Position (NED or ENU)	m	5:7
Velocity (NED or ENU)	m/s	8:10
Gyroscope Bias (XYZ)	rad/s	11:13
Accelerometer Bias (XYZ)	m/s <sup>2</sup>	14:16
Visual Odometry Scale (XYZ)	N/A	17

The default initial state corresponds to an object at rest located at [0 0 0] in geodetic LLA coordinates.

Data Types: `single` | `double`

#### **StateCovariance — State error covariance for Kalman filter**

`ones(16)` (default) | 16-by-16 matrix

State error covariance for the Kalman filter, specified as a 16-by-16-element matrix of real numbers. The state error covariance values represent:

State Covariance	Row/Column Index
6 Rotation Vector (XYZ)	1:3
6 Position (NED or ENU)	4:6
6 Velocity (NED or ENU)	7:9
6 Gyroscope Bias (XYZ)	10:12
6 Accelerometer Bias (XYZ)	13:15
6 Visual Odometry Scale (XYZ)	16

Note that because this is an error-state Kalman filter, it tracks the errors in the states. 6 represents the error in the corresponding state.

Data Types: `single` | `double`

#### **Object Functions**

<code>predict</code>	Update states using accelerometer and gyroscope data for <code>insfilterErrorState</code>
<code>correct</code>	Correct states using direct state measurements for <code>insfilterErrorState</code>
<code>residual</code>	Residuals and residual covariances from direct state measurements for <code>insfilterErrorState</code>
<code>fusegps</code>	Correct states using GPS data for <code>insfilterErrorState</code>
<code>residualgps</code>	Residuals and residual covariance from GPS measurements for <code>insfilterErrorState</code>
<code>fusemvo</code>	Correct states using monocular visual odometry for <code>insfilterErrorState</code>
<code>residualmvo</code>	Residuals and residual covariance from monocular visual odometry measurements for <code>insfilterErrorState</code>
<code>pose</code>	Current orientation and position estimate for <code>insfilterErrorState</code>
<code>reset</code>	Reset internal states for <code>insfilterErrorState</code>
<code>stateinfo</code>	Display state vector information for <code>insfilterErrorState</code>
<code>copy</code>	Create copy of <code>insfilterErrorState</code>

#### **Examples**

## Estimate Pose of Ground Vehicle

Load logged data of a ground vehicle following a circular trajectory. The .mat file contains IMU and GPS sensor measurements and ground truth orientation and position.

```
load('loggedGroundVehicleCircle.mat', ...
    'imuFs', 'localOrigin', ...
    'initialStateCovariance', ...
    'accelData', 'gyroData', ...
    'gpsFs', 'gpsLLA', 'Rpos', 'gpsVel', 'Rvel', ...
    'trueOrient', 'truePos');
```

Create an INS filter to fuse IMU and GPS data using an error-state Kalman filter.

```
initialState = [compact(trueOrient(1)),truePos(1,:),-6.8e-3,2.5002,0,zeros(1,6),1].';
filt = insfilterErrorState;
filt.IMUSampleRate = imuFs;
filt.ReferenceLocation = localOrigin;
filt.State = initialState;
filt.StateCovariance = initialStateCovariance;
```

Preallocate variables for position and orientation. Allocate a variable for indexing into the GPS data.

```
numIMUSamples = size(accelData,1);
estOrient = ones(numIMUSamples,1,'quaternion');
estPos = zeros(numIMUSamples,3);

gpsIdx = 1;
```

Fuse accelerometer, gyroscope, and GPS data. The outer loop predicts the filter forward at the fastest sample rate (the IMU sample rate).

```
for idx = 1:numIMUSamples

    % Use predict to estimate the filter state based on the accelData and
    % gyroData arrays.
    predict(filt,accelData(idx,:),gyroData(idx,:));

    % GPS data is collected at a lower sample rate than IMU data. Fuse GPS
    % data at the lower rate.
    if mod(idx, imuFs / gpsFs) == 0
        % Correct the filter states based on the GPS data.
        fusegps(filt,gpsLLA(gpsIdx,:),Rpos,gpsVel(gpsIdx,:),Rvel);
        gpsIdx = gpsIdx + 1;
    end

    % Log the current pose estimate
    [estPos(idx,:), estOrient(idx,:)] = pose(filt);
end
```

Calculate the RMS errors between the known true position and orientation and the output from the error-state filter.

```
pErr = truePos - estPos;
qErr = rad2deg(dist(estOrient,trueOrient));

pRMS = sqrt(mean(pErr.^2));
qRMS = sqrt(mean(qErr.^2));
```

```
fprintf('Position RMS Error\n');

Position RMS Error

fprintf('\tX: %.2f, Y: %.2f, Z: %.2f (meters)\n\n',pRMS(1),pRMS(2),pRMS(3));

X: 0.40, Y: 0.24, Z: 0.05 (meters)

fprintf('Quaternion Distance RMS Error\n');

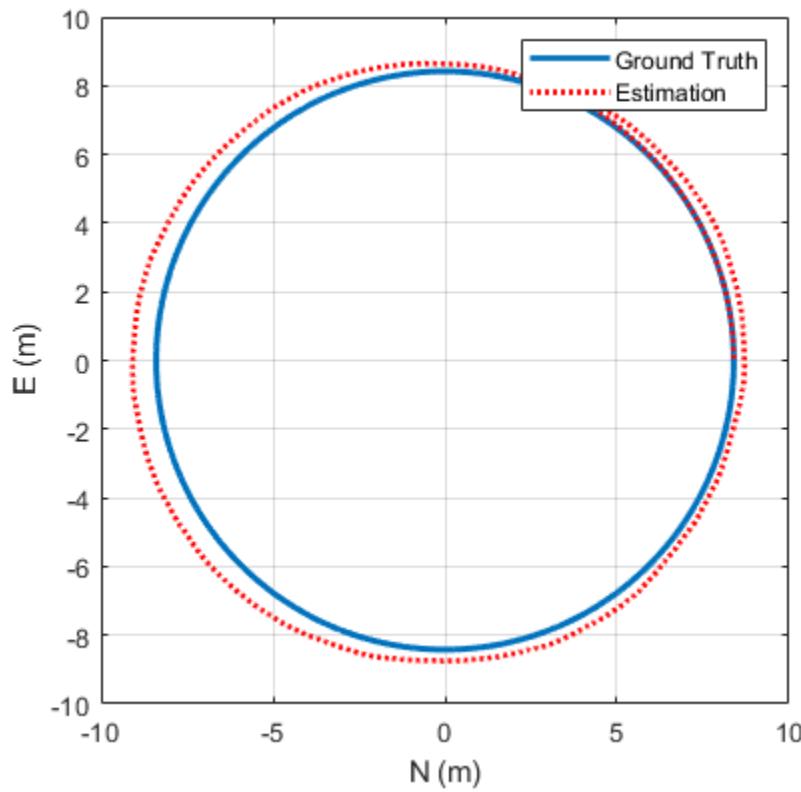
Quaternion Distance RMS Error

fprintf('\t%.2f (degrees)\n\n',qRMS);

0.30 (degrees)
```

Visualize the true position and the estimated position.

```
plot(truePos(:,1),truePos(:,2),estPos(:,1),estPos(:,2),'r','LineWidth',2)
grid on
axis square
xlabel('N (m)')
ylabel('E (m)')
legend('Ground Truth','Estimation')
```



## Algorithms

*Note: The following algorithm only applies to an NED reference frame.*

`insfilterErrorState` uses a 17-axis error state Kalman filter structure to estimate pose in the NED reference frame. The state is defined as:

$$x = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ position_N \\ position_E \\ position_D \\ v_N \\ v_E \\ v_D \\ gyrobias_X \\ gyrobias_Y \\ gyrobias_Z \\ accelbias_X \\ accelbias_Y \\ accelbias_Z \\ scaleFactor \end{bmatrix}$$

where

- $q_0, q_1, q_2, q_3$  -- Parts of orientation quaternion. The orientation quaternion represents a frame rotation from the platform's current orientation to the local NED coordinate system.
- $position_N, position_E, position_D$  -- Position of the platform in the local NED coordinate system.
- $gyrobias_X, gyrobias_Y, gyrobias_Z$  -- Bias in the gyroscope reading.
- $accelbias_X, accelbias_Y, accelbias_Z$  -- Bias in the accelerometer reading.
- $scaleFactor$  -- Scale factor of the pose estimate.

Given the conventional formulation of the state transition function,

$$x_{k|k-1} = f(\hat{x}_{k-1|k-1})$$

the predicted state estimate is:

$$x_{k|k-1} =$$

$$\begin{aligned}
& \left[ \begin{array}{l}
q_0 + \Delta t * q_1(gyrobias_X/2 - gyro_X/2) + \Delta t * q_2(gyrobias_Y/2 - gyro_Y/2) + \Delta t * q_3(gyrobias_Z/2 - gyro_Z/2) \\
q_1 - \Delta t * q_0(gyrobias_X/2 - gyro_X/2) + \Delta t * q_3(gyrobias_Y/2 - gyro_Y/2) - \Delta t * q_2(gyrobias_Z/2 - gyro_Z/2) \\
q_2 - \Delta t * q_3(gyrobias_X/2 - gyro_X/2) - \Delta t * q_0(gyrobias_Y/2 - gyro_Y/2) + \Delta t * q_1(gyrobias_Z/2 - gyro_Z/2) \\
q_3 + \Delta t * q_2(gyrobias_X/2 - gyro_X/2) - \Delta t * q_1(gyrobias_Y/2 - gyro_Y/2) - \Delta t * q_0(gyrobias_Z/2 - gyro_Z/2)
\end{array} \right] \\
& \quad \left[ \begin{array}{l}
position_N + \Delta t * v_N \\
position_E + \Delta t * v_E \\
position_D + \Delta t * v_D
\end{array} \right] \\
& v_N - \Delta t * \left[ \begin{array}{l}
q_0 * (q_0 * (accelbias_X - accel_X) - q_3 * (accelbias_Y - accel_Y) + q_2 * (accelbias_Z - accel_Z)) + g_N + \\
q_2 * (q_1 * (accelbias_Y - accel_Y) - q_2 * (accelbias_X - accel_X) + q_0 * (accelbias_Z - accel_Z)) + \\
q_1 * (q_1 * (accelbias_X - accel_X) + q_2 * (accelbias_Y - accel_Y) + q_3 * (accelbias_Z - accel_Z)) - \\
q_3 * (q_3 * (accelbias_X - accel_X) + q_0 * (accelbias_Y - accel_Y) - q_1 * (accelbias_Z - accel_Z))
\end{array} \right] \\
& v_E - \Delta t * \left[ \begin{array}{l}
q_0 * (q_3 * (accelbias_X - accel_X) + q_0 * (accelbias_Y - accel_Y) - q_1 * (accelbias_Z - accel_Z)) + g_E - \\
q_1 * (q_1 * (accelbias_Y - accel_Y) - q_2 * (accelbias_X - accel_X) + q_0 * (accelbias_Z - accel_Z)) + \\
q_2 * (q_1 * (accelbias_X - accel_X) + q_2 * (accelbias_Y - accel_Y) + q_3 * (accelbias_Z - accel_Z)) + \\
q_3 * (q_0 * (accelbias_X - accel_X) - q_3 * (accelbias_Y - accel_Y) + q_2 * (accelbias_Z - accel_Z))
\end{array} \right. \\
& \quad \left. \begin{array}{l}
q_0 * (q_1 * (accelbias_Y - accel_Y) - q_2 * (accelbias_X - accel_X) + q_0 * (accelbias_Z - accel_Z)) + g_D + \\
q_1 * (q_2 * (accelbias_Y - accel_Y) + q_3 * (accelbias_X - accel_X) - q_1 * (accelbias_Z - accel_Z))
\end{array} \right]
\end{aligned}$$

2-59

where

- $\Delta t$  -- IMU sample time.
- $g_N$ ,  $g_E$ ,  $g_D$  -- Constant gravity vector in the NED frame.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[insfilterAsync](#) | [insfilterMARG](#) | [insfilterNonholonomic](#)

**Introduced in R2019a**

# copy

Create copy of insfilterAsync

## Syntax

```
newFilter = copy(filter)
```

## Description

`newFilter = copy(filter)` returns a copy of the `insfilterAsync`, `filter`, that has exactly the same property values.

## Input Arguments

### **filter — Filter to be copied**

`insfilterAsync`

Filter to be copied, specified as an `insfilterAsync` object.

## Output Arguments

### **newFilter — New copied filter**

`insfilterAsync`

New copied filter, returned as an `insfilterAsync` object.

## See Also

`insfilterAsync`

## Introduced in R2020b

# stateinfo

Display state vector information for `insfilterAsync`

## Syntax

```
stateinfo(FUSE)
info = stateinfo(FUSE)
```

## Description

`stateinfo(FUSE)` displays the description of each index of the `State` property of the `insfilterAsync` object and the associated units.

`info = stateinfo(FUSE)` returns a structure with fields containing descriptions of the elements of the state vector of the filter, `FUSE`.

## Examples

### State information of `insfilterAsync`

Create an `insfilterAsync` object.

```
filter = insfilterAsync;
```

Display the state information of the created filter.

```
stateinfo(filter)

States                      Units    Index
Orientation (quaternion parts) 1:4
Angular Velocity (XYZ)        rad/s    5:7
Position (NAV)                m        8:10
Velocity (NAV)                m/s      11:13
Acceleration (NAV)            m/s^2    14:16
Accelerometer Bias (XYZ)     m/s^2    17:19
Gyroscope Bias (XYZ)          rad/s    20:22
Geomagnetic Field Vector (NAV) μT       23:25
Magnetometer Bias (XYZ)       μT       26:28
```

Output the state information of the filter as a structure.

```
info = stateinfo(filter)

info = struct with fields:
    Orientation: [1 2 3 4]
    AngularVelocity: [5 6 7]
        Position: [8 9 10]
        Velocity: [11 12 13]
    Acceleration: [14 15 16]
    AccelerometerBias: [17 18 19]
    GyroscopeBias: [20 21 22]
```

```
GeomagneticFieldVector: [23 24 25]
MagnetometerBias: [26 27 28]
```

## Input Arguments

**FUSE — insfilterAsync object**  
object

insfilterAsync, specified as an object.

## Output Arguments

**info — State information**  
structure

State information, returned as a structure with fields containing descriptions of the elements of the state vector of the filter. The values of each field are the corresponding indices of the state vector.

## Extended Capabilities

**C/C++ Code Generation**  
Generate C and C++ code using MATLAB® Coder™.

## See Also

[insfilter](#) | [insfilterAsync](#)

**Introduced in R2019a**

## reset

Reset internal states for `insfilterAsync`

### Syntax

```
reset(FUSE)
```

### Description

`reset(FUSE)` resets the `State` and `StateCovariance` properties of the `insfilterAsync` object to their default values.

### Input Arguments

**FUSE — `insfilterAsync` object**  
object

`insfilterAsync`, specified as an object.

### Extended Capabilities

**C/C++ Code Generation**  
Generate C and C++ code using MATLAB® Coder™.

### See Also

`insfilter` | `insfilterAsync`

**Introduced in R2019a**

# tune

Tune `insfilterAsync` parameters to reduce estimation error

## Syntax

```
tunedMeasureNoise = tune(filter,measureNoise,sensorData,groundTruth)
tunedMeasureNoise = tune(____,config)
```

## Description

`tunedMeasureNoise = tune(filter,measureNoise,sensorData,groundTruth)` adjusts the properties of the `insfilterAsync` filter object, `filter`, and measurement noises to reduce the root-mean-squared (RMS) state estimation error between the fused sensor data and the ground truth. The function also returns the tuned measurement noise, `tunedMeasureNoise`. The function uses the property values in the filter and the measurement noise provided in the `measureNoise` structure as the initial estimate for the optimization algorithm.

`tunedMeasureNoise = tune(____,config)` specifies the tuning configuration based on a `tunerconfig` object, `config`.

## Examples

### Tune `insfilterAsync` to Optimize Pose Estimate

Load the recorded sensor data and ground truth data.

```
load('insfilterAsyncTuneData.mat');
```

Create timetables for the sensor data and the truth data.

```
sensorData = timetable(Accelerometer, Gyroscope, ...
    Magnetometer, GPSPosition, GPSVelocity, 'SampleRate', 100);
groundTruth = timetable(Orientation, Position, ...
    'SampleRate', 100);
```

Create an `insfilterAsync` filter object that has a few noise properties.

```
filter = insfilterAsync('State', initialState, ...
    'StateCovariance', initialStateCovariance, ...
    'AccelerometerBiasNoise', 1e-7, ...
    'GyroscopeBiasNoise', 1e-7, ...
    'MagnetometerBiasNoise', 1e-7, ...
    'GeomagneticVectorNoise', 1e-7);
```

Create a tuner configuration object for the filter. Set the maximum iterations to two. Also, set the tunable parameters as the unspecified properties.

```
config = tunerconfig('insfilterAsync','MaxIterations',8);
config.TunableParameters = setdiff(config.TunableParameters, ...
    {'GeomagneticVectorNoise', 'AccelerometerBiasNoise', ...}
```

```
'GyroscopeBiasNoise', 'MagnetometerBiasNoise'}));
config.TunableParameters

ans = 1×10 string
    "AccelerationNoise"    "AccelerometerNoise"    "AngularVelocityNoise"    "GPSPositionNoise"
```

Use the tuner noise function to obtain a set of initial sensor noises used in the filter.

```
measNoise = tunernoise('insfilterAsync')

measNoise = struct with fields:
    AccelerometerNoise: 1
        GyroscopeNoise: 1
    MagnetometerNoise: 1
        GPSPositionNoise: 1
        GPSVelocityNoise: 1
```

Tune the filter and obtain the tuned parameters.

```
tunedParams = tune(filter,measNoise,sensorData,groundTruth,config);
```

Iteration	Parameter	Metric
1	AccelerationNoise	2.1345
1	AccelerometerNoise	2.1264
1	AngularVelocityNoise	1.9659
1	GPSPositionNoise	1.9341
1	GPSVelocityNoise	1.8420
1	GyroscopeNoise	1.7589
1	MagnetometerNoise	1.7362
1	PositionNoise	1.7362
1	QuaternionNoise	1.7218
1	VelocityNoise	1.7218
2	AccelerationNoise	1.7190
2	AccelerometerNoise	1.7170
2	AngularVelocityNoise	1.6045
2	GPSPositionNoise	1.5948
2	GPSVelocityNoise	1.5323
2	GyroscopeNoise	1.4803
2	MagnetometerNoise	1.4703
2	PositionNoise	1.4703
2	QuaternionNoise	1.4632
2	VelocityNoise	1.4632
3	AccelerationNoise	1.4596
3	AccelerometerNoise	1.4548
3	AngularVelocityNoise	1.3923
3	GPSPositionNoise	1.3810
3	GPSVelocityNoise	1.3322
3	GyroscopeNoise	1.2998
3	MagnetometerNoise	1.2976
3	PositionNoise	1.2976
3	QuaternionNoise	1.2943
3	VelocityNoise	1.2943
4	AccelerationNoise	1.2906
4	AccelerometerNoise	1.2836
4	AngularVelocityNoise	1.2491
4	GPSPositionNoise	1.2258

4	GPSVelocityNoise	1.1880
4	GyroscopeNoise	1.1701
4	MagnetometerNoise	1.1698
4	PositionNoise	1.1698
4	QuaternionNoise	1.1688
4	VelocityNoise	1.1688
5	AccelerationNoise	1.1650
5	AccelerometerNoise	1.1569
5	AngularVelocityNoise	1.1454
5	GPSPositionNoise	1.1100
5	GPSVelocityNoise	1.0778
5	GyroscopeNoise	1.0709
5	MagnetometerNoise	1.0675
5	PositionNoise	1.0675
5	QuaternionNoise	1.0669
5	VelocityNoise	1.0669
6	AccelerationNoise	1.0634
6	AccelerometerNoise	1.0549
6	AngularVelocityNoise	1.0549
6	GPSPositionNoise	1.0180
6	GPSVelocityNoise	0.9866
6	GyroscopeNoise	0.9810
6	MagnetometerNoise	0.9775
6	PositionNoise	0.9775
6	QuaternionNoise	0.9768
6	VelocityNoise	0.9768
7	AccelerationNoise	0.9735
7	AccelerometerNoise	0.9652
7	AngularVelocityNoise	0.9652
7	GPSPositionNoise	0.9283
7	GPSVelocityNoise	0.8997
7	GyroscopeNoise	0.8947
7	MagnetometerNoise	0.8920
7	PositionNoise	0.8920
7	QuaternionNoise	0.8912
7	VelocityNoise	0.8912
8	AccelerationNoise	0.8885
8	AccelerometerNoise	0.8811
8	AngularVelocityNoise	0.8807
8	GPSPositionNoise	0.8479
8	GPSVelocityNoise	0.8238
8	GyroscopeNoise	0.8165
8	MagnetometerNoise	0.8165
8	PositionNoise	0.8165
8	QuaternionNoise	0.8159
8	VelocityNoise	0.8159

Fuse the sensor data using the tuned filter.

```
dt = seconds(diff(groundTruth.Time));
N = size(sensorData,1);
qEst = quaternion.zeros(N,1);
posEst = zeros(N,3);
% Iterate the filter for prediction and correction using sensor data.
for ii=1:N
    if ii ~= 1
        predict(filter, dt(ii-1));
    end
end
```

```
if all(~isnan(Accelerometer(ii,:)))
    fuseaccel(filter, Accelerometer(ii,:), ...
        tunedParams.AccelerometerNoise);
end
if all(~isnan(Gyroscope(ii,:)))
    fusegyro(filter, Gyroscope(ii,:), ...
        tunedParams.GyroscopeNoise);
end
if all(~isnan(Magnetometer(ii,1)))
    fusemag(filter, Magnetometer(ii,:), ...
        tunedParams.MagnetometerNoise);
end
if all(~isnan(GPSPosition(ii,1)))
    fusegps(filter, GPSPosition(ii,:), ...
        tunedParams.GPSPositionNoise, GPSVelocity(ii,:), ...
        tunedParams.GPSVelocityNoise);
end
[posEst(ii,:), qEst(ii,:)] = pose(filter);
end
```

Compute the RMS errors.

```
orientationError = rad2deg(dist(qEst, Orientation));
rmsorientationError = sqrt(mean(orientationError.^2))

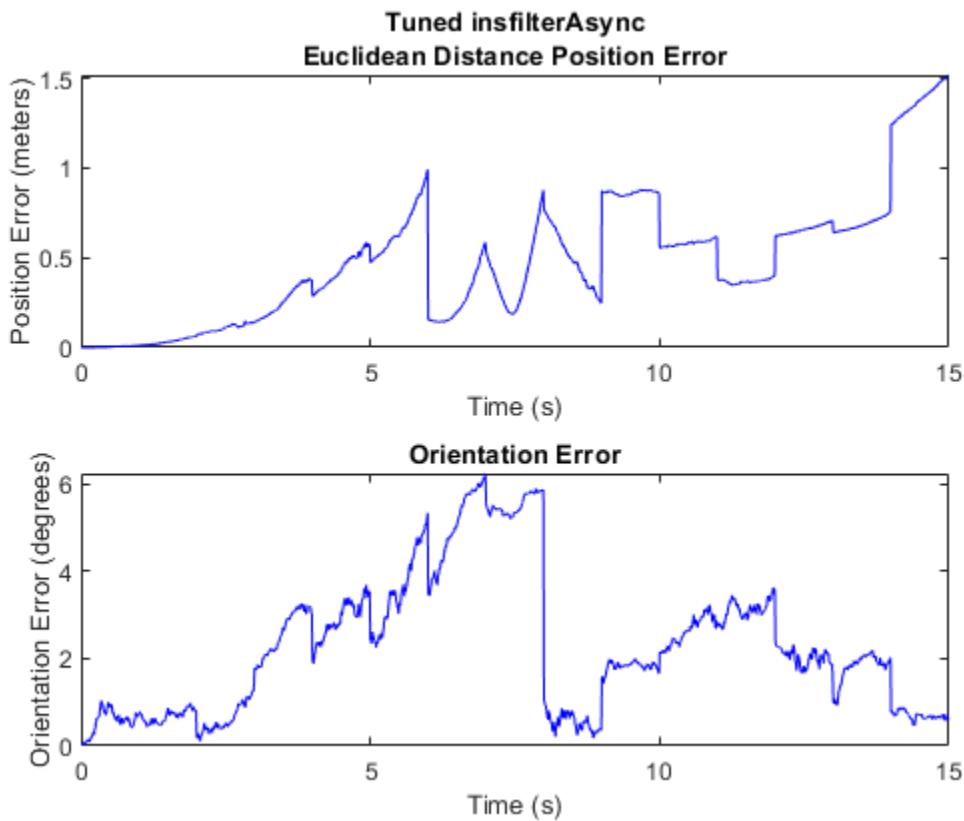
rmsorientationError = 2.7801

positionError = sqrt(sum((posEst - Position).^2, 2));
rmspositionError = sqrt(mean(positionError.^2))

rmspositionError = 0.5966
```

Visualize the results.

```
figure();
t = (0:N-1)./ groundTruth.Properties.SampleRate;
subplot(2,1,1)
plot(t, positionError, 'b');
title("Tuned insfilterAsync" + newline + "Euclidean Distance Position Error")
xlabel('Time (s)');
ylabel('Position Error (meters)')
subplot(2,1,2)
plot(t, orientationError, 'b');
title("Orientation Error")
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



## Input Arguments

**filter — Filter object**  
infiltterAsync object

Filter object, specified as an `insfilterAsync` object.

**measureNoise — Measurement noise**  
structure

Measurement noise, specified as a structure. The function uses the measurement noise input as the initial guess for tuning the measurement noise. The structure must contain these fields:

Field name	Description
AccelerometerNoise	Variance of accelerometer noise, specified as a scalar in $(\text{m}^2/\text{s})$
GyroscopeNoise	Variance of gyroscope noise, specified as a scalar in $(\text{rad}/\text{s})^2$
MagnetometerNoise	Variance of magnetometer noise, specified as a scalar in $(\mu\text{T})^2$
GPSPositionNoise	Variance of GPS position noise, specified as a scalar in $\text{m}^2$

Field name	Description
GPSVelocityNoise	Variance of GPS velocity noise, specified as a scalar in $(\text{m}/\text{s})^2$

**sensorData — Sensor data**

duration

Sensor data, specified as a `timetable`. In each row, the time and sensor data is specified as:

- **Time** — Time at which the data is obtained, specified as a scalar in seconds.
- **Accelerometer** — Accelerometer data, specified as a 1-by-3 vector of scalars in  $\text{m}^2/\text{s}$ .
- **Gyroscope** — Gyroscope data, specified as a 1-by-3 vector of scalars in  $\text{rad}/\text{s}$ .
- **Magnetometer** — Magnetometer data, specified as a 1-by-3 vector of scalars in  $\mu\text{T}$ .
- **GPSPosition** — GPS position data, specified as a 1-by-3 vector of scalars in meters.
- **GPSVelocity** — GPS velocity data, specified as a 1-by-3 vector of scalars in  $\text{m}/\text{s}$ .

If a sensor does not produce measurements, specify the corresponding entry as `NaN`. If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `sensorData` input based on your choice.

**groundTruth — Ground truth data**

duration

Ground truth data, specified as a `timetable`. In each row, the table can optionally contain any of these variables:

- **Orientation** — Orientation from the navigation frame to the body frame, specified as a quaternion or a 3-by-3 rotation matrix.
- **AngularVelocity** — Angular velocity in body frame, specified as a 1-by-3 vector of scalars in  $\text{rad}/\text{s}$ .
- **Position** — Position in navigation frame, specified as a 1-by-3 vector of scalars in meters.
- **Velocity** — Velocity in navigation frame, specified as a 1-by-3 vector of scalars in  $\text{m}/\text{s}$ .
- **Acceleration** — Acceleration in navigation frame, specified as a 1-by-3 vector of scalars in  $\text{m}^2/\text{s}$ .
- **AccelerometerBias** — Accelerometer delta angle bias in body frame, specified as a 1-by-3 vector of scalars in  $\text{m}^2/\text{s}$ .
- **GyroscopeBias** — Gyroscope delta angle bias in body frame, specified as a 1-by-3 vector of scalars in  $\text{rad}/\text{s}$ .
- **GeomagneticFieldVector** — Geomagnetic field vector in navigation frame, specified as a 1-by-3 vector of scalars.
- **MagnetometerBias** — Magnetometer bias in body frame, specified as a 1-by-3 vector of scalars in  $\mu\text{T}$ .

The function processes each row of the `sensorData` and `groundTruth` tables sequentially to calculate the state estimate and RMS error from the ground truth. State variables not present in `groundTruth` input are ignored for the comparison. The `sensorData` and the `groundTruth` tables must have the same time steps.

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `groundTruth` input based on your choice.

**config — Tuner configuration**

tunerconfig object

Tuner configuration, specified as a `tunerconfig` object.**Output Arguments****tunedMeasureNoise — Tuned measurement noise**

structure

Tuned measurement noise, returned as a structure. The structure contains these fields.

Field name	Description
AccelerometerNoise	Variance of accelerometer noise, specified as a scalar in $(m^2/s)^2$
GyroscopeNoise	Variance of gyroscope noise, specified as a scalar in $(rad/s)^2$
MagnetometerNoise	Variance of magnetometer noise, specified as a scalar in $(\mu T)^2$
GPSPositionNoise	Variance of GPS position noise, specified as a scalar in $m^2$
GPSVelocityNoise	Variance of GPS velocity noise, specified as a scalar in $(m/s)^2$

**References**

- [1] Abbeel, P., Coates, A., Montemerlo, M., Ng, A.Y. and Thrun, S. Discriminative Training of Kalman Filters. In *Robotics: Science and systems*, Vol. 2, pp. 1, 2005.

**See Also****Introduced in R2020b**

# **predict**

Update states based on motion model for `insfilterAsync`

## **Syntax**

```
predict(FUSE,dt)
```

## **Description**

`predict(FUSE,dt)` updates states based on the motion model.

## **Input Arguments**

### **FUSE — `insfilterAsync` object**

object

`insfilterAsync`, specified as an object.

### **dt — Delta time to propagate forward (s)**

scalar

Delta time to propagate forward in seconds, specified as a positive scalar.

Data Types: `single` | `double`

## **Extended Capabilities**

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

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

## **See Also**

`insfilter` | `insfilterAsync`

## **Introduced in R2019a**

# pose

Current position, orientation, and velocity estimate for `insfilterAsync`

## Syntax

```
[position,orientation,velocity] = pose(FUSE)
[position,orientation,velocity] = pose(FUSE,format)
```

## Description

`[position,orientation,velocity] = pose(FUSE)` returns the current estimate of the pose.

`[position,orientation,velocity] = pose(FUSE,format)` returns the current estimate of the pose with orientation in the specified orientation format.

## Input Arguments

**FUSE — insfilterAsync object**  
object

`insfilterAsync`, specified as an object.

**format — Output orientation format**  
`'quaternion'` (default) | `'rotmat'`

Output orientation format, specified as either `'quaternion'` for a quaternion or `'rotmat'` for a rotation matrix.

Data Types: `char` | `string`

## Output Arguments

**position — Position estimate in the local NED coordinate system (m)**  
3-element row vector

Position estimate in the local NED coordinate system in meters, returned as a 3-element row vector.

Data Types: `single` | `double`

**orientation — Orientation estimate in the local NED coordinate system**  
`quaternion` (default) | 3-by-3 rotation matrix

Orientation estimate in the local NED coordinate system, returned as a scalar quaternion or 3-by-3 rotation matrix, depending on the specified orientation `format`. The quaternion or rotation matrix represents a frame rotation from the local NED reference frame to the body reference frame.

Data Types: `single` | `double` | `quaternion`

**velocity — Velocity estimate in the local NED coordinate system (m/s)**  
3-element row vector

Velocity estimate in the local NED coordinate system in m/s, returned as a 3-element row vector.

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`insfilter` | `insfilterAsync`

**Introduced in R2019a**

# fusemag

Correct states using magnetometer data for `insfilterAsync`

## Syntax

```
[res, resCov] = fusemag(FUSE, magReadings, magReadingsCovariance)
```

## Description

`[res, resCov] = fusemag(FUSE, magReadings, magReadingsCovariance)` fuses magnetometer data to correct the state estimate.

## Input Arguments

**FUSE — insfilterAsync object**  
object

`insfilterAsync`, specified as an object.

**magReadings — Magnetometer readings ( $\mu\text{T}$ )**  
3-element row vector

Magnetometer readings in  $\mu\text{T}$ , specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

**magReadingsCovariance — Magnetometer readings error covariance ( $\mu\text{T}^2$ )**  
scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in  $\mu\text{T}^2$ , specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

**res — Residual**  
1-by-3 vector of real values

Residual, returned a 1-by-3 vector of real values in  $\mu\text{T}$ .

Data Types: `single` | `double`

**resCov — Residual covariance**  
3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in  $(\mu\text{T})^2$ .

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[insfilter](#) | [insfilterAsync](#)

**Introduced in R2019a**

# residualmag

Residuals and residual covariance from magnetometer measurements for `insfilterAsync`

## Syntax

```
[res, resCov] = residualmag(FUSE, magReadings, magReadingsCovariance)
```

## Description

`[res, resCov] = residualmag(FUSE, magReadings, magReadingsCovariance)` computes the residual, residual, and the residual covariance, `resCov`, based on the magnetometer readings and the corresponding covariance.

## Input Arguments

**FUSE — insfilterAsync**  
ahrs10filter | object

`insfilterAsync`, specified as an object.

**magReadings — Magnetometer readings ( $\mu\text{T}$ )**  
3-element row vector

Magnetometer readings in  $\mu\text{T}$ , specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

**magReadingsCovariance — Magnetometer readings error covariance ( $\mu\text{T}^2$ )**  
scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in  $\mu\text{T}^2$ , specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

**res — Residual**  
1-by-3 vector of real values

Residual, returned as a 1-by-3 vector of real values in  $\mu\text{T}$ .

Data Types: `single` | `double`

**resCov — Residual covariance**  
3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in  $(\mu\text{T})^2$ .

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`insfilterAsync`

Introduced in R2020a

# fusegyro

Correct states using gyroscope data for `insfilterAsync`

## Syntax

```
[res, resCov] = fusegyro(FUSE, gyroReadings, gyroCovariance)
```

## Description

`[res, resCov] = fusegyro(FUSE, gyroReadings, gyroCovariance)` fuses gyroscope data to correct the state estimate.

## Input Arguments

**FUSE — insfilterAsync object**  
object

`insfilterAsync`, specified as an object.

**gyroReadings — Gyroscope readings in local sensor body coordinate system (rad/s)**  
3-element row vector

Gyroscope readings in local sensor body coordinate system in rad/s, specified as a 3-element row vector.

Data Types: `single` | `double`

**gyroCovariance — Covariance of gyroscope measurement error ((rad/s)<sup>2</sup>)**  
scalar | 3-element row vector | 3-by-3 matrix

Covariance of gyroscope measurement error in (rad/s)<sup>2</sup>, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

**res — Residual**  
1-by-3 vector of real values

Residual, returned as a 1-by-3 vector of real values in rad/s.

**resCov — Residual covariance**  
3-by-3 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values in (rad/s)<sup>2</sup>.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[insfilter](#) | [insfilterAsync](#)

**Introduced in R2019a**

# residualgyro

Residuals and residual covariance from gyroscope measurements for `insfilterAsync`

## Syntax

```
[res, resCov] = residualgyro(FUSE, gyroReadings, gyroCovariance)
```

## Description

`[res, resCov] = residualgyro(FUSE, gyroReadings, gyroCovariance)` computes the residual, `res`, and the innovation covariance, `resCov`, based on the gyroscope readings and the corresponding covariance.

## Input Arguments

**FUSE — `insfilterAsync`**  
`ahrs10filter` | object

`insfilterAsync`, specified as an object.

**gyroReadings — Gyroscope readings in local sensor body coordinate system (rad/s)**  
3-element row vector

Gyroscope readings in local sensor body coordinate system in rad/s, specified as a 3-element row vector.

Data Types: `single` | `double`

**gyroCovariance — Covariance of gyroscope measurement error ((rad/s)<sup>2</sup>)**  
scalar | 3-element row vector | 3-by-3 matrix

Covariance of gyroscope measurement error in (rad/s)<sup>2</sup>, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

**res — Residual**  
1-by-3 vector of real values

Residual, returned as a 1-by-3 vector of real values in rad/s.

**resCov — Residual covariance**  
3-by-3 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values in (rad/s)<sup>2</sup>.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[insfilter](#) | [insfilterAsync](#)

**Introduced in R2020a**

# fusegps

Correct states using GPS data for `insfilterAsync`

## Syntax

```
[res,resCov] = fusegps(FUSE,position,positionCovariance)
[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

## Description

`[res,resCov] = fusegps(FUSE,position,positionCovariance)` fuses GPS position data to correct the state estimate.

`[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)` fuses GPS position and velocity data to correct the state estimate.

## Input Arguments

**FUSE — insfilterAsync object**  
object

`insfilterAsync`, specified as an object.

**position — Position of GPS receiver (LLA)**  
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

**positionCovariance — Position measurement covariance of GPS receiver ( $\text{m}^2$ )**  
3-by-3 matrix

Position measurement covariance of GPS receiver in  $\text{m}^2$ , specified as a 3-by-3 matrix.

Data Types: `single` | `double`

**velocity — Velocity of GPS receiver in local NED coordinate system (m/s)**  
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

**velocityCovariance — Velocity measurement covariance of GPS receiver ( $\text{m}/\text{s}^2$ )**  
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s<sup>2</sup>, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **res — Position and velocity residual**

1-by-6 vector of real values

Position and velocity residual, returned as a 1-by-6 vector of real values in m and m/s, respectively.

### **resCov — Residual covariance**

6-by-6 matrix of real values

Residual covariance, returned as a 6-by-6 matrix of real values.

## Extended Capabilities

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

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

## See Also

`insfilter` | `insfilterAsync` | `insfilterMARG`

## Introduced in R2019a

# fuseaccel

Correct states using accelerometer data for `insfilterAsync`

## Syntax

```
[res,resCov] = fuseaccel(FUSE,acceleration,accelerationCovariance)
```

## Description

`[res,resCov] = fuseaccel(FUSE,acceleration,accelerationCovariance)` fuses accelerometer data to correct the state estimate.

## Input Arguments

**FUSE — insfilterAsync object**  
object

`insfilterAsync`, specified as an object.

**acceleration — Accelerometer readings in local sensor body coordinate system (m/s<sup>2</sup>)**  
3-element row vector

Accelerometer readings in local sensor body coordinate system in m/s<sup>2</sup>, specified as a 3-element row vector

Data Types: `single` | `double`

**accelerationCovariance — Acceleration error covariance of accelerometer measurement ((m/s<sup>2</sup>)<sup>2</sup>)**  
scalar | 3-element row vector | 3-by-3 matrix

Acceleration error covariance of the accelerometer measurement in (m/s<sup>2</sup>)<sup>2</sup>, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

**res — Residual**  
1-by-3 vector of real values

Residual, returned a 1-by-3 vector of real values in m/s<sup>2</sup>.

**resCov — Residual covariance**  
3-by-3 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values in (m/s<sup>2</sup>)<sup>2</sup>.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[insfilter](#) | [insfilterAsync](#)

**Introduced in R2019a**

# residualaccel

Residuals and residual covariance from accelerometer measurements for `insfilterAsync`

## Syntax

```
[res,resCov] = residualaccel(FUSE,acceleration,accelerationCovariance)
```

## Description

`[res,resCov] = residualaccel(FUSE,acceleration,accelerationCovariance)`  
computes the residual, `res`, and the residual covariance, `resCov`, based on the acceleration readings and the corresponding covariance.

## Input Arguments

**FUSE — `insfilterAsync`**  
`ahrs10filter` | object

`insfilterAsync`, specified as an object.

**acceleration — Accelerometer readings in local sensor body coordinate system (m/s<sup>2</sup>)**  
3-element row vector

Accelerometer readings in local sensor body coordinate system in m/s<sup>2</sup>, specified as a 3-element row vector

Data Types: `single` | `double`

**accelerationCovariance — Acceleration error covariance of accelerometer measurement ((m/s<sup>2</sup>)<sup>2</sup>)**  
scalar | 3-element row vector | 3-by-3 matrix

Acceleration error covariance of the accelerometer measurement in (m/s<sup>2</sup>)<sup>2</sup>, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

**res — Residual**  
1-by-3 vector of real values

Residual, returned a 1-by-3 vector of real values in m/s<sup>2</sup>.

**resCov — Residual covariance**  
3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in (m/s<sup>2</sup>)<sup>2</sup>.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[insfilter](#) | [insfilterAsync](#)

**Introduced in R2020a**

# correct

Correct states using direct state measurements for `insfilterAsync`

## Syntax

```
correct(FUSE,idx,measurement,measurementCovariance)
```

## Description

`correct(FUSE,idx,measurement,measurementCovariance)` corrects the state and state estimation error covariance based on the measurement and measurement covariance. The measurement maps directly to the state specified by the indices `idx`.

## Input Arguments

**FUSE — `insfilterAsync` object**  
object

`insfilterAsync`, specified as an object.

**idx — State vector index of measurement to correct**  
*N*-element vector of increasing integers in the range [1, 28]

State vector index of measurement to correct, specified as an *N*-element vector of increasing integers in the range [1, 28].

The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Angular Velocity (XYZ)	rad/s	5:7
Position (NED)	m	8:10
Velocity (NED)	m/s	11:13
Acceleration (NED)	m/s <sup>2</sup>	14:16
Accelerometer Bias (XYZ)	m/s <sup>2</sup>	17:19
Gyroscope Bias (XYZ)	rad/s	20:22
Geomagnetic Field Vector (NED)	µT	23:25
Magnetometer Bias (XYZ)	µT	26:28

Data Types: `single` | `double`

**measurement — Direct measurement of state**  
*N*-element vector

Direct measurement of state, specified as an *N*-element vector. *N* is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

**measurementCovariance — Covariance of measurement**

scalar |  $N$ -element vector |  $N$ -by- $N$  matrix

Covariance of measurement, specified as a scalar,  $N$ -element vector, or  $N$ -by- $N$  matrix.  $N$  is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`insfilter` | `insfilterAsync`

**Introduced in R2019a**

# residual

Residuals and residual covariances from direct state measurements for `insfilterAsync`

## Syntax

```
[res, resCov] = residual(FUSE, idx, measurement, measurementCovariance)
```

## Description

`[res, resCov] = residual(FUSE, idx, measurement, measurementCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the direct state measurement and measurement covariance. The `measurement` maps directly to the states specified by indices, `idx`.

## Input Arguments

**FUSE — `insfilterAsync`**  
`ahrs10filter` | object

`insfilterAsync`, specified as an object.

**idx — State vector index of measurement to correct**

`N`-element vector of increasing integers in the range [1, 28]

State vector index of measurement to correct, specified as an `N`-element vector of increasing integers in the range [1, 28].

The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Angular Velocity (XYZ)	rad/s	5:7
Position (NED)	m	8:10
Velocity (NED)	m/s	11:13
Acceleration (NED)	m/s <sup>2</sup>	14:16
Accelerometer Bias (XYZ)	m/s <sup>2</sup>	17:19
Gyroscope Bias (XYZ)	rad/s	20:22
Geomagnetic Field Vector (NED)	µT	23:25
Magnetometer Bias (XYZ)	µT	26:28

Data Types: `single` | `double`

**measurement — Direct measurement of state**

`N`-element vector

Direct measurement of state, specified as a `N`-element vector.  $N$  is the number of elements of the index argument, `idx`.

**measurementCovariance — Covariance of measurement**

*N*-by-*N* matrix

Covariance of measurement, specified as an *N*-by-*N* matrix. *N* is the number of elements of the index argument, `idx`.

## Output Arguments

**res — Measurement residual**

1-by-*N* vector of real values

Measurement residual, returned as a 1-by-*N* vector of real values.

**resCov — Residual covariance**

*N*-by-*N* matrix of real values

Residual covariance, returned as a *N*-by-*N* matrix of real values.

## Extended Capabilities

**C/C++ Code Generation**

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

**See Also**

`insfilterAsync`

**Introduced in R2020a**

# residualgps

Residuals and residual covariance from GPS measurements for `insfilterAsync`

## Syntax

```
[res,resCov] = residualgps(FUSE,position,positionCovariance)
[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

## Description

`[res,resCov] = residualgps(FUSE,position,positionCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

`[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,
velocityCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

## Input Arguments

**FUSE — `insfilterAsync`**  
`ahrs10filter` | object

`insfilterAsync`, specified as an object.

**position — Position of GPS receiver (LLA)**  
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

**positionCovariance — Position measurement covariance of GPS receiver ( $\text{m}^2$ )**  
3-by-3 matrix

Position measurement covariance of GPS receiver in  $\text{m}^2$ , specified as a 3-by-3 matrix.

Data Types: `single` | `double`

**velocity — Velocity of GPS receiver in local NED coordinate system (m/s)**  
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

**velocityCovariance — Velocity measurement covariance of GPS receiver ( $\text{m}/\text{s}^2$ )**  
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s<sup>2</sup>, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **res — Position and velocity residual**

1-by-3 vector of real values | 1-by-6 vector of real values

Position and velocity residual, returned as a 1-by-3 vector of real values if the inputs only contain position information, and returned as 1-by-6 vector of real values if the inputs also contain velocity information.

### **resCov — Residual covariance**

3-by-3 matrix of real values | 6-by-6 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values if the inputs only contain position information, and a 6-by-6 matrix of real values if the inputs also contain velocity information.

## Extended Capabilities

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

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

### **See Also**

`insfilterAsync`

### **Introduced in R2020a**

# insfilterAsync

Estimate pose from asynchronous MARG and GPS data

## Description

The `insfilterAsync` object implements sensor fusion of MARG and GPS data to estimate pose in the NED (or ENU) reference frame. MARG (magnetic, angular rate, gravity) data is typically derived from magnetometer, gyroscope, and accelerometer data, respectively. The filter uses a 28-element state vector to track the orientation quaternion, velocity, position, MARG sensor biases, and geomagnetic vector. The `insfilterAsync` object uses a continuous-discrete extended Kalman filter to estimate these quantities.

## Creation

### Syntax

```
filter = insfilterAsync
filter = insfilterAsync('ReferenceFrame',RF)
filter = insfilterAsync(___,Name,Value)
```

### Description

`filter = insfilterAsync` creates an `insfilterAsync` object to fuse asynchronous MARG and GPS data with default property values.

`filter = insfilterAsync('ReferenceFrame',RF)` allows you to specify the reference frame, RF, of the filter. Specify RF as 'NED' (North-East-Down) or 'ENU' (East-North-Up). The default value is 'NED'.

`filter = insfilterAsync(___,Name,Value)` also allows you set properties of the created filter using one or more name-value pairs. Enclose each property name in single quotes.

## Properties

### ReferenceLocation — Reference location (deg, deg, meters)

[0 0 0] (default) | three-element positive row vector

Reference location, specified as a three-element row vector in geodetic coordinates (latitude, longitude, and altitude). Altitude is the height above the reference ellipsoid model, WGS84. The reference location units are [degrees degrees meters].

Data Types: `single` | `double`

### QuaternionNoise — Additive quaternion process noise variance

[1e-6 1e-6 1e-6 1e-6] (default) | scalar | four-element row vector

Additive quaternion process noise variance, specified as a scalar or four-element vector of quaternion parts.

Data Types: single | double

**AngularVelocityNoise — Additive angular velocity process noise in local navigation coordinate system ((rad/s)<sup>2</sup>)**

[0.005 0.005 0.005] (default) | scalar | three-element row vector

Additive angular velocity process noise in the local navigation coordinate system in (rad/s)<sup>2</sup>, specified as a scalar or three-element row vector of positive real finite numbers.

- If **AngularVelocityNoise** is a row vector, the elements correspond to the noise in the x, y, and z axes of the local navigation coordinate system, respectively.
- If **AngularVelocityNoise** is a scalar, the single element is applied to each axis.

Data Types: single | double

**PositionNoise — Additive position process noise variance in local navigation coordinate system (m<sup>2</sup>)**

[1e-6 1e-6 1e-6] (default) | scalar | three-element row vector

Additive position process noise in the local navigation coordinate system in m<sup>2</sup>, specified as a scalar or three-element row vector of positive real finite numbers.

- If **PositionNoise** is a row vector, the elements correspond to the noise in the x, y, and z axes of the local navigation coordinate system, respectively.
- If **PositionNoise** is a scalar, the single element is applied to each axis.

Data Types: single | double

**VelocityNoise — Additive velocity process noise variance in local navigation coordinate system ((m/s)<sup>2</sup>)**

[1e-6 1e-6 1e-6] (default) | scalar | three-element row vector

Additive velocity process noise in the local navigation coordinate system in (m/s)<sup>2</sup>, specified as a scalar or three-element row vector of positive real finite numbers.

- If **VelocityNoise** is a row vector, the elements correspond to the noise in the x, y, and z axes of the local navigation coordinate system, respectively.
- If **VelocityNoise** is a scalar, the single element is applied to each axis.

Data Types: single | double

**AccelerationNoise — Additive acceleration process noise variance in local navigation coordinate system ((m/s<sup>2</sup>)<sup>2</sup>)**

[50 50 50] (default) | scalar | three-element row vector

Additive acceleration process noise in (m/s<sup>2</sup>)<sup>2</sup>, specified as a scalar or three-element row vector of positive real finite numbers.

- If **AccelerationNoise** is a row vector, the elements correspond to the noise in the x, y, and z axes of the local navigation coordinate system, respectively.
- If **AccelerationNoise** is a scalar, the single element is applied to each axis.

Data Types: single | double

**GyroscopeBiasNoise — Additive process noise variance from gyroscope bias ((rad/s)<sup>2</sup>)**  
 $[1e-10 \ 1e-10 \ 1e-10]$  (default) | scalar | three-element row vector

Additive process noise variance from the gyroscope bias in (rad/s)<sup>2</sup>, specified as a scalar or three-element row vector of positive real finite numbers.

- If GyroscopeBiasNoise is a row vector, the elements correspond to the noise in the x, y, and z axes of the gyroscope, respectively.
- If GyroscopeBiasNoise is a scalar, the single element is applied to each axis.

Data Types: single | double

**AccelerometerBiasNoise — Additive process noise variance from accelerometer bias ((m/s<sup>2</sup>)<sup>2</sup>)**  
 $[1e-4 \ 1e-4 \ 1e-4]$  (default) | positive scalar | three-element row vector

Additive process noise variance from accelerometer bias in (m/s<sup>2</sup>)<sup>2</sup>, specified as a scalar or three-element row vector of positive real numbers.

- If AccelerometerBiasNoise is a row vector, the elements correspond to the noise in the x, y, and z axes of the accelerometer, respectively.
- If AccelerometerBiasNoise is a scalar, the single element is applied to each axis.

**GeomagneticVectorNoise — Additive process noise variance of geomagnetic vector in local navigation coordinate system ( $\mu\text{T}^2$ )**

$[1e-6 \ 1e-6 \ 1e-6]$  (default) | positive scalar | three-element row vector

Additive process noise variance of geomagnetic vector in  $\mu\text{T}^2$ , specified as a scalar or three-element row vector of positive real numbers.

- If GeomagneticVectorNoise is a row vector, the elements correspond to the noise in the x, y, and z axes of the local navigation coordinate system, respectively.
- If GeomagneticVectorNoise is a scalar, the single element is applied to each axis.

**MagnetometerBiasNoise — Additive process noise variance from magnetometer bias ( $\mu\text{T}^2$ )**  
 $[0.1 \ 0.1 \ 0.1]$  (default) | positive scalar | three-element row vector

Additive process noise variance from magnetometer bias in  $\mu\text{T}^2$ , specified as a scalar or three-element row vector of positive real numbers.

- If MagnetometerBiasNoise is a row vector, the elements correspond to the noise in the x, y, and z axes of the magnetometer, respectively.
- If MagnetometerBiasNoise is a scalar, the single element is applied to each axis.

**State — State vector of extended Kalman filter**

28-element column vector

State vector of the extended Kalman filter. The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Angular Velocity (XYZ)	rad/s	5:7

State	Units	Index
Position (NED or ENU)	m	8:10
Velocity (NED or ENU)	m/s	11:13
Acceleration (NED or ENU)	m/s <sup>2</sup>	14:16
Accelerometer Bias (XYZ)	m/s <sup>2</sup>	17:19
Gyroscope Bias (XYZ)	rad/s	20:22
Geomagnetic Field Vector (NED or ENU)	µT	23:25
Magnetometer Bias (XYZ)	µT	26:28

The default initial state corresponds to an object at rest located at [0 0 0] in geodetic LLA coordinates.

Data Types: single | double

### **StateCovariance — State error covariance for extended Kalman filter**

eye(28) (default) | 28-by-28 matrix

State error covariance for the extended Kalman filter, specified as a 28-by-28-element matrix of real numbers.

Data Types: single | double

## **Object Functions**

predict	Update states based on motion model for insfilterAsync
fuseaccel	Correct states using accelerometer data for insfilterAsync
fusegyro	Correct states using gyroscope data for insfilterAsync
fusemag	Correct states using magnetometer data for insfilterAsync
fusegps	Correct states using GPS data for insfilterAsync
correct	Correct states using direct state measurements for insfilterAsync
residual	Residuals and residual covariances from direct state measurements for insfilterAsync
residualaccel	Residuals and residual covariance from accelerometer measurements for insfilterAsync
residualgps	Residuals and residual covariance from GPS measurements for insfilterAsync
residualmag	Residuals and residual covariance from magnetometer measurements for insfilterAsync
residualgyro	Residuals and residual covariance from gyroscope measurements for insfilterAsync
pose	Current position, orientation, and velocity estimate for insfilterAsync
reset	Reset internal states for insfilterAsync
stateinfo	Display state vector information for insfilterAsync
copy	Create copy of insfilterAsync
tune	Tune insfilterAsync parameters to reduce estimation error
tunernoise	Noise structure of fusion filter

## **Examples**

### **Estimate Pose of UAV**

Load logged sensor data and ground truth pose.

```
load('uavshort.mat','refloc','initstate','imuFs', ...
    'accel','gyro','mag','lla','gpsvel', ...
    'trueOrient','truePos')
```

Create an INS filter to fuse asynchronous MARG and GPS data to estimate pose.

```
filt = insfilterAsync;
filt.ReferenceLocation = refloc;
filt.State = [initstate(1:4);0;0;0;initstate(5:10);0;0;0;initstate(11:end)];
```

Define sensor measurement noises. The noises were determined from datasheets and experimentation.

```
Rmag = 80;
Rvel = 0.0464;
Racc = 800;
Rgyro = 1e-4;
Rpos = 34;
```

Preallocate variables for position and orientation. Allocate a variable for indexing into the GPS data.

```
N = size(accel,1);
p = zeros(N,3);
q = zeros(N,1,'quaternion');

gpsIdx = 1;
```

Fuse accelerometer, gyroscope, magnetometer, and GPS data. The outer loop predicts the filter forward one time step and fuses accelerometer and gyroscope data at the IMU sample rate.

```
for ii = 1:N

    % Predict the filter forward one time step
    predict(filt,1./imuFs);

    % Fuse accelerometer and gyroscope readings
    fuseaccel(filt,accel(ii,:),Racc);
    fusegyro(filt,gyro(ii,:),Rgyro);

    % Fuse magnetometer at 1/2 the IMU rate
    if ~mod(ii, fix(imuFs/2))
        fusemag(filt,mag(ii,:),Rmag);
    end

    % Fuse GPS once per second
    if ~mod(ii,imuFs)
        fusegps(filt,lla(gpsIdx,:),Rpos,gpsvel(gpsIdx,:),Rvel);
        gpsIdx = gpsIdx + 1;
    end

    % Log the current pose estimate
    [p(ii,:),q(ii)] = pose(filt);

end
```

Calculate the RMS errors between the known true position and orientation and the output from the asynchronous IMU filter.

```
posErr = truePos - p;
qErr = rad2deg(dist(trueOrient,q));

pRMS = sqrt(mean(posErr.^2));
qRMS = sqrt(mean(qErr.^2));

fprintf('Position RMS Error\n');

Position RMS Error

fprintf ('\tX: %.2f, Y: %.2f, Z: %.2f (meters)\n\n',pRMS(1),pRMS(2),pRMS(3));

X: 0.55, Y: 0.71, Z: 0.74 (meters)

fprintf('Quaternion Distance RMS Error\n');

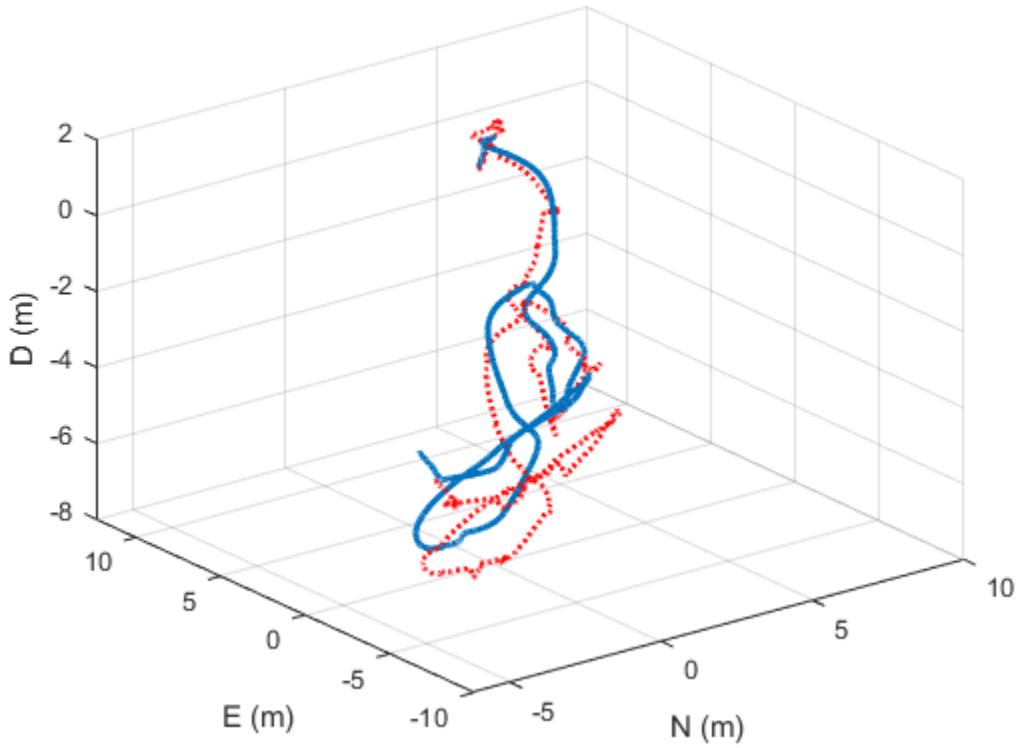
Quaternion Distance RMS Error

fprintf ('\t%.2f (degrees)\n\n', qRMS);

4.72 (degrees)
```

Visualize the true position and the estimated position.

```
plot3(truePos(:,1),truePos(:,2),truePos(:,3),'LineWidth',2)
hold on
plot3(p(:,1),p(:,2),p(:,3),'r','LineWidth',2)
grid on
xlabel('N (m)')
ylabel('E (m)')
zlabel('D (m)')
```



## Algorithms

### Dynamic Model Used in `insfilterAsync`

*Note: The following algorithm only applies to an NED reference frame.*

`insfilterAsync` implements a 28-axis continuous-discrete extended Kalman filter using sequential fusion. The filter relies on the assumption that individual sensor measurements are uncorrelated. The filter uses an omnidirectional motion model and assumes constant angular velocity and constant acceleration. The state is defined as:

$$x = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ angVel_x \\ angVel_y \\ angVel_z \\ position_N \\ position_E \\ position_D \\ v_N \\ v_E \\ v_D \\ accel_N \\ accel_E \\ accel_D \\ accelbias_x \\ accelbias_y \\ accelbias_z \\ gyrobias_x \\ gyrobias_y \\ gyrobias_z \\ geomagneticFieldVector_N \\ geomagneticFieldVector_E \\ geomagneticFieldVector_D \\ magbias_x \\ magbias_y \\ magbias_z \end{bmatrix}$$

where

- $q_0, q_1, q_2, q_3$  -- Parts of orientation quaternion. The orientation quaternion represents a frame rotation from the platform's current orientation to the local NED coordinate system.
- $angVel_x, angVel_y, angVel_z$  -- Angular velocity relative to the platform's body frame.
- $position_N, position_E, position_D$  -- Position of the platform in the local NED coordinate system.
- $v_N, v_E, v_D$  -- Velocity of the platform in the local NED coordinate system.
- $accel_N, accel_E, accel_D$  -- Acceleration of the platform in the local NED coordinate system.
- $accelbias_x, accelbias_y, accelbias_z$  -- Bias in the accelerometer reading.
- $gyrobias_x, gyrobias_y, gyrobias_z$  -- Bias in the gyroscope reading.

- $\text{geomagneticFieldVector}_N$ ,  $\text{geomagneticFieldVector}_E$ ,  $\text{geomagneticFieldVector}_D$  -- Estimate of the geomagnetic field vector at the reference location.
  - $\text{magbias}_X$ ,  $\text{magbias}_Y$ ,  $\text{magbias}_Z$  -- Bias in the magnetometer readings.

Given the conventional formation of the process equation,  $\dot{x} = f(x) + w$ ,  $w$  is the process noise,  $\dot{x}$  is the derivative of  $x$ , and:

## Extended Capabilities

## C/C++ Code Generation

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

**See Also**

`insfilterErrorState` | `insfilterMARG` | `insfilterNonholonomic`

**Introduced in R2019a**

# copy

Create copy of insfilterMARG

## Syntax

```
newFilter = copy(filter)
```

## Description

`newFilter = copy(filter)` returns a copy of the `insfilterMARG`, `filter`, with the exactly same property values.

## Input Arguments

### **filter — Filter to be copied**

`insfilterMARG`

Filter to be copied, specified as an `insfilterMARG` object.

## Output Arguments

### **newFilter — New copied filter**

`insfilterMARG`

New copied filter, returned as an `insfilterMARG` object.

## See Also

`insfilterMARG`

## Introduced in R2020b

## correct

Correct states using direct state measurements for `insfilterMARG`

### Syntax

```
correct(FUSE,idx,measurement,measurementCovariance)
```

### Description

`correct(FUSE,idx,measurement,measurementCovariance)` corrects the state and state estimation error covariance based on the measurement and measurement covariance. The measurement maps directly to the state specified by the indices `idx`.

### Input Arguments

**FUSE — insfilterMARG object**  
object

`insfilterMARG`, specified as an object.

**idx — State vector Index of measurement to correct**  
*N*-element vector of increasing integers in the range [1,22]

State vector index of measurement to correct, specified as an *N*-element vector of increasing integers in the range [1, 22].

The state values represent:

State	Units	Index
Orientation (quaternion parts)		1:4
Position (NED)	m	5:7
Velocity (NED)	m/s	8:10
Delta Angle Bias (XYZ)	rad	11:13
Delta Velocity Bias (XYZ)	m/s	14:16
Geomagnetic Field Vector (NED)	$\mu$ T	17:19
Magnetometer Bias (XYZ)	$\mu$ T	20:22

Data Types: `single` | `double`

**measurement — Direct measurement of state**  
*N*-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

**measurementCovariance — Covariance of measurement**scalar |  $N$ -element vector |  $N$ -by- $N$  matrix

Covariance of measurement, specified as a scalar,  $N$ -element vector, or  $N$ -by- $N$  matrix.  $N$  is the number of elements of the index argument, idx.

Data Types: single | double

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[insfilter](#) | [insfilterMARG](#)

**Introduced in R2018b**

## fusegps

Correct states using GPS data for `insfilterMARG`

### Syntax

```
[res,resCov] = fusegps(FUSE,position,positionCovariance)
[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

### Description

`[res,resCov] = fusegps(FUSE,position,positionCovariance)` fuses GPS position data to correct the state estimate.

`[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)` fuses GPS position and velocity data to correct the state estimate.

### Input Arguments

**FUSE — insfilterMARG object**  
object

`insfilterMARG`, specified as an object.

**position — Position of GPS receiver (LLA)**  
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

**positionCovariance — Position measurement covariance of GPS receiver ( $\text{m}^2$ )**  
3-by-3 matrix

Position measurement covariance of GPS receiver in  $\text{m}^2$ , specified as a 3-by-3 matrix.

Data Types: `single` | `double`

**velocity — Velocity of GPS receiver in local NED coordinate system (m/s)**  
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

**velocityCovariance — Velocity measurement covariance of GPS receiver ( $\text{m/s}^2$ )**  
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s<sup>2</sup>, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **res — Position and velocity residual**

1-by-6 vector of real values

Position and velocity residual, returned as a 1-by-6 vector of real values in m and m/s, respectively.

### **resCov — Residual covariance**

6-by-6 matrix of real values

Residual covariance, returned as a 6-by-6 matrix of real values.

## Extended Capabilities

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

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

## See Also

`insfilter` | `insfilterMARG`

## Introduced in R2018b

## fusemag

Correct states using magnetometer data for `insfilterMARG`

### Syntax

```
[res, resCov] = fusemag(FUSE, magReadings, magReadingsCovariance)
```

### Description

`[res, resCov] = fusemag(FUSE, magReadings, magReadingsCovariance)` fuses magnetometer data to correct the state estimate.

### Input Arguments

**FUSE — insfilterMARG object**  
object

`insfilterMARG`, specified as an object.

**magReadings — Magnetometer readings ( $\mu\text{T}$ )**  
3-element row vector

Magnetometer readings in  $\mu\text{T}$ , specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

**magReadingsCovariance — Magnetometer readings error covariance ( $\mu\text{T}^2$ )**  
scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in  $\mu\text{T}^2$ , specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

### Output Arguments

**res — Residual**  
1-by-3 vector of real values

Residual, returned a 1-by-3 vector of real values in  $\mu\text{T}$ .

Data Types: `single` | `double`

**resCov — Residual covariance**  
3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in  $(\mu\text{T})^2$ .

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`insfilter` | `insfilterMARG`

Introduced in R2018b

# residualmag

Residuals and residual covariance from magnetometer measurements for `insfilterMARG`

## Syntax

```
[res, resCov] = residualmag(FUSE, magReadings, magReadingsCovariance)
```

## Description

`[res, resCov] = residualmag(FUSE, magReadings, magReadingsCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the magnetometer readings and the corresponding covariance.

## Input Arguments

### **FUSE — insfilterMARG object**

object

`insfilterMARG`, specified as an object.

### **magReadings — Magnetometer readings ( $\mu\text{T}$ )**

3-element row vector

Magnetometer readings in  $\mu\text{T}$ , specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

### **magReadingsCovariance — Magnetometer readings error covariance ( $\mu\text{T}^2$ )**

scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in  $\mu\text{T}^2$ , specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **res — Residual**

1-by-3 vector of real values

Residual, returned as a 1-by-3 vector of real values in  $\mu\text{T}$ .

Data Types: `single` | `double`

### **resCov — Residual covariance**

3-by-3 matrix of real values

Residual covariance, returned a 3-by-3 matrix of real values in  $(\mu\text{T})^2$ .

## See Also

`insfilter` | `insfilterMARG`

**Introduced in R2020a**

## pose

Current orientation and position estimate for `insfilterMARG`

### Syntax

```
[position,orientation,velocity] = pose(FUSE)
[position,orientation,velocity] = pose(FUSE,format)
```

### Description

`[position,orientation,velocity] = pose(FUSE)` returns the current estimate of the pose and velocity.

`[position,orientation,velocity] = pose(FUSE,format)` returns the current estimate of the pose with orientation in the specified orientation format.

### Input Arguments

**FUSE — insfilterMARG object**  
object

`insfilterMARG`, specified as an object.

**format — Output orientation format**  
`'quaternion'` (default) | `'rotmat'`

Output orientation format, specified as either `'quaternion'` for a quaternion or `'rotmat'` for a rotation matrix.

Data Types: `char` | `string`

### Output Arguments

**position — Position estimate in the local NED coordinate system (m)**  
3-element row vector

Position estimate in the local NED coordinate system in meters, returned as a 3-element row vector.

Data Types: `single` | `double`

**orientation — Orientation estimate in the local NED coordinate system**  
`quaternion` (default) | 3-by-3 rotation matrix

Orientation estimate in the local NED coordinate system, specified as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix represents a frame rotation from the local NED reference frame to the body reference frame.

Data Types: `single` | `double` | `quaternion`

**velocity — Velocity estimate in the local NED coordinate system (m/s)**  
3-element row vector

---

Velocity estimate in the local NED coordinate system in m/s, returned as a 3-element row vector.

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`insfilter` | `insfilterMARG`

**Introduced in R2018b**

# **predict**

Update states using accelerometer and gyroscope data for `insfilterMARG`

## **Syntax**

```
predict(FUSE,accelReadings,gyroReadings)
```

## **Description**

`predict(FUSE,accelReadings,gyroReadings)` fuses accelerometer and gyroscope data to update the state estimate.

## **Input Arguments**

**FUSE — `insfilterMARG` object**  
object

`insfilterMARG`, specified as an object.

**accelReadings — Accelerometer readings in local sensor body coordinate system (m/s<sup>2</sup>)**  
3-element row vector

Accelerometer readings in m/s<sup>2</sup>, specified as a 3-element row vector.

Data Types: `single` | `double`

**gyroReadings — Gyroscope readings in local sensor body coordinate system (rad/s)**  
3-element row vector

Gyroscope readings in rad/s, specified as a 3-element row vector.

Data Types: `single` | `double`

## **Extended Capabilities**

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

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

## **See Also**

`insfilter` | `insfilterMARG`

**Introduced in R2018b**

# reset

Reset internal states for `insfilterMARG`

## Syntax

```
reset(FUSE)
```

## Description

`reset(FUSE)` resets the State, StateCovariance, and internal integrators to their default values.

## Input Arguments

### FUSE — `insfilterMARG` object

object

`insfilterMARG`, specified as an object.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

`insfilter` | `insfilterMARG`

Introduced in R2018b

# stateinfo

Display state vector information for `insfilterMARG`

## Syntax

```
stateinfo(FUSE)
info = stateinfo(FUSE)
```

## Description

`stateinfo(FUSE)` displays the meaning of each index of the State property and the associated units.

`info = stateinfo(FUSE)` returns a structure with fields containing descriptions of the elements of the state vector of the filter, `FUSE`.

## Examples

### State Information of `insfilterMARG`

Create an `insfilterMARG` object.

```
filter = insfilterMARG;
```

Display the state information of the created filter.

```
stateinfo(filter)
```

States	Units	Index
Orientation (quaternion parts)		1:4
Position (NAV)	m	5:7
Velocity (NAV)	m/s	8:10
Delta Angle Bias (XYZ)	rad	11:13
Delta Velocity Bias (XYZ)	m/s	14:16
Geomagnetic Field Vector (NAV)	µT	17:19
Magnetometer Bias (XYZ)	µT	20:22

Output the state information of the filter as a structure.

```
info = stateinfo(filter)

info = struct with fields:
    Orientation: [1 2 3 4]
    Position: [5 6 7]
    Velocity: [8 9 10]
    DeltaAngleBias: [11 12 13]
    DeltaVelocityBias: [14 15 16]
    GeomagneticFieldVector: [17 18 19]
    MagnetometerBias: [20 21 22]
```

## Input Arguments

**FUSE — insfilterMARG object**  
object

insfilterMARG, specified as an object.

**info — State information**  
structure

State information, returned as a structure with fields containing descriptions of the elements of the state vector of the filter. The values of each field are the corresponding indices of the state vector.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[insfilter](#) | [insfilterMARG](#)

**Introduced in R2018b**

# residual

Residuals and residual covariances from direct state measurements for `insfilterMARG`

## Syntax

```
[res, resCov] = residual(FUSE, idx, measurement, measurementCovariance)
```

## Description

`[res, resCov] = residual(FUSE, idx, measurement, measurementCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the direct state measurement and measurement covariance. The measurement maps directly to the states specified by indices, `idx`.

## Input Arguments

### FUSE — insfilterMARG object

object

`insfilterMARG`, specified as an object.

### idx — State vector index of measurement

*N*-element vector of increasing integers in the range [1,22]

State vector index of measurement, specified as an *N*-element vector of increasing integers in the range [1, 22].

The state values represent:

State	Units	Index
Orientation (quaternion parts)		1:4
Position (NED)	m	5:7
Velocity (NED)	m/s	8:10
Delta Angle Bias (XYZ)	rad	11:13
Delta Velocity Bias (XYZ)	m/s	14:16
Geomagnetic Field Vector (NED)	$\mu$ T	17:19
Magnetometer Bias (XYZ)	$\mu$ T	20:22

### measurement — Direct measurement of state

*N*-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, `idx`.

### measurementCovariance — Covariance of measurement

*N*-by-*N* matrix

Covariance of measurement, specified as an *N*-by-*N* matrix. *N* is the number of elements of the index argument, `idx`.

## Output Arguments

### **res — Measurement residual**

1-by- $N$  vector of real values

Measurement residual, returned as a 1-by- $N$  vector of real values.

### **resCov — Residual covariance**

$N$ -by- $N$  matrix of real values

Residual covariance, returned as a  $N$ -by- $N$  matrix of real values.

## See Also

`insfilterMARG`

## Introduced in R2020a

# residualgps

Residuals and residual covariance from GPS measurements for `insfilterMARG`

## Syntax

```
[res,resCov] = residualgps(FUSE,position,positionCovariance)
[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

## Description

`[res,resCov] = residualgps(FUSE,position,positionCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

`[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,
velocityCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

## Input Arguments

**FUSE — insfilterMARG object**  
object

`insfilterMARG`, specified as an object.

**position — Position of GPS receiver (LLA)**  
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

**positionCovariance — Position measurement covariance of GPS receiver (m<sup>2</sup>)**  
3-by-3 matrix

Position measurement covariance of GPS receiver in m<sup>2</sup>, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

**velocity — Velocity of GPS receiver in local NED coordinate system (m/s)**  
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

**velocityCovariance — Velocity measurement covariance of GPS receiver (m/s<sup>2</sup>)**  
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s<sup>2</sup>, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **res — Position and velocity residual**

1-by-3 vector of real values | 1-by-6 vector of real values

Position and velocity residual, returned as a 1-by-3 vector of real values if the inputs only contain position information, and returned as 1-by-6 vector of real values if the inputs also contain velocity information.

### **resCov — Residual covariance**

3-by-3 matrix of real values | 6-by-6 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values if the inputs only contain position information, and a 6-by-6 matrix of real values if the inputs also contain velocity information.

## See Also

`insfilter` | `insfilterMARG`

**Introduced in R2020a**

# insfilterMARG

Estimate pose from MARG and GPS data

## Description

The `insfilterMARG` object implements sensor fusion of MARG and GPS data to estimate pose in the NED (or ENU) reference frame. MARG (magnetic, angular rate, gravity) data is typically derived from magnetometer, gyroscope, and accelerometer sensors. The filter uses a 22-element state vector to track the orientation quaternion, velocity, position, MARG sensor biases, and geomagnetic vector. The `insfilterMARG` object uses an extended Kalman filter to estimate these quantities.

## Creation

### Syntax

```
filter = insfilterMARG  
filter = insfilterMARG('ReferenceFrame',RF)  
filter = insfilterMARG(___,Name,Value)
```

### Description

`filter = insfilterMARG` creates an `insfilterMARG` object with default property values.

`filter = insfilterMARG('ReferenceFrame',RF)` allows you to specify the reference frame, RF, of the filter. Specify RF as 'NED' (North-East-Down) or 'ENU' (East-North-Up). The default value is 'NED'.

`filter = insfilterMARG(___,Name,Value)` also allows you set properties of the created `filter` using one or more name-value pairs. Enclose each property name in single quotes.

## Properties

### IMUSampleRate — Sample rate of the IMU (Hz)

100 (default) | positive scalar

Sample rate of the inertial measurement unit (IMU) in Hz, specified as a positive scalar.

Data Types: `single` | `double`

### ReferenceLocation — Reference location (deg, deg, meters)

[0 0 0] (default) | 3-element positive row vector

Reference location, specified as a 3-element row vector in geodetic coordinates (latitude, longitude, and altitude). Altitude is the height above the reference ellipsoid model, WGS84. The reference location units are [degrees degrees meters].

Data Types: `single` | `double`

**GyroscopeNoise — Multiplicative process noise variance from gyroscope (rad/s)<sup>2</sup>**

1e-9 (default) | scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope in (rad/s)<sup>2</sup>, specified as a scalar or 3-element row vector of positive real finite numbers.

- If GyroscopeNoise is specified as a row vector, the elements correspond to the noise in the x, y, and z axes of the gyroscope, respectively.
- If GyroscopeNoise is specified as a scalar, the single element is applied to the x, y, and z axes of the gyroscope.

Data Types: single | double

**GyroscopeBiasNoise — Multiplicative process noise variance from gyroscope bias (rad/s)<sup>2</sup>**

1e-10 (default) | positive scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope bias in (rad/s)<sup>2</sup>, specified as a scalar or 3-element row vector of positive real numbers.

- If GyroscopeBiasNoise is specified as a row vector, the elements correspond to the noise in the x, y, and z axes of the gyroscope bias, respectively.
- If GyroscopeBiasNoise is specified as a scalar, the single element is applied to each axis.

Data Types: single | double

**AccelerometerNoise — Multiplicative process noise variance from accelerometer (m/s<sup>2</sup>)<sup>2</sup>**

1e-4 (default) | scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer in (m/s<sup>2</sup>)<sup>2</sup>, specified as a scalar or 3-element row vector of positive real finite numbers.

- If AccelerometerNoise is specified as a row vector, the elements correspond to the noise in the x, y, and z axes of the accelerometer, respectively.
- If AccelerometerNoise is specified as a scalar, the single element is applied to each axis.

Data Types: single | double

**AccelerometerBiasNoise — Multiplicative process noise variance from accelerometer bias (m/s<sup>2</sup>)<sup>2</sup>**

1e-4 (default) | positive scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer bias in (m/s<sup>2</sup>)<sup>2</sup>, specified as a scalar or 3-element row vector of positive real numbers.

- If AccelerometerBiasNoise is specified as a row vector, the elements correspond to the noise in the x, y, and z axes of the accelerometer bias, respectively.
- If AccelerometerBiasNoise is specified as a scalar, the single element is applied to each axis.

Data Types: single | double

**GeomagneticVectorNoise — Additive process noise for geomagnetic vector (µT<sup>2</sup>)**

1e-6 (default) | positive scalar | 3-element row vector

Additive process noise for geomagnetic vector in µT<sup>2</sup>, specified as a scalar or 3-element row vector of positive real numbers.

- If `GeomagneticVectorNoise` is specified as a row vector, the elements correspond to the noise in the  $x$ ,  $y$ , and  $z$  axes of the geomagnetic vector, respectively.
- If `GeomagneticVectorNoise` is specified as a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

### **MagnetometerBiasNoise — Additive process noise for magnetometer bias ( $\mu\text{T}^2$ )**

`0.1` (default) | positive scalar | 3-element row vector

Additive process noise for magnetometer bias in  $\mu\text{T}^2$ , specified as a scalar or 3-element row vector.

- If `MagnetometerBiasNoise` is specified as a row vector, the elements correspond to the noise in the  $x$ ,  $y$ , and  $z$  axes of the magnetometer bias, respectively.
- If `MagnetometerBiasNoise` is specified as a scalar, the single element is applied to each axis.

Data Types: `single` | `double`

### **State — State vector of extended Kalman filter**

22-element column vector

State vector of the extended Kalman filter. The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Position (NED or ENU)	m	5:7
Velocity (NED or ENU)	m/s	8:10
Delta Angle Bias (XYZ)	rad	11:13
Delta Velocity Bias (XYZ)	m/s	14:16
Geomagnetic Field Vector (NED or ENU)	$\mu\text{T}$	17:19
Magnetometer Bias (XYZ)	$\mu\text{T}$	20:22

Data Types: `single` | `double`

### **StateCovariance — State error covariance for extended Kalman filter**

`eye(22)*1e-6` (default) | 22-by-22 matrix

State error covariance for the extended Kalman filter, specified as a 22-by-22-element matrix, or real numbers.

Data Types: `single` | `double`

## **Object Functions**

<code>correct</code>	Correct states using direct state measurements for <code>insfilterMARG</code>
<code>residual</code>	Residuals and residual covariances from direct state measurements for <code>insfilterMARG</code>
<code>fusegps</code>	Correct states using GPS data for <code>insfilterMARG</code>
<code>residualgps</code>	Residuals and residual covariance from GPS measurements for <code>insfilterMARG</code>
<code>fusemag</code>	Correct states using magnetometer data for <code>insfilterMARG</code>
<code>residualmag</code>	Residuals and residual covariance from magnetometer measurements for <code>insfilterMARG</code>
<code>pose</code>	Current orientation and position estimate for <code>insfilterMARG</code>

---

<code>predict</code>	Update states using accelerometer and gyroscope data for insfilterMARG
<code>reset</code>	Reset internal states for insfilterMARG
<code>stateinfo</code>	Display state vector information for insfilterMARG
<code>copy</code>	Create copy of insfilterMARG

## Examples

### Estimate Pose of UAV

This example shows how to estimate the pose of an unmanned aerial vehicle (UAV) from logged sensor data and ground truth pose.

Load the logged sensor data and ground truth pose of an UAV.

```
load uavshort.mat
```

Initialize the `insfilterMARG` filter object.

```
f = insfilterMARG;
f.IMUSampleRate = imuFs;
f.ReferenceLocation = refloc;
f.AccelerometerBiasNoise = 2e-4;
f.AccelerometerNoise = 2;
f.GyroscopeBiasNoise = 1e-16;
f.GyroscopeNoise = 1e-5;
f.MagnetometerBiasNoise = 1e-10;
f.GeomagneticVectorNoise = 1e-12;
f.StateCovariance = 1e-9*ones(22);
f.State = initstate;

gpsidx = 1;
N = size(accel,1);
p = zeros(N,3);
q = zeros(N,1, 'quaternion');
```

Fuse accelerometer, gyroscope, magnetometer, and GPS data.

```
for ii = 1:size(accel,1) % Fuse IMU
    f.predict(accel(ii,:), gyro(ii,:));

    if ~mod(ii,fix(imuFs/2)) % Fuse magnetometer at 1/2 the IMU rate
        f.fusemag(mag(ii,:),Rmag);
    end

    if ~mod(ii,imuFs) % Fuse GPS once per second
        f.fusegps(lla(gpsidx,:),Rpos, gpsvel(gpsidx,:),Rvel);
        gpsidx = gpsidx + 1;
    end

    [p(ii,:),q(ii)] = pose(f); %Log estimated pose
end
```

Calculate and display RMS errors.

```
posErr = truePos - p;
qErr = rad2deg(dist(trueOrient,q));
```

```

pRMS = sqrt(mean(posErr.^2));
qRMS = sqrt(mean(qErr.^2));
fprintf('Position RMS Error\n\tX: %.2f, Y: %.2f, Z: %.2f (meters)\n\n',pRMS(1),pRMS(2),pRMS(3));

Position RMS Error
    X: 0.57, Y: 0.53, Z: 0.68 (meters)

fprintf('Quaternion Distance RMS Error\n\t%.2f (degrees)\n\n',qRMS);

Quaternion Distance RMS Error
    0.28 (degrees)

```

## Algorithms

*Note: The following algorithm only applies to an NED reference frame.*

`insfilterMARG` uses a 22-axis extended Kalman filter structure to estimate pose in the NED reference frame. The state is defined as:

$$x = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ position_N \\ position_E \\ position_D \\ v_N \\ v_E \\ v_D \\ \Delta\theta bias_X \\ \Delta\theta bias_Y \\ \Delta\theta bias_Z \\ \Delta v bias_X \\ \Delta v bias_Y \\ \Delta v bias_Z \\ geomagneticFieldVector_N \\ geomagneticFieldVector_E \\ geomagneticFieldVector_D \\ magbias_X \\ magbias_Y \\ magbias_Z \end{bmatrix}$$

where

- $q_0, q_1, q_2, q_3$  -- Parts of orientation quaternion. The orientation quaternion represents a frame rotation from the platform's current orientation to the local NED coordinate system.
- $position_N, position_E, position_D$  -- Position of the platform in the local NED coordinate system.
- $\nu_N, \nu_E, \nu_D$  -- Velocity of the platform in the local NED coordinate system.
- $\Delta\theta bias_X, \Delta\theta bias_Y, \Delta\theta bias_Z$  -- Bias in the integrated gyroscope reading.
- $\Delta\nu bias_X, \Delta\nu bias_Y, \Delta\nu bias_Z$  -- Bias in the integrated accelerometer reading.
- $geomagneticFieldVector_N, geomagneticFieldVector_E, geomagneticFieldVector_D$  -- Estimate of the geomagnetic field vector at the reference location.
- $magbias_X, magbias_Y, magbias_Z$  -- Bias in the magnetometer readings.

Given the conventional formation of the predicted state estimate,

$$x_{k|k-1} = f(\hat{x}_{k-1|k-1}, u_k)$$

$u_k$  is controlled by accelerometer and gyroscope data that has been converted to delta velocity and delta angle through trapezoidal integration. The predicted state estimation is:

$x_k|k-1 =$

$$\left| \begin{array}{l}
 q_0 - q_1 \left( \frac{\Delta\theta_X - \Delta\theta bias_X}{2} \right) - q_2 \left( \frac{\Delta\theta_Y - \Delta\theta bias_Y}{2} \right) - q_3 \left( \frac{\Delta\theta_Z - \Delta\theta bias_Z}{2} \right) \\
 q_1 + q_0 \left( \frac{\Delta\theta_X - \Delta\theta bias_X}{2} \right) - q_3 \left( \frac{\Delta\theta_Y - \Delta\theta bias_Y}{2} \right) + q_2 \left( \frac{\Delta\theta_Z - \Delta\theta bias_Z}{2} \right) \\
 q_2 + q_3 \left( \frac{\Delta\theta_X - \Delta\theta bias_X}{2} \right) + q_0 \left( \frac{\Delta\theta_Y - \Delta\theta bias_Y}{2} \right) - q_1 \left( \frac{\Delta\theta_Z - \Delta\theta bias_Z}{2} \right) \\
 q_3 - q_2 \left( \frac{\Delta\theta_X - \Delta\theta bias_X}{2} \right) + q_1 \left( \frac{\Delta\theta_Y - \Delta\theta bias_Y}{2} \right) + q_0 \left( \frac{\Delta\theta_Z - \Delta\theta bias_Z}{2} \right) \\
 \quad position_N + (\Delta t)(v_N) \\
 \quad position_E + (\Delta t)(v_E) \\
 \quad position_D + (\Delta t)(v_D) \\
 \\ 
 v_N + (\Delta t)(g_N) + (\Delta\nu_X - \Delta\nu bias_X)(q_0^2 + q_1^2 - q_2^2 - q_3^2) - 2(\Delta\nu_Y - \Delta\nu bias_Y)(q_0 q_3 - q_1 q_2) + 2(\Delta\nu_Z - \Delta\nu bias_Z)(q_0 q_2 + q_1 q_3) \\
 v_E + (\Delta t)(g_E) + (\Delta\nu_Y - \Delta\nu bias_Y)(q_0^2 - q_1^2 + q_2^2 - q_3^2) + 2(\Delta\nu_X - \Delta\nu bias_X)(q_0 q_3 + q_1 q_2) - 2(\Delta\nu_Z - \Delta\nu bias_Z)(q_0 q_1 - q_2 q_3) \\
 v_D + (\Delta t)(g_D) + (\Delta\nu_Z - \Delta\nu bias_Z)(q_0^2 - q_1^2 - q_2^2 + q_3^2) - 2(\Delta\nu_X - \Delta\nu bias_X)(q_0 q_2 - q_1 q_3) + 2(\Delta\nu_Y - \Delta\nu bias_Y)(q_0 q_1 + q_2 q_3) \\
 \quad \Delta\theta bias_X \\
 \quad \Delta\theta bias_Y
 \end{array} \right|$$

where

- $\Delta\theta_X, \Delta\theta_Y, \Delta\theta_Z$  -- Integrated gyroscope reading.
- $\Delta\nu_X, \Delta\nu_Y, \Delta\nu_Z$  -- Integrated accelerometer readings.
- $\Delta t$  -- IMU sample time.
- $g_N, g_E, g_D$  -- Constant gravity vector in the NED frame.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[insfilterAsync](#) | [insfilterErrorState](#) | [insfilterNonholonomic](#)

### Topics

[“Estimate Position and Orientation of a Ground Vehicle”](#)

### Introduced in R2018b

## copy

Create copy of `insfilterNonholonomic`

### Syntax

```
newFilter = copy(filter)
```

### Description

`newFilter = copy(filter)` returns a copy of the `insfilterNonholonomic`, `filter`, with the exactly same property values.

### Input Arguments

#### **filter — Filter to be copied**

`insfilterNonholonomic`

Filter to be copied, specified as an `insfilterNonholonomic` object.

### Output Arguments

#### **newFilter — New copied filter**

`insfilterNonholonomic`

New copied filter, returned as an `insfilterNonholonomic` object.

### See Also

`insfilterNonholonomic`

### Introduced in R2020b

# correct

Correct states using direct state measurements for `insfilterNonholonomic`

## Syntax

```
correct(FUSE,idx,measurement,measurementCovariance)
```

## Description

`correct(FUSE,idx,measurement,measurementCovariance)` corrects the state and state estimation error covariance based on the measurement and measurement covariance. The measurement maps directly to the state specified by the indices `idx`.

## Input Arguments

**FUSE — `insfilterNonholonomic` object**  
object

`insfilterNonholonomic`, specified as an object.

**idx — State vector Index of measurement to correct**  
*N*-element vector of increasing integers in the range [1,16]

State vector index of measurement to correct, specified as an *N*-element vector of increasing integers in the range [1,16].

The state values represent:

State	Units	Index
Orientation (quaternion parts)		1:4
Gyroscope bias (XYZ)	rad/s	5:7
Position (NED)	m	8:10
Velocity (NED)	m/s	11:13
Accelerometer Bias (XYZ)	m/s <sup>2</sup>	14:16

Data Types: `single` | `double`

**measurement — Direct measurement of state**  
*N*-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

**measurementCovariance — Covariance of measurement**  
scalar | *N*-element vector | *N*-by-*N* matrix

Covariance of measurement, specified as a scalar,  $N$ -element vector, or  $N$ -by- $N$  matrix.  $N$  is the number of elements of the index argument, `idx`.

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`insfilter` | `insfilterNonholonomic`

**Introduced in R2018b**

# fusegps

Correct states using GPS data for `insfilterNonholonomic`

## Syntax

```
[res,resCov] = fusegps(FUSE,position,positionCovariance)
[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

## Description

`[res,resCov] = fusegps(FUSE,position,positionCovariance)` fuses GPS position data to correct the state estimate.

`[res,resCov] = fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)` fuses GPS position and velocity data to correct the state estimate.

## Input Arguments

**FUSE — insfilterNonholonomic object**  
object

`insfilterNonholonomic`, specified as an object.

**position — Position of GPS receiver (LLA)**  
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

**positionCovariance — Position measurement covariance of GPS receiver ( $\text{m}^2$ )**  
3-by-3 matrix

Position measurement covariance of GPS receiver in  $\text{m}^2$ , specified as a 3-by-3 matrix.

Data Types: `single` | `double`

**velocity — Velocity of GPS receiver in local NED coordinate system (m/s)**  
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

**velocityCovariance — Velocity measurement covariance of GPS receiver ( $\text{m}/\text{s}^2$ )**  
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s<sup>2</sup>, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **res — Position and course residual**

1-by-4 vector of real values

Position and course residual, returned as a 1-by-6 vector of real values in m and rad/s, respectively.

### **resCov — Residual covariance**

4-by-4 matrix of real values

Residual covariance, returned as a 4-by-4 matrix of real values.

## Extended Capabilities

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

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

## See Also

`insfilter` | `insfilterNonholonomic`

## Introduced in R2018b

# residual

Residuals and residual covariances from direct state measurements for `insfilterNonholonomic`

## Syntax

```
[res, resCov] = residual(FUSE, idx, measurement, measurementCovariance)
```

## Description

`[res, resCov] = residual(FUSE, idx, measurement, measurementCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the direct state measurement and measurement covariance. The `measurement` maps directly to the states specified by indices, `idx`.

## Input Arguments

**FUSE — insfilterNonholonomic object**  
object

`insfilterNonholonomic`, specified as an object.

**idx — State vector Index of measurement to correct**  
*N*-element vector of increasing integers in the range [1,16]

State vector index of measurement to correct, specified as an *N*-element vector of increasing integers in the range [1,16].

The state values represent:

State	Units	Index
Orientation (quaternion parts)		1:4
Gyroscope bias (XYZ)	rad/s	5:7
Position (NED)	m	8:10
Velocity (NED)	m/s	11:13
Accelerometer Bias (XYZ)	m/s <sup>2</sup>	14:16

Data Types: `single` | `double`

**measurement — Direct measurement of state**  
*N*-element vector

Direct measurement of state, specified as a *N*-element vector. *N* is the number of elements of the index argument, `idx`.

**measurementCovariance — Covariance of measurement**  
*N*-by-*N* matrix

Covariance of measurement, specified as an *N*-by-*N* matrix. *N* is the number of elements of the index argument, `idx`.

## Output Arguments

### **res — Measurement residual**

1-by- $N$  vector of real values

Measurement residual, returned as a 1-by- $N$  vector of real values.

### **resCov — Residual covariance**

$N$ -by- $N$  matrix of real values

Residual covariance, returned as a  $N$ -by- $N$  matrix of real values.

## Extended Capabilities

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

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

## See Also

[insfilter](#) | [insfilterNonholonomic](#)

**Introduced in R2020a**

# residualgps

Residuals and residual covariance from GPS measurements for `insfilterNonholonomic`

## Syntax

```
[res,resCov] = residualgps(FUSE,position,positionCovariance)
[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

## Description

`[res,resCov] = residualgps(FUSE,position,positionCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

`[res,resCov] = residualgps(FUSE,position,positionCovariance,velocity,
velocityCovariance)` computes the residual, `res`, and the residual covariance, `resCov`, based on the GPS position measurement and covariance.

## Input Arguments

**FUSE — insfilterNonholonomic object**  
object

`insfilterNonholonomic`, specified as an object.

**position — Position of GPS receiver (LLA)**  
3-element row vector

Position of GPS receiver in geodetic latitude, longitude, and altitude (LLA) specified as a real finite 3-element row vector. Latitude and longitude are in degrees with north and east being positive. Altitude is in meters.

Data Types: `single` | `double`

**positionCovariance — Position measurement covariance of GPS receiver (m<sup>2</sup>)**  
3-by-3 matrix

Position measurement covariance of GPS receiver in m<sup>2</sup>, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

**velocity — Velocity of GPS receiver in local NED coordinate system (m/s)**  
3-element row vector

Velocity of the GPS receiver in the local NED coordinate system in m/s, specified as a 3-element row vector.

Data Types: `single` | `double`

**velocityCovariance — Velocity measurement covariance of GPS receiver (m/s<sup>2</sup>)**  
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s<sup>2</sup>, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

## Output Arguments

### **res — Position and course residual**

1-by-3 vector of real values | 1-by-4 vector of real values

Position and course residual, returned as a 1-by-3 vector of real values if the inputs only contain position information, and returned as a 1-by-4 vector of real values if the inputs also contain velocity information.

### **resCov — Residual covariance**

3-by-3 matrix of real values | 4-by-4 matrix of real values

Residual covariance, returned as a 3-by-3 matrix of real values if the inputs only contain position information, and a 4-by-4 vector of real values if the inputs also contain velocity information.

## Extended Capabilities

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

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

### **See Also**

`insfilterNonholonomic`

### **Introduced in R2020a**

# fuserSourceConfiguration

Configuration of source used with track fuser

## Description

A fuserSourceConfiguration object contains the configuration information of a source used with a track fuser. A source of a track fuser is a tracking system (such as a tracker or another track fuser) that outputs tracks to the track fuser.

## Creation

### Syntax

```
config = fuserSourceConfiguration(SourceIndex)
config = fuserSourceConfiguration(SourceIndex,Name,Value)
```

#### Description

`config = fuserSourceConfiguration(SourceIndex)` creates a source configuration object to use with a track fuser. You must specify `SourceIndex` as a positive integer. The other properties of the configuration take default values.

`config = fuserSourceConfiguration(SourceIndex,Name,Value)` allows you to specify additional properties using one or more name-value pairs. Enclose each property name in single quotes.

## Properties

### SourceIndex — Unique index for source system

positive integer

Unique index for the source system, specified as a positive integer. This property distinguishes different source systems that output tracks to the fuser.

Example: 2

### IsInternalSource — Indicate if the source is internal to the fuser

true (default) | false

Indicate if the source is internal to the fuser, specified as `true` or `false`. An internal source is a source that the fuser directly fuses tracks from even if the tracks are not self reported. For example, if the fuser is at the vehicle level, a tracking radar installed on this vehicle is considered internal, while another vehicle that reports fused tracks is considered external.

Data Types: logical

### IsInitializingCentralTracks — Indicate if source can initialize central track

true (default) | false

Indicate if the source can initialize a central track in the fuser, specified as `true` or `false`. A central track is a track maintained in the fuser.

Example: `false`

Data Types: logical

**LocalToCentralTransformFcn — Function to transform track from local to central state space**

`@track(track)` (default) | function handle

Function to transform a track from local to central state space, specified as a function handle. The default transform function, `@track(track)`, makes no transformation.

Data Types: function\_handle

**CentralToLocalTransformFcn — Function to transform track from central to local state space**

`@track(track)` (default) | function handle

Function to transform a track from central to local state space, specified as a function handle. The default transform function, `@track(track)`, makes no transformation.

Data Types: function\_handle

## Examples

### Create Fusion Configuration for Source

Create a fusion configuration for a source with `SourceIndex` equal to 3.

```
config = fuserSourceConfiguration(3)

config =
  fuserSourceConfiguration with properties:

    SourceIndex: 3
    IsInternalSource: 1
    IsInitializingCentralTracks: 1
    LocalToCentralTransformFcn: @(track)track
    CentralToLocalTransformFcn: @(track)track
```

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`objectTrack` | `trackFuser`

### Introduced in R2019b

# ggiwphd

Gamma Gaussian Inverse Wishart (GGIW) PHD filter

## Description

The `ggiwphd` object is a filter that implements the probability hypothesis density (PHD) using a mixture of Gamma Gaussian Inverse-Wishart components. GGIW implementation of a PHD filter is typically used to track extended objects. An extended object can produce multiple detections per sensor, and the GGIW filter uses the random matrix model to account for the spatial distribution of these detections. The filter consists of three distributions to represent the state of an extended object.

- 1 Gaussian distribution — represents the kinematic state of the extended object.
- 2 Gamma distribution — represents the expected number of detections on a sensor from the extended object.
- 3 Inverse-Wishart (IW) distribution — represents the spatial extent of the target. In 2-D space, the extent is represented by a 2-by-2 random positive definite matrix, which corresponds to a 2-D ellipse description. In 3-D space, the extent is represented by a 3-by-3 random matrix, which corresponds to a 3-D ellipsoid description. The probability density of these random matrices is given as an Inverse-Wishart distribution.

For details about `ggiwphd`, see [1] and [2].

---

**Note** `ggiwphd` object is not compatible with `trackerGNN`, `trackerJPDA`, and `trackerTOMHT` system objects.

---

## Creation

### Syntax

```
PHD = ggiwphd
PHD = ggiwphd(States,StateCovariances)
phd = ggiwphd(States,StateCovariances,Name,Value)
```

### Description

`PHD = ggiwphd` creates a `ggiwphd` filter with default property values.

`PHD = ggiwphd(States,StateCovariances)` allows you to specify the `States` and `StateCovariances` of the Gaussian distribution for each component in the density. `States` and `StateCovariances` set the properties of the same names.

`phd = ggiwphd(States,StateCovariances,Name,Value)` also allows you to set properties for the filter using one or more name-value pairs. Enclose each property name in quotes.

## Properties

### **States — State of each component in filter**

*P*-by-*N* matrix

State of each component in the filter, specified as a *P*-by-*N* matrix, where *P* is the dimension of the state and *N* is the number of components. Each column of the matrix corresponds to the state of each component. The default value for **States** is a 6-by-2 matrix, in which the elements of the first column are all 0, and the elements of the second column are all 1.

Data Types: `single` | `double`

### **StateCovariances — State estimate error covariance of each component in filter**

*P*-by-*P*-by-*N* array

State estimate error covariance of each component in the filter, specified as a *P*-by-*P*-by-*N* array, where *P* is the dimension of the state and *N* is the number of components. Each page (*P*-by-*P* matrix) of the array corresponds to the covariance matrix of each component. The default value for **StateCovariances** is a 6-by-6-by-2 array, in which each page (6-by-6 matrix) is an identity matrix.

Data Types: `single` | `double`

### **PositionIndex — Indices of position coordinates in state**

[1 3 5] | row vector of positive integers

Indices of position coordinates in the state, specified as a row vector of positive integers. For example, by default the state is arranged as [x;vx;y;vy;z;vz] and the corresponding position index is [1 3 5] representing x-, y- and z-position coordinates.

Example: [1 2 3]

Data Types: `single` | `double`

### **StateTransitionFcn — State transition function**

`@constvel` (default) | 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 also include noise values.

- If **HasAdditiveProcessNoise** is `true`, specify the function using one of these syntaxes:

`x(k) = transitionfcn(x(k-1))`

`x(k) = transitionfcn(x(k-1),dT)`

where `x(k)` is the state estimate at time *k*, and `dT` is the time step.

- If **HasAdditiveProcessNoise** is `false`, specify the function using one of these syntaxes:

`x(k) = transitionfcn(x(k-1),w(k-1))`

`x(k) = transitionfcn(x(k-1),w(k-1),dT)`

where `x(k)` is the state estimate at time *k*, `w(k)` is the process noise at time *k*, and `dT` is the time step.

Example: `@constacc`

Data Types: `function_handle`

**StateTransitionJacobianFcn — Jacobian of state transition function**

@constveljac (default) | function handle

The Jacobian of the state transition function, specified as a function handle. This function has the same input arguments as the state transition function.

- If **HasAdditiveProcessNoise** is **true**, specify the Jacobian function using one of these syntaxes:

```
Jx(k) = statejacobianfcn(x(k))
```

```
Jx(k) = statejacobianfcn(x(k),dT)
```

where  $x(k)$  is the state at time  $k$ ,  $dT$  is the time step, and  $Jx(k)$  denotes the Jacobian of the state transition function with respect to the state. The Jacobian is an  $M$ -by- $M$  matrix at time  $k$ , where  $M$  is the dimension of the state.

- If **HasAdditiveProcessNoise** is **false**, specify the Jacobian function using one of these syntaxes:

```
[Jx(k),Jw(k)] = statejacobianfcn(x(k),w(k))
```

```
[Jx(k),Jw(k)] = statejacobianfcn(x(k),w(k),dT)
```

where  $w(k)$  is a  $Q$ -element vector of the process noise at time  $k$ .  $Q$  is the dimension of the process noise. Unlike the case of additive process noise, the process noise vector in the nonadditive noise case need not have the same dimensions as the state vector.

$Jw(k)$  denotes the  $M$ -by- $Q$  Jacobian of the predicted state with respect to the process noise elements, where  $M$  is the dimension of the state.

If not specified, the Jacobians are computed by numerical differencing at each call of the **predict** function. This computation can increase the processing time and numerical inaccuracy.

Example: @constaccjac

Data Types: function\_handle

**ProcessNoise — Process noise covariance**

eye(3) (default) | positive real-valued scalar | positive-definite real-valued matrix

Process noise covariance:

- When **HasAdditiveProcessNoise** is **true**, specify the process noise covariance as a 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** object function.

Example: [1.0 0.05; 0.05 2]

**HasAdditiveProcessNoise — Model additive process noise**

false (default)

Option to model processes 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.

Example: true

**Shapes — Shape parameter of Gamma distribution for each component**

[1 1] (default) | 1-by- $N$  row vector of positive real values

Shape parameter of Gamma distribution for each component, specified as a 1-by- $N$  row vector of positive real values.  $N$  is the number of components in the density.

Example: [1.0 0.95 2]

Data Types: single | double

**Rates — Rate parameter of Gamma distribution for each component**

[1 1] (default) | 1-by- $N$  row vector of positive real value

Rate parameter of Gamma distribution for each component, specified as a 1-by- $N$  row vector of positive real values.  $N$  is the number of components in the density.

Example: [1.2 0.85 1.5]

Data Types: single | double

**GammaForgettingFactors — Forgetting factor of Gamma distribution for each component**

[1 1] (default) | 1-by- $N$  row vector of positive real value

Forgetting factor of Gamma distribution for each component, specified as a 1-by- $N$  row vector of positive real values.  $N$  is the number of components in the density. During prediction, for each component, the Gamma distribution parameters, shape ( $\alpha$ ) and rate ( $\beta$ ), are both divided by forgetting factor  $n$ :

$$a_{k+1 \mid k} = \frac{\alpha_k}{n_k}$$

$$\beta_{k+1 \mid k} = \frac{\beta_k}{n_k}$$

where  $k$  and  $k+1$  represent two consecutive time steps. The mean ( $E$ ) and variance ( $Var$ ) of a Gamma distribution are:

$$E = \frac{\alpha}{\beta}$$

$$Var = \frac{\alpha}{\beta^2}$$

Therefore, the division action will keep the expected measurement rate as a constant, but increase the variance of the Gamma distribution exponentially with time if the forgetting factor  $n$  is larger than 1.

Example: [1.2 1.1 1.4]

Data Types: single | double

**DegreesOfFreedom — Degrees of freedom parameter of Inverse-Wishart distribution for each component**

[100 100] (default) | 1-by- $N$  row vector of positive real value

Degrees of freedom parameter of Inverse-Wishart distribution for each component, specified as a 1-by- $N$  row vector of positive real values.  $N$  is the number of components in the density.

Example: [55.2 31.1 20.4]

Data Types: single | double

**ScaleMatrices — Scale matrix of Inverse-Wishart distribution for each component**  
 $d$ -by- $d$ -by- $N$  array of positive real value

Scale matrix of Inverse-Wishart distribution for each component, specified as a  $d$ -by- $d$ -by- $N$  array of positive real values.  $d$  is the dimension of the space (for example,  $d = 2$  for 2-D space), and  $N$  is the number of components in the density. The default value for **ScaleMatrices** is a 3-by-3-by-2 array, where each page (3-by-3 matrix) of the array is `100*eye(3)`.

Example: `20*eye(3,3,4)`

Data Types: single | double

**ExtentRotationFcn — Rotation transition function of target's extent**  
 $@(x, varargin) eye(3)$  (default) | function handle

Rotation transition function of target's extent, specified as a function handle. The function allows predicting the rotation of the target's extent when the object's angular velocity is estimated in the state vector. To define your own extent rotation function, follow the syntax given by

`R = myRotationFcn(x, dT)`

where  $x$  is the component state,  $dT$  is the time step, and  $R$  is the corresponding rotation matrix. Note that  $R$  is returned as a 2-by-2 matrix if the extent is 2-D, and a 3-by-3 matrix if the extent is 3-D. The extent at the next step is given by

$$Ex(t + dT) = R \times Ex(t) \times R^T$$

where  $Ex(t)$  is the extent at time  $t$ .

Example: `@myRotationFcn`

Data Types: function\_handle

**TemporalDecay — Temporal decay factor of IW distribution**  
 $100$  (default) | positive scalar

Temporal decay factor of IW distribution, specified as a positive scalar. You can use this property to control the extent uncertainty (variance of IW distribution) during prediction. The smaller the **TemporalDecay** value is, the faster the variance of IW distribution increases.

Example: `120`

Data Types: single | double

**Labels — Label of each component in mixture**  
 $[0 \ 0]$  (default) | 1-by- $N$  row vector of nonnegative integer

Label of each component in the mixture, specified as a 1-by- $N$  row vector of nonnegative integers.  $N$  is the number of components in the density. Each component can only have one label, but multiple components can share the same label.

Example: `[1 2 3]`

Data Types: single | double

**Weights — Weight of each component in mixture**

[1 1] (default) | 1-by- $N$  row vector of positive real value

Weight of each component in the density, specified as a 1-by- $N$  row vector of positive real values.  $N$  is the number of components in the density. The weights are given in the sequence as shown in the `labels` property.

Example: [1.1 0.82 1.1]

Data Types: `single` | `double`

**Detections — Detections**

$K$ -element cell array of `objectDetection` objects

Detections, specified as a  $K$ -element cell array of `objectDetection` objects, where  $K$  is the number of detections. You can create detections directly, or you can obtain detections from the outputs of sensor objects, such as `radarSensor`, `monostaticRadarSensor`, `irSensor`, and `sonarSensor`.

Data Types: `single` | `double`

**MeasurementFcn — Measurement model function**

@cvmeas (default) | function handle

Measurement model function, specified as a function handle. This function specifies the transition from state to measurement. Input to the function is the  $P$ -element state vector. The output is the  $M$ -element measurement vector. The function can take additional input arguments, such as sensor position and orientation.

- If `HasAdditiveMeasurementNoise` is `true`, specify the function using one of these syntaxes:

`z(k) = measurementfcn(x(k))`

`z(k) = measurementfcn(x(k), parameters)`

where `x(k)` is the state at time  $k$  and `z(k)` is the corresponding measurement. 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)`

where `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.

Example: @cameas

Data Types: `function_handle`

**MeasurementJacobianFcn — Jacobian of measurement function**

@cvmeasjac (default) | 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 `HasAdditiveMeasurmentNoise` is `true`, specify the Jacobian function using one of these syntaxes:

```
Jmx(k) = measjacobianfcn(x(k))
Jmx(k) = measjacobianfcn(x(k),parameters)
```

where  $x(k)$  is the state at time  $k$ .  $Jmx(k)$  denotes the  $M$ -by- $P$  Jacobian of the measurement function with respect to the state.  $M$  is the dimension of the measurement, and  $P$  is the dimension of 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:

```
[Jmx(k),Jmv(k)] = measjacobianfcn(x(k),v(k))
[Jmx(k),Jmv(k)] = measjacobianfcn(x(k),v(k),parameters)
```

where  $x(k)$  is the state at time  $k$  and  $v(k)$  is an  $R$ -dimensional sample noise vector.  $Jmx(k)$  denotes the  $M$ -by- $P$  Jacobian matrix of the measurement function with respect to the state.  $Jmv(k)$  denotes the Jacobian of the  $M$ -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`

**HasAdditiveMeasurementNoise — Model additive measurement noise**  
`false` (default)

Option to model measurement noise as additive, specified as `true` or `false`. When this property is `true`, measurement noise is added to the state vector. Otherwise, noise is incorporated into the measurement function.

Example: `true`

**MaxNumDetections — Maximum number of detections**  
`100` (default) | positive integer

Maximum number of detections the `ggiwphd` filter can take as input, specified as a positive integer.

Example: `50`

Data Types: `single` | `double`

**MaxNumComponents — Maximum number of components**  
`1000` (default) | positive integer

Maximum number of components the `ggiwphd` filter can maintain, specified as a positive integer.

Data Types: `single` | `double`

## Object Functions

<code>append</code>	Append two phd filter objects
<code>correct</code>	Correct phd filter with detections
<code>correctUndetected</code>	Correct phd filter with no detection hypothesis

extractState	Extract target state estimates from the phd filter
labeledDensity	Keep components with a given label ID
likelihood	Log-likelihood of association between detection cells and components in the density
merge	Merge components in the density of phd filter
predict	Predict probability hypothesis density of phd filter
prune	Prune the filter by removing selected components
scale	Scale weights of components in the density
clone	Create duplicate phd filter object

## Examples

### Create ggiwphd Filter with Two 3-D Components

Creating a ggiwphd filter with two 3-D constant velocity components. The initial states of the two components are [0;0;0;0;0] and [1;0;1;0;1;0], respectively. Both these components have position covariance equal to 1 and velocity covariance equal to 100. By default, ggiwphd creates a 3-D extent matrix for each component.

```
states = [zeros(6,1),[1;0;1;0;1;0]];
cov1 = diag([1 100 1 100 1 100]);
covariances = cat(3,cov1,cov1);

phd = ggiwphd(states,covariances,'StateTransitionFcn',@constvel, ...
    'StateTransitionJacobianFcn',@constveljac, ...
    'MeasurementFcn',@cvmeas,'MeasurementJacobianFcn',@cvmeasjac, ...
    'ProcessNoise',eye(3),'HasAdditiveProcessNoise',false, ...
    'PositionIndex',[1;3;5]);
```

Specify information about extent.

```
dofs = [21 30];
scaleMatrix1 = 13*diag([4.7 1.8 1.4].^2);
scaleMatrix2 = 22*diag([1.8 4.7 1.4].^2);
scaleMatrices = cat(3,scaleMatrix1,scaleMatrix2);
phd.DegreesOfFreedom = dofs;
phd.ScaleMatrices = scaleMatrices;
phd.ExtentRotationFcn = @(x,dT)eye(3); % No rotation during prediction
```

Predict the filter 0.1 second ahead.

```
predict(phd,0.1);
```

Specify detections at 0.1 second. The filter receives 10 detections at the current scan.

```
detections = cell(10,1);
rng(2018); % Reproducible results
for i = 1:10
    detections{i} = objectDetection(0.1,randi([0 1]) + randn(3,1));
end
phd.Detections = detections;
```

Select two detection cells and calculate their likelihoods.

```
detectionIDs = false(10,2);
detectionIDs([1 3 5 7 9],1) = true;
```

```
detectionIDs([2 4 6 8 10],2) = true;
lhood = likelihood(phd,detectionIDs)
```

`lhood = 2×2`

1.5575	-0.3183
0.1513	-0.7616

Correct the filter with the two detection cells and associated likelihoods.

```
correct(phd,detectionIDs, exp(lhood)./sum(exp(lhood),1));
phd
```

`phd =`  
ggiwphd with properties:

States: [6x4 double]
StateCovariances: [6x6x4 double]
PositionIndex: [3x1 double]
StateTransitionFcn: @constvel
StateTransitionJacobianFcn: @constveljac
ProcessNoise: [3x3 double]
HasAdditiveProcessNoise: 0
Shapes: [6 6 6 6]
Rates: [2 2 2 2]
GammaForgettingFactors: [1 1 1 1]
DegreesOfFreedom: [25.9870 34.9780 25.9870 34.9780]
ScaleMatrices: [3x3x4 double]
ExtentRotationFcn: @(x,dT)eye(3)
TemporalDecay: 100
Weights: [0.8032 0.1968 0.6090 0.3910]
Labels: [0 0 0 0]
Detections: {1x10 cell}
MeasurementFcn: @cvmeas
MeasurementJacobianFcn: @cvmeasjac
HasAdditiveMeasurementNoise: 1

Merge components in the filter.

```
merge(phd,5);
phd
```

`phd =`  
ggiwphd with properties:

States: [6x2 double]
StateCovariances: [6x6x2 double]
PositionIndex: [3x1 double]
StateTransitionFcn: @constvel
StateTransitionJacobianFcn: @constveljac
ProcessNoise: [3x3 double]
HasAdditiveProcessNoise: 0

```
Shapes: [6 6.0000]
Rates: [2 2]
GammaForgettingFactors: [1 1]

DegreesOfFreedom: [25.9870 34.9780]
ScaleMatrices: [3x3x2 double]
ExtentRotationFcn: @(x,dT)eye(3)
TemporalDecay: 100

Weights: [1.4122 0.5878]
Labels: [0 0]

Detections: {1x10 cell}
MeasurementFcn: @cvmeas
MeasurementJacobianFcn: @cvmeasjac
HasAdditiveMeasurementNoise: 1
```

Extract state estimates and detections.

```
targetStates = extractState(phd,0.5);
tStates = targetStates.State

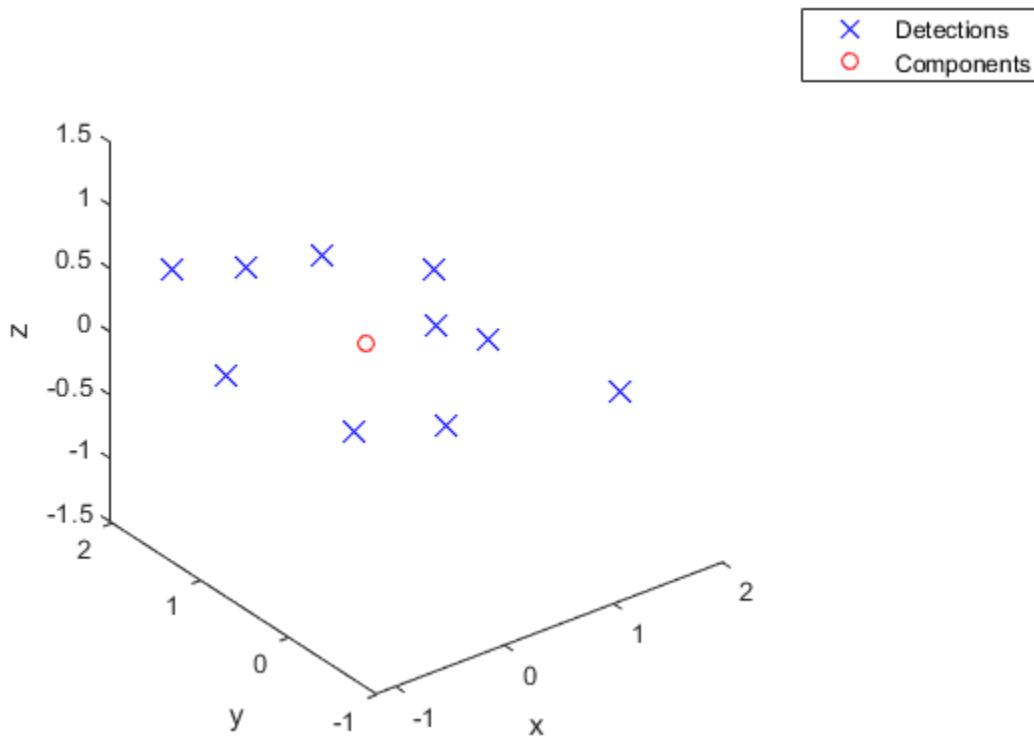
tStates = 6×1

0.1947
0.9733
0.8319
4.1599
-0.0124
-0.0621
```

```
d = [detections{:}];
measurements = [d.Measurement];
```

Visualize the results.

```
figure()
plot3(measurements(1,:),measurements(2,:),measurements(3,:),'x','MarkerSize',10,'MarkerEdgeColor'
hold on;
plot3( tStates(1,:),tStates(3,:),tStates(5,:),'ro');
xlabel('x')
ylabel('y')
zlabel('z')
legend('Detections','Components')
```



## References

- [1] Granstrom, K., and O. Orguner." A PHD filter for tracking multiple extended targets using random matrices." *IEEE Transactions on Signal Processing*. Vol. 60, Number 11, 2012, pp. 5657-5671.
- [2] Granstrom, K., and A. Natale, P. Braca, G. Ludeno, and F. Serafino."Gamma Gaussian inverse Wishart probability hypothesis density for extended target tracking using X-band marine radar data." *IEEE Transactions on Geoscience and Remote Sensing*. Vol. 53, Number 12, 2015, pp. 6617-6631.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

- The code generation configuration must allow recursion to use `merge` method.

## See Also

[gmpdh](#) | [partitionDetections](#) | [trackerPHD](#) | [trackingSensorConfiguration](#)

**Introduced in R2019a**

# append

Append two phd filter objects

## Syntax

```
append(phd1,phd2)
```

## Description

`append(phd1,phd2)` appends the components in `phd2` to the components in `phd1`. The total number of components in the appended filter must not exceed the value specified by the `MaxNumComponents` property of `phd1`.

## Input Arguments

### **phd1 — phd filter**

`ggiwphd` filter object | `gmphd` filter object

`phd` filter, specified as a `ggiwphd` filter object or a `gmphd` filter object.

Example: `phd`

Data Types: object

### **phd2 — phd filter**

`ggiwphd` filter object | `gmphd` filter object

`phd` filter, specified as a `ggiwphd` filter object or a `gmphd` filter object.

Example: `phd`

Data Types: object

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

`ggiwphd` | `gmphd` | `ttrackerPHD`

## Introduced in R2019a

## clone

Create duplicate phd filter object

### Syntax

```
phd2 = clone(phd1)
```

### Description

`phd2 = clone(phd1)` creates a duplicate phd filter, `phd2`, from a phd filter, `phd1`.

### Input Arguments

#### **phd1 — phd filter**

`ggiwphd` filter object | `gmphd` filter object

phd filter, specified as a `ggiwphd` filter object or a `gmphd` filter object.

Example: `phd`

Data Types: object

### Output Arguments

#### **phd2 — phd filter**

`ggiwphd` filter object | `gmphd` filter object

phd filter, returned as a `ggiwphd` filter object or a `gmphd` filter object.

Example: `phd`

Data Types: object

### Extended Capabilities

#### C/C++ Code Generation

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

### See Also

`ggiwphd` | `gmphd` | `trackerPHD`

### Introduced in R2019a

# correct

Correct phd filter with detections

## Syntax

```
correct(phd,detectionIndices,likelihood)
```

## Description

`correct(phd,detectionIndices,likelihood)` corrects phd filter object using detections specified by `detectionIndices` and corresponding detection likelihoods, `likelihood`.

## Input Arguments

### **phd — phd filter**

`ggiwphd` filter object | `gmphd` filter object

`phd` filter, specified as a `ggiwphd` filter object or a `gmphd` filter object.

Example: `phd`

Data Types: object

### **detectionIndices — Indices of detection cells**

`M`-by-`P` logical matrix

Indices of detection cells, specified as an `M`-by-`P` logical matrix. `M` is the number of detections, and `P` is the number of detection cells. In each column, if the value of the  $i$ th element is 1, then the  $i$ th detection belongs to the detection cell specified by this column. On the contrary, if the value of the  $i$ th element is 0, then the  $i$ th detection does not belong to the detection cell specified by this column.

Example: `[1 0 0; 0 1 1; 1 1 0]`

Data Types: logical

### **likelihood — Likelihood of association between detection cells and components**

`N`-by-`P` real-valued matrix

Likelihood of association between detection cells and components in the density, specified as an `N`-by-`P` real-valued matrix. `N` is the number of components in the density of PHD filter, and `P` is the number of detection cells specified by `detectionIndices`. The  $(i,j)$  element of `likelihood` matrix represents the likelihood of association between component  $i$  and detection cell  $j$ . The weight of a component after correction is equal to its original weight multiplied by its likelihood.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

`ggiwphd` | `gmphd` | `ttrackerPHD`

**Introduced in R2019a**

# correctUndetected

Correct phd filter with no detection hypothesis

## Syntax

```
correctUndetect(phd,Pd)
correctUndetect(phd,Pd,PzeroDets)
```

## Description

`correctUndetect(phd,Pd)` corrects the phd filter, `phd`, with the sensor detection probability, `Pd`. If used with `ggiwphd`, the function calculates the probability of generating zero detections using the current Gamma distribution of the filter. If used with `gmphd`, the probability of generating zero detections is assumed to be zero.

`correctUndetect(phd,Pd,PzeroDets)` allows you to specify the conditional probability for generating zero detections using `PzeroDets`.

## Input Arguments

### phd — phd filter

`ggiwphd` filter object | `gmphd` filter object

`phd` filter, specified as a `ggiwphd` filter object or a `gmphd` filter object.

Example: `phd`

Data Types: `object`

### Pd — Sensor's detection probability for each component

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

Sensor's detection probability for each component in the density of the PHD filter, specified as a 1-by- $N$  real-valued row vector, where  $N$  is the number of components.

Example: [0.5 0.6 0.55]

Data Types: `single` | `double`

### PzeroDets — Probability of generating zero detection for each component

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

Probability of generating zero detection for each component in the density of the PHD filter, specified as a 1-by- $N$  real-valued row vector, where  $N$  is the number of components.

Example: [0.1 0.2 0.15]

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[ggiwphd](#) | [gmphd](#) | [trackerPHD](#)

**Introduced in R2019a**

# extractState

Extract target state estimates from the phd filter

## Syntax

```
[States,Indices] = extractState(phd,threshold)
```

## Description

[States,Indices] = extractState(phd,threshold) returns all states of components, States, whose weights are above the threshold given by threshold, and their corresponding indices, Indices, in the phd filter, phd.

## Input Arguments

### phd — phd filter

ggiwphd filter object | gmphd filter object

phd filter, specified as a ggiwphd filter object or a gmphd filter object.

Example: phd

Data Types: object

### threshold — Extraction threshold

real positive scalar

Extraction threshold of component weight, specified as a real positive scalar.

Example: 0.2

Data Types: single | double

## Output Arguments

### States — Extracted states

structure | 1-by-N array of structure

Extracted states, returned as a structure or a 1-by-N array of structure, where N is the number of extracted states. Given the type of the phd filter, each structure contains:

- ggiwphd:

Field	Description
State	State estimate of the target.
StateCovariance	Uncertainty covariance matrix.
Extent	Spatial extent estimate of the tracked object, returned as a d-by-d matrix, where d is the dimension of the object.

MeasurementRate	Expected number of detections from the tracked object.
-----------------	--

- `gmphd`:

Field	Description
State	State estimate of the target.
StateCovariance	Uncertainty covariance matrix.

Data Types: struct

#### Indices — Indices of extracted states

1-by- $N$  vector of nonnegative integers

Indices of extracted states, returned as an 1-by- $N$  vector of nonnegative integers, where  $N$  is the number of extracted states. Each element of the vector is the index of the corresponding extracted state in `States`.

Data Types: double

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`ggiwphd` | `gmphd` | `trackerPHD`

### Introduced in R2019a

# **labeledDensity**

Keep components with a given label ID

## **Syntax**

```
labeledDensity(phd,labelID)
```

## **Description**

`labeledDensity(phd,labelID)` keeps components with the specified `labelID` and removes all other components in the density.

## **Input Arguments**

### **phd — phd filter**

`ggiwphd` filter object | `gmphd` filter object

`phd` filter, specified as a `ggiwphd` filter object or a `gmphd` filter object.

Example: `phd`

Data Types: `object`

### **labelID — label ID of reserved components**

nonnegative integer

label ID of the components to be kept, specified as a nonnegative integer.

Example: `1`

Data Types: `double`

## **Extended Capabilities**

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

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

## **See Also**

`ggiwphd` | `gmphd` | `trackerPHD`

## **Introduced in R2019a**

# likelihood

Log-likelihood of association between detection cells and components in the density

## Syntax

```
lhood = likelihood(phd,detectionIndices)
```

## Description

`lhood = likelihood(phd,detectionIndices)` returns the log-likelihood of association between detection cells specified by `detectionIndices`, and components in the `phd` filter, `phd`.

## Input Arguments

### **phd — phd filter**

`ggiwphd` filter object | `gmphd` filter object

`phd` filter, specified as a `ggiwphd` filter object or a `gmphd` filter object.

Example: `phd`

Data Types: `object`

### **detectionIndices — Indices of detection cells**

`M`-by-`P` logical matrix

Indices of detection cells, specified as an  $M$ -by- $P$  logical matrix.  $M$  is the number of detections, and  $P$  is the number of detection cells. In each column, if the value of the  $i$ th element is 1, then the  $i$ th detection belongs to the detection cell specified by this column. On the contrary, if the value of the  $i$ th element is 0, then the  $i$ th detection does not belong to the detection cell specified by this column.

Example: `[1 0 0; 0 1 1; 1 1 0]`

Data Types: `logical`

## Output Arguments

### **lhood — log-likelihood of association between detection cells and components**

`N`-by-`P` real-valued matrix

Log-likelihood of association between detection cells and components in the density, specified as an  $N$ -by- $P$  real-valued matrix.  $N$  is the number of components in the density of PHD filter, and  $P$  is the number of detection cells specified by `detectionIndices`. The  $(i,j)$  element of `lhood` matrix represents the log-likelihood of association between component  $i$  and detection cell  $j$ .

## Extended Capabilities

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

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

**See Also**

[ggiwphd](#) | [gmphd](#) | [trackerPHD](#)

**Introduced in R2019a**

## merge

Merge components in the density of phd filter

### Syntax

```
merge(phd,mergingThreshold)
```

### Description

`merge(phd,mergingThreshold)` merges components whose Kullback-Leibler difference is below the threshold, `mergingThreshold`.

### Input Arguments

#### **phd — phd filter**

`ggiwphd` filter object | `gmphd` filter object

`phd` filter, specified as a `ggiwphd` filter object or a `gmphd` filter object.

Example: `phd`

Data Types: `object`

#### **mergingThreshold — Threshold for components merging**

real positive scalar

Threshold for components merging, specified as a real positive scalar. If the Kullback-Leibler difference between two components is smaller than the value specified by the `mergingThreshold` argument, then these two components will be merged into one component. The merged weight of the new component is equal to the summation of the weights of the two pre-merged components.

Example: `30`

Data Types: `single` | `double`

### Extended Capabilities

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

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

### See Also

`ggiwphd` | `gmphd` | `trackerPHD`

### Introduced in R2019a

# predict

Predict probability hypothesis density of phd filter

## Syntax

```
predict(phd,dt)
```

## Description

`predict(phd,dt)` predicts the density of the phd filter object, `phd`, forward by time step, `dt`.

## Input Arguments

### **phd — phd filter**

`ggiwphd` filter object | `gmphd` filter object

phd filter, specified as a `ggiwphd` filter object or a `gmphd` filter object.

Example: `phd`

Data Types: object

### **dt — time step of prediction**

real positive scalar

Time step of prediction, specified as a real positive scalar.

Example: `0.1`

Data Types: double

## Extended Capabilities

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

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

## See Also

`ggiwphd` | `gmphd` | `trackerPHD`

## Introduced in R2019a

## prune

Prune the filter by removing selected components

### Syntax

```
prune(phd, pruneIndices)
```

### Description

`prune(phd, pruneIndices)` removes components in `phd` filter object, `phd`, specified by `pruneIndices`.

### Input Arguments

#### **phd — phd filter**

`ggiwphd` filter object | `gmphd` filter object

`phd` filter, specified as a `ggiwphd` filter object or a `gmphd` filter object.

Example: `phd`

Data Types: `object`

#### **pruneIndices — Indices of components to be pruned**

1-by- $N$  logical vector

Indices of components to be pruned, specified as an 1-by- $N$  logical vector, where  $N$  is the number of components in the density. If the  $i$ th element of the vector is 1 instead of 0, then the  $i$ th component will be removed from the density.

Example: [0 1 0 1 0 0]

Data Types: `logical`

### Extended Capabilities

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

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

### See Also

`ggiwphd` | `gmphd` | `ttrackerPHD`

### Introduced in R2019a

# scale

Scale weights of components in the density

## Syntax

```
scale(phd,ScaleFactor)
```

## Description

`scale(phd,ScaleFactor)` scales the weights of components in the density of the phd filter, `phd`, by factor, `ScaleFactor`.

## Input Arguments

### **phd — phd filter**

`ggiwphd` filter object | `gmphd` filter object

`phd` filter, specified as a `ggiwphd` filter object or a `gmphd` filter object.

Example: `phd`

Data Types: `object`

### **ScaleFactor — Scale factor**

`positive scalar` | `1-by-N vector of positive scalars`

Scale factor of components in the density, specified as a positive scalar, or an 1-by-*N* vector of positive scalars, where *N* is the number of components in the density. If the scale factor is specified as a scalar, then the weight of each component is multiplied by this scalar. If the scale factor is specified as a vector, then the weight of each component is multiplied by the corresponding element in the vector.

Example: `[0.9 1.1 0.8]`

Data Types: `single` | `double`

## Extended Capabilities

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

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

## See Also

`ggiwphd` | `gmphd` | `ttrackerPHD`

## Introduced in R2019a

## gmphd

Gaussian mixture (GM) PHD filter

### Description

The `gmphd` object is a filter that implements the probability hypothesis density (PHD) using a mixture of Gaussian components. The filter assumes the target states are Gaussian and represents these states using a mixture of Gaussian components. You can use a `gmphd` filter to track extended objects or point targets. In tracking, a point object returns at most one detection per sensor scan, and an extended object can return multiple detections per sensor scan.

You can directly create a `gmphd` filter. You can also initialize a `gmphd` filter used with `trackerPHD` by specifying the `FilterInitializationFcn` property of `trackingSensorConfiguration`. You can use the provided `initcvgmphd`, `initctgmphd`, `initcagmhd`, and `initctrectgmphd` as initialization functions. Or, you can create your own initialization functions.

### Creation

#### Syntax

```
phd = gmphd  
phd = gmphd(states,stateCovariances)  
phd = gmphd(states,stateCovariances,Name,Value)
```

#### Description

`phd = gmphd` creates a `gmphd` filter with default property values.

`phd = gmphd(states,stateCovariances)` allows you to specify the states and corresponding state covariances of the Gaussian distribution for each component in the density. `states` and `stateCovariances` set the `States` and `StateCovariances` properties of the filter.

`phd = gmphd(states,stateCovariances,Name,Value)` also allows you to specify properties for the filter using one or more name-value pairs. Enclose each property name in quotes.

### Properties

#### States — State of each component in filter

*P*-by-*N* matrix

State of each component in the filter, specified as a *P*-by-*N* matrix, where *P* is the dimension of the state and *N* is the number of components. Each column of the matrix corresponds to the state of one component. The default value for `States` is a 6-by-2 matrix, in which the elements of the first column are all 0, and the elements of the second column are all 1.

Data Types: `single` | `double`

**StateCovariances — State estimate error covariance of each component in filter**  
 $P$ -by- $P$ -by- $N$  array

State estimate error covariance of each component in the filter, specified as a  $P$ -by- $P$ -by- $N$  array, where  $P$  is the dimension of the state and  $N$  is the number of components. Each page ( $P$ -by- $P$  matrix) of the array corresponds to the covariance matrix of each component. The default value for **StateCovariances** is a 6-by-6-by-2 array, in which each page (6-by-6 matrix) is an identity matrix.

Data Types: `single` | `double`

**StateTransitionFcn — State transition function**

`@constvel` (default) | 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 also include noise values.

- If **HasAdditiveProcessNoise** is `true`, specify the function using one of these syntaxes:

```
x(k) = transitionfcn(x(k-1))
x(k) = transitionfcn(x(k-1),dT)
```

where  $x(k)$  is the state estimate at time  $k$ , and  $dT$  is the time step.

- If **HasAdditiveProcessNoise** is `false`, specify the function using one of these syntaxes:

```
x(k) = transitionfcn(x(k-1),w(k-1))
x(k) = transitionfcn(x(k-1),w(k-1),dT)
```

where  $x(k)$  is the state estimate at time  $k$ ,  $w(k)$  is the process noise at time  $k$ , and  $dT$  is the time step.

Example: `@constacc`

Data Types: `function_handle`

**StateTransitionJacobianFcn — Jacobian of state transition function**

`@constveljac` (default) | 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.

- If **HasAdditiveProcessNoise** is `true`, specify the Jacobian function using one of these syntaxes:

```
Jx(k) = statejacobianfcn(x(k))
Jx(k) = statejacobianfcn(x(k),dT)
```

where  $x(k)$  is the state at time  $k$ ,  $dT$  is the time step, and  $Jx(k)$  denotes the Jacobian of the state transition function with respect to the state. The Jacobian is a  $P$ -by- $P$  matrix at time  $k$ , where  $P$  is the dimension of the state.

- If **HasAdditiveProcessNoise** is `false`, specify the Jacobian function using one of these syntaxes:

```
[Jx(k),Jw(k)] = statejacobianfcn(x(k),w(k))
[Jx(k),Jw(k)] = statejacobianfcn(x(k),w(k),dT)
```

where  $w(k)$  is a  $Q$ -element vector of the process noise at time  $k$ . Unlike the case of additive process noise, the process noise vector in the non-additive noise case doesn't need to have the same dimensions as the state vector.

$Jw(k)$  denotes the  $P$ -by- $Q$  Jacobian of the predicted state with respect to the process noise elements, where  $P$  is the dimension of the state.

If not specified, the Jacobians are computed by numerical differencing at each call of the `predict` function. This computation can increase the processing time and numerical inaccuracy.

Example: `@constaccjac`

Data Types: `function_handle`

**ProcessNoise — Process noise covariance**

`eye(3)` (default) | positive real-valued scalar | positive definite real-valued matrix

Process noise covariance:

- When `HasAdditiveProcessNoise` is `true`, specify the process noise covariance as a real-valued scalar or a positive definite  $P$ -by- $P$  matrix.  $P$  is the dimension of the state vector. When specified as a scalar, the matrix is a multiple of the  $P$ -by- $P$  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` object function.

Example: `[1.0 0.05; 0.05 2]`

**HasAdditiveProcessNoise — Model additive process noise**

`false` (default) | `true`

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.

Example: `true`

**HasExtent — Indicate if components have extent**

`false` (default) | `true`

Indicate if components have extent, specified as `true` or `false`. Set this property to `true` if the filter is intended to track extended objects. An extended object can generate more than one measurement per sensor scan. Set this property to `false` if the filter is only intended to track point targets.

Example: `true`

**MeasurementOrigin — Origination of measurements from extended objects**

`'center'` (default) | `'extent'`

Origination of measurements from extended objects, specified as:

- `'center'` — The filter assumes the measurements originate from the mean state of a target. This approach is applicable when the state does not model the extent of the target even though the target may generate more than one measurement.
- `'extent'` — The filter assumes measurements are not centered at the mean state of a target. For computational efficiency, the expected measurement is often calculated as a function of the reported measurements specified by the measurement model function.

Note that the function setups of `MeasurementFcn` and `MeasurementJacobianFcn` are different for 'center' and 'extent' options. See the descriptions of `MeasurementFcn` and `MeasurementJacobianFcn` for more details.

### **Dependencies**

To enable this property, set the `HasExtent` property to 'true'.

Data Types: double

### **Labels — Label of each component in mixture**

[0 0] (default) | 1-by- $N$  row vector of nonnegative integer

Label of each component in the mixture, specified as a 1-by- $N$  row vector of nonnegative integers.  $N$  is the number of components in the mixture. Each component can only have one label, but multiple components can share the same label.

Example: [1 2 3]

Data Types: single | double

### **Weights — Weight of each component in mixture**

[1 1] (default) | 1-by- $N$  row vector of positive real value

Weight of each component in the mixture, specified as a 1-by- $N$  row vector of positive real values.  $N$  is the number of components in the mixture. The weight of each component is given in the same order as the `Labels` property.

Example: [1.1 0.82 1.1]

Data Types: single | double

### **Detections — Detections**

$D$ -element cell array of `objectDetection` objects

Detections, specified as a  $D$ -element cell array of `objectDetection` objects. You can create detections directly, or you can obtain detections from the outputs of sensor objects, such as `radarSensor`, `monostaticRadarSensor`, `irSensor`, and `sonarSensor`.

Data Types: single | double

### **MeasurementFcn — Measurement model function**

@cvmeas (default) | function handle

Measurement model function, specified as a function handle. This function specifies the transition from state to measurement. Depending on the `HasExtent` and `MeasurementOrigin` properties, the measurement model function needs to be specified differently:

- 1 `HasExtent` is false, or `HasExtent` is true and `MeasurementOrigin` is 'center'. In these two cases,

- If `HasAdditiveMeasurementNoise` is true, specify the function using one of these syntaxes:

```
z = measurementfcn(x)
z = measurementfcn(x,parameters)
```

where the  $P$ -by- $N$  matrix  $x$  is the estimated Gaussian states at time  $k$  and  $x(:, i)$  represents the  $i$ th state component in the mixture. The  $M$ -by- $N$  matrix  $z$  is the corresponding

measurement, and  $z(:, i)$  represents the measurement resulting from the  $i$ th component. Parameters are `MeasurementParameters` provided in the `objectDetections` set in the `Detections` property.

- If `HasAdditiveMeasurementNoise` is `false`, specify the function using one of these syntaxes:

```
z = measurementfcn(x, v)
z = measurementfcn(x, v, parameters)
```

where  $v$  is an  $R$ -dimensional measurement noise vector.

- 2 `HasExtent` is `true` and `MeasurementOrigin` is '`extent`'. In this case, the expected measurements originate from the extent of the target and rely on the actual distribution of the detections:

- If `HasAdditiveMeasurementNoise` is `true`, specify the function using:

```
z = measurementfcn(x, detections)
```

where the  $P$ -by- $N$  matrix  $x$  is the estimated Gaussian states at time  $k$  and  $x(:, i)$  represents the  $i$ th state component in the mixture. `detections` is a cell array of `objectDetection` objects, and  $z$  is the expected measurement. Note that  $z(:, i, j)$  must return the expected measurement based on the  $i$ th state component and the  $j$ th `objectDetection` in `detections`.

- If `HasAdditiveMeasurementNoise` is `false`, specify the function using:

```
z = measurementfcn(x, v, detections)
```

where  $v$  is an  $R$ -dimensional measurement noise vector.

<code>HasExtent</code>	<code>MeasurementOrigin</code>	<b>Measurement Function</b>		<b>Note</b>					
<code>false</code>	NA								
<code>true</code>	'center'	<table border="1"> <tr> <th><code>HasAdditiveMeasurementNoise</code></th> <th><code>Syntaxes</code></th> </tr> <tr> <td><code>true</code></td> <td> <math>z = \text{measurementfcn}(x)</math>  <math>z = \text{measurementfcn}(x, para)</math> </td> </tr> <tr> <td><code>false</code></td> <td> <math>z = \text{measurementfcn}(x, v)</math>  <math>z = \text{measurementfcn}(x, v, para)</math> </td> </tr> </table>	<code>HasAdditiveMeasurementNoise</code>	<code>Syntaxes</code>	<code>true</code>	$z = \text{measurementfcn}(x)$ $z = \text{measurementfcn}(x, para)$	<code>false</code>	$z = \text{measurementfcn}(x, v)$ $z = \text{measurementfcn}(x, v, para)$	$x(:, i)$ represents the $i$ th state component in the mixture. $z(:, i)$ represents the measurement resulting from the $i$ th component.
<code>HasAdditiveMeasurementNoise</code>	<code>Syntaxes</code>								
<code>true</code>	$z = \text{measurementfcn}(x)$ $z = \text{measurementfcn}(x, para)$								
<code>false</code>	$z = \text{measurementfcn}(x, v)$ $z = \text{measurementfcn}(x, v, para)$								

true	'extent'	<b>HasAdditiveMeasurementNoise</b>	<b>Syntaxes</b>	$x(:,i)$ represents the $i$ th state component in the mixture. $z(:,i,j)$ must return the expected measurement based on the $i$ th state component and the $j$ th object detection in detections.
		true	$z = \text{measurementfcn}(x, \text{detections})$	
		false	$z = \text{measurementfcn}(x, v, \text{detections})$	

Data Types: function\_handle

**MeasurementJacobianFcn — Jacobian of measurement function**  
`@cvmeasjac` (default) | function handle

Jacobian of the measurement function, specified as a function handle. Depending on the `HasExtent` and `MeasurementOrigin` properties, the measurement Jacobian function needs to be specified differently:

**1** `HasExtent` is `false`, or `HasExtent` is `true` and `MeasurementOrigin` is '`center`'. In these two cases:

- If `HasAdditiveMeasurementNoise` is `true`, specify the Jacobian function using one of these syntaxes:

```
Jmx = measjacobianfcn(x)
```

```
Jmx = measjacobianfcn(x, parameters)
```

where the  $P$ -element vector  $x$  is one state component at time  $k$  and  $Jmx$  is the  $M$ -by- $P$  Jacobian of the measurement function with respect to the state.  $M$  is the dimension of the measurement. Parameters are `MeasurementParameters` provided in the `objectDetections` set in the `Detections` property.

- If `HasAdditiveMeasurementNoise` is `false`, specify the Jacobian function using one of these syntaxes:

```
[Jmx, Jmv] = measjacobianfcn(x, v)
```

```
[Jmx, Jmv] = measjacobianfcn(x, v, parameters)
```

where  $v$  is an  $R$ -dimensional measurement noise vector, and  $J_{mv}$  is the  $M$ -by- $R$  Jacobian of the measurement function with respect to the measurement noise.

- 2 HasExtent is true and MeasurementOrigin is 'extent'. In this case, the expected measurements originate from the extent of the target and rely on the actual distribution of the detections. The measurement Jacobian function must support one of these two syntaxes:

- If HasAdditiveMeasurementNoise is true, specify the Jacobian function using:

```
Jmx = measjacobianfcn(x,detections)
```

where  $x$  is one state estimate component at time  $k$ . `detections` is a set of detections defined as a cell array of `objectDetection` objects.  $J_{mx}$  denotes the  $M$ -by- $P$ -by- $D$  Jacobian of the measurement function with respect to the state.  $M$  is the dimension of the measurement,  $P$  is the dimension of the state, and  $D$  is the number of `objectDetection` objects in `detections`.

- If HasAdditiveMeasurementNoise is false, specify the Jacobian function using:

```
[Jmx,Jmv] = measjacobianfcn(x,v,detections)
```

where  $v$  is an  $R$ -dimensional measurement noise vector, and  $J_{mv}$  is the  $M$ -by- $R$ -by- $D$  Jacobian of the measurement function with respect to the measurement noise.

Note that  $J_{mx}(:,:,j)$  must define the state Jacobian corresponding to the  $j$ th `objectDetection` in `detections`.  $J_{mv}(:,:,j)$  defines the measurement noise Jacobian corresponding to the  $j$ th `objectDetection` in `detections`.

HasExtent	MeasurementOrigin	Measurement Jacobian Function		Note
		HasAdditiveMeasurementNoise	Syntaxes	
false	NA			
true	'center'	true	$J_{mx} =$ <code>measjacobianfcn(x)</code>  $J_{mx} =$ <code>measjacobianfcn(x, para)</code>	$x$ is only one Gaussian component in the mixture.
		false	$[J_{mx}, J_{mv}] =$ <code>measjacobianfcn(x, v)</code>  $[J_{mx}, J_{mv}] =$ <code>measjacobianfcn(x, v, para)</code>	

<code>true</code>	<code>'extent'</code>	<b>HasAdditiveMeasurementNoise</b>	<b>Syntaxes</b>	
		<code>true</code>	$z = \text{measurementfcn}(x, \text{detections})$	$J_{\text{mx}}(:, :, j)$ defines the state Jacobian corresponding to the $j$ th object detection in detections.
		<code>false</code>	$z = \text{measurementfcn}(x, v, \text{detections})$	$J_{\text{mv}}(:, :, j)$ defines the measurement noise Jacobian corresponding to the $j$ th object detection in detections.

Data Types: function\_handle

**HasAdditiveMeasurementNoise — Model additive measurement noise**  
`false` (default) | `true`

Option to model measurement noise as additive, specified as `true` or `false`. When this property is `true`, measurement noise is added to the state vector. Otherwise, noise is incorporated into the measurement function.

Example: `true`

**MaxNumDetections — Maximum number of detections**  
`1000` (default) | positive integer

Maximum number of detections the gmpfd filter can take as input, specified as a positive integer.

Example: `50`

Data Types: `single` | `double`

**MaxNumComponents — Maximum number of components**  
`1000` (default) | positive integer

Maximum number of components the gmpfd filter can maintain, specified as a positive integer.

Data Types: `single` | `double`

## Object Functions

<code>predict</code>	Predict probability hypothesis density of phd filter
<code>correctUndetected</code>	Correct phd filter with no detection hypothesis
<code>correct</code>	Correct phd filter with detections
<code>likelihood</code>	Log-likelihood of association between detection cells and components in the density
<code>append</code>	Append two phd filter objects
<code>merge</code>	Merge components in the density of phd filter
<code>scale</code>	Scale weights of components in the density
<code>prune</code>	Prune the filter by removing selected components
<code>labeledDensity</code>	Keep components with a given label ID
<code>extractState</code>	Extract target state estimates from the phd filter
<code>clone</code>	Create duplicate phd filter object

## Examples

### Run gmphd Filter for Point Objects

Create a filter with two 3-D constant velocity components. The initial state of one component is [0;0;0;0;0]. The initial state of the other component is [1;0;1;0;1;0]. Each component is initialized with position covariance equal to 1 and velocity covariance equal to 100.

```
states = [zeros(6,1) [1;0;1;0;1;0]];
cov1 = diag([1 100 1 100 1 100]);
covariances = cat(3,cov1,cov1);
phd = gmphd(states, covariances, 'StateTransitionFcn', @constvel, ...
    'StateTransitionJacobianFcn', @constveljac, ...
    'MeasurementFcn', @cvmeas, ...
    'MeasurementJacobianFcn', @cvmeasjac, ...
    'ProcessNoise', eye(3), ...
    'HasAdditiveProcessNoise', false);
```

Predict the filter 0.1 time step ahead.

```
predict(phd,0.1);
```

Define three detections using `objectDetection`.

```
rng(2019);
detections = cell(3,1);
detections{1} = objectDetection(0,[1;1;1] + randn(3,1));
detections{2} = objectDetection(0,[0;0;0] + randn(3,1));
detections{3} = objectDetection(0,[4;5;5] + randn(3,1));
phd.Detections = detections;
```

Calculate the likelihood of each detection. For a point-target filter, the partition of detections is unnecessary, and each detection occupies a cell. Therefore, `detectionIndices` is an identity matrix. The resulting likelihood of detection 1 and 2 is higher than that of detection 3 because they are closer to the components.

```
detectionIndices = logical(eye(3));
logLikelihood = likelihood(phd,detectionIndices)
```

```
logLikelihood = 2x3
-5.2485    -4.7774   -22.8899
-4.5171    -5.0008   -17.3973
```

Correct the filter with the scaled likelihood.

```
lhood = exp(logLikelihood);
lhood = lhood./sum(lhood,2);
correct(phd,detectionIndices,lhood);
```

Merge the components with a merging threshold equal to 1.

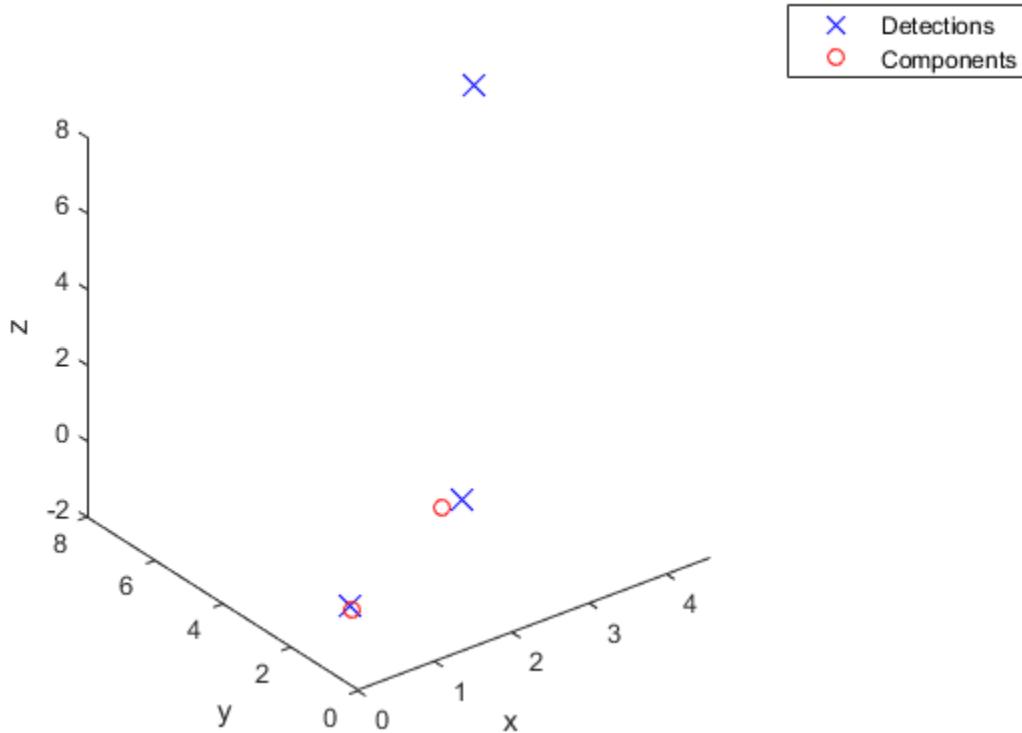
```
merge(phd,1);
```

Extract state estimates with an extract threshold equal to 0.5.

```
minWeight = 0.5;
targetStates = extractState(phd,minWeight);
[ts1,ts2]= targetStates.State;
```

Visualize the results.

```
% Extract the measurements.
d = [detections{:}];
measurements = [d.Measurement];
% Plot the measurements and estimates.
figure()
plot3(measurements(1,:),measurements(2,:),measurements(3,:),'x','MarkerSize',10,'MarkerEdgeColor','k');
hold on;
plot3(ts1(1),ts1(3),ts1(5),'ro');
hold on;
plot3(ts2(1),ts2(3),ts2(5),'ro');
xlabel('x')
ylabel('y')
zlabel('z')
hold on;
legend('Detections','Components')
```



## References

- [1] Vo, B. -T, and W. K. Ma. "The Gaussian mixture Probability Hypothesis Density Filter." *IEEE Transactions on Signal Processing*, Vol. 54, No. 11, pp. 4091-4104, 2006.
- [2] Granstrom, Karl, Christian Lundquist, and Omut Orguner. "Extended target tracking using a Gaussian-mixture PHD filter." *IEEE Transactions on Aerospace and Electronic Systems* 48, no. 4 (2012): 3268-3286.

## Extended Capabilities

### C/C++ Code Generation

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

Usage notes and limitations:

- The code generation configuration must allow recursion to use the `merge` method.

### See Also

`ggiwphd` | `initcagmphd` | `initctgmphd` | `initctrectgmphd` | `initcvgmphd` |  
`partitionDetections` | `trackerPHD` | `trackingSensorConfiguration`

**Introduced in R2019b**

## pose

Current orientation and position estimate for `insfilterNonholonomic`

### Syntax

```
[position,orientation,velocity] = pose(FUSE)
[position,orientation,velocity] = pose(FUSE,format)
```

### Description

`[position,orientation,velocity] = pose(FUSE)` returns the current estimate of the pose.

`[position,orientation,velocity] = pose(FUSE,format)` returns the current estimate of the pose with orientation in the specified orientation format.

### Input Arguments

**FUSE — NHConstrainedIMUGPSFuser object**  
object

`insfilterNonholonomic`, specified as an object.

**format — Output orientation format**  
`'quaternion'` (default) | `'rotmat'`

Output orientation format, specified as either `'quaternion'` for a quaternion or `'rotmat'` for a rotation matrix.

Data Types: `char` | `string`

### Output Arguments

**position — Position estimate in the local NED coordinate system (m)**  
3-element row vector

Position estimate in the local NED coordinate system in meters, returned as a 3-element row vector.

Data Types: `single` | `double`

**orientation — Orientation estimate in the local NED coordinate system**  
`quaternion` (default) | 3-by-3 rotation matrix

Orientation estimate in the local NED coordinate system, specified as a scalar quaternion or 3-by-3 rotation matrix. The quaternion or rotation matrix represents a frame rotation from the local NED reference frame to the body reference frame.

Data Types: `single` | `double` | `quaternion`

**velocity — Velocity estimate in local NED coordinate system (m/s)**  
3-element row vector

---

Velocity estimate in the local NED coordinate system in m/s, returned as a 3-element row vector.

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`insfilter` | `insfilterNonholonomic`

**Introduced in R2018b**

# **predict**

Update states using accelerometer and gyroscope data for `insfilterNonholonomic`

## **Syntax**

```
predict(FUSE,accelReadings,gyroReadings)
```

## **Description**

`predict(FUSE,accelReadings,gyroReadings)` fuses accelerometer and gyroscope data to update the state estimate.

## **Input Arguments**

**FUSE — `insfilterNonholonomic` object**  
object

`insfilterNonholonomic`, specified as an object.

**accelReadings — Accelerometer readings in local sensor body coordinate system (m/s<sup>2</sup>)**  
3-element row vector

Accelerometer readings in m/s<sup>2</sup>, specified as a 3-element row vector.

Data Types: `single` | `double`

**gyroReadings — Gyroscope readings in local sensor body coordinate system (rad/s)**  
3-element row vector

Gyroscope readings in rad/s, specified as a 3-element row vector.

Data Types: `single` | `double`

## **Extended Capabilities**

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

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

### **See Also**

`insfilter` | `insfilterNonholonomic`

**Introduced in R2018b**

# reset

Reset internal states for `insfilterNonholonomic`

## Syntax

```
reset(FUSE)
```

## Description

`reset(FUSE)` resets the State, StateCovariance, and internal integrators to their default values.

## Input Arguments

### FUSE — `insfilterNonholonomic` object

object

`insfilterNonholonomic`, specified as an object.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

`insfilter` | `insfilterNonholonomic`

## Introduced in R2018b

## stateinfo

Display state vector information for `insfilterNonholonomic`

### Syntax

```
stateinfo(FUSE)
info = stateinfo(FUSE)
```

### Description

`stateinfo(FUSE)` displays the meaning of each index of the State property and the associated units.

`info = stateinfo(FUSE)` returns a structure with fields containing descriptions of the elements of the state vector of the filter, `FUSE`.

### Examples

#### State information of `insfilterNonholonomic`

Create an `insfilterNonholonomic` object.

```
filter = insfilterErrorState;
```

Display the state information of the created filter.

```
stateinfo(filter)
```

States	Units	Index
Orientation (quaternion parts)		1:4
Position (NAV)	m	5:7
Velocity (NAV)	m/s	8:10
Gyroscope Bias (XYZ)	rad/s	11:13
Accelerometer Bias (XYZ)	m/s <sup>2</sup>	14:16
Visual Odometry Scale		17

Output the state information of the filter as a structure.

```
info = stateinfo(filter)

info = struct with fields:
    Orientation: [1 2 3 4]
    Position: [5 6 7]
    Velocity: [8 9 10]
    GyroscopeBias: [11 12 13]
    AccelerometerBias: [14 15 16]
    VisualOdometryScale: 17
```

## Input Arguments

**FUSE — insfilterNonholonomic object**  
object

`insfilterNonholonomic`, specified as an object.

## Output Arguments

**info — State information**  
structure

State information, returned as a structure with fields containing descriptions of the elements of the state vector of the filter. The values of each field are the corresponding indices of the state vector.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

`insfilter` | `insfilterNonholonomic`

**Introduced in R2018b**

# insfilterNonholonomic

Estimate pose with nonholonomic constraints

## Description

The `insfilterNonholonomic` object implements sensor fusion of inertial measurement unit (IMU) and GPS data to estimate pose in the NED (or ENU) reference frame. IMU data is derived from gyroscope and accelerometer data. The filter uses a 16-element state vector to track the orientation quaternion, velocity, position, and IMU sensor biases. The `insfilterNonholonomic` object uses an extended Kalman filter to estimate these quantities.

## Creation

### Syntax

```
filter = insfilterNonholonomic  
filter = insfilterNonholonomic('ReferenceFrame',RF)  
filter = insfilterNonholonomic( ___,Name,Value)
```

### Description

`filter = insfilterNonholonomic` creates an `insfilterErrorState` object with default property values.

`filter = insfilterNonholonomic('ReferenceFrame',RF)` allows you to specify the reference frame, RF, of the filter. Specify RF as 'NED' (North-East-Down) or 'ENU' (East-North-Up). The default value is 'NED'.

`filter = insfilterNonholonomic( ___,Name,Value)` also allows you set properties of the created filter using one or more name-value pairs. Enclose each property name in single quotes.

## Properties

### IMUSampleRate — Sample rate of the IMU (Hz)

`[100]` (default) | positive scalar

Sample rate of the IMU in Hz, specified as a positive scalar.

Data Types: `single` | `double`

### ReferenceLocation — Reference location (deg, deg, meters)

`[0 0 0]` (default) | 3-element positive row vector

Reference location, specified as a 3-element row vector in geodetic coordinates (latitude, longitude, and altitude). Altitude is the height above the reference ellipsoid model, WGS84. The reference location units are [degrees degrees meters].

Data Types: `single` | `double`

**DecimationFactor — Decimation factor for kinematic constraint correction**

2 (default) | positive integer scalar

Decimation factor for kinematic constraint correction, specified as a positive integer scalar.

Data Types: single | double

**GyroscopeNoise — Multiplicative process noise variance from gyroscope (rad/s)<sup>2</sup>**

[4.8e-6 4.8e-6 4.8e-6] (default) | scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope in (rad/s)<sup>2</sup>, specified as a scalar or 3-element row vector of positive real finite numbers.

- If GyroscopeNoise is specified as a row vector, the elements correspond to the noise in the x, y, and z axes of the gyroscope, respectively.
- If GyroscopeNoise is specified as a scalar, the single element is applied to the x, y, and z axes of the gyroscope.

Data Types: single | double

**GyroscopeBiasNoise — Multiplicative process noise variance from gyroscope bias (rad/s)<sup>2</sup>**

[4e-14 4e-14 4e-14] (default) | scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope bias in (rad/s)<sup>2</sup>, specified as a scalar or 3-element row vector of positive real finite numbers. Gyroscope bias is modeled as a lowpass filtered white noise process.

- If GyroscopeBiasNoise is specified as a row vector, the elements correspond to the noise in the x, y, and z axes of the gyroscope, respectively.
- If GyroscopeBiasNoise is specified as a scalar, the single element is applied to the x, y, and z axes of the gyroscope.

Data Types: single | double

**GyroscopeBiasDecayFactor — Decay factor for gyroscope bias**

0 .999 (default) | scalar in the range [0,1]

Decay factor for gyroscope bias, specified as a scalar in the range [0,1]. A decay factor of 0 models gyroscope bias as a white noise process. A decay factor of 1 models the gyroscope bias as a random walk process.

Data Types: single | double

**AccelerometerNoise — Multiplicative process noise variance from accelerometer (m/s<sup>2</sup>)<sup>2</sup>**

[4.8e-2 4.8e-2 4.8e-2] (default) | scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer in (m/s<sup>2</sup>)<sup>2</sup>, specified as a scalar or 3-element row vector of positive real finite numbers.

- If AccelerometerNoise is specified as a row vector, the elements correspond to the noise in the x, y, and z axes of the accelerometer, respectively.
- If AccelerometerNoise is specified as a scalar, the single element is applied to each axis.

Data Types: single | double

**AccelerometerBiasNoise — Multiplicative process noise variance from accelerometer bias (m/s<sup>2</sup>)<sup>2</sup>**

[4e-14 4e-14 4e-14] (default) | positive scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer bias in (m/s<sup>2</sup>)<sup>2</sup>, specified as a scalar or 3-element row vector of positive real numbers. Accelerometer bias is modeled as a lowpass filtered white noise process.

- If `AccelerometerBiasNoise` is specified as a row vector, the elements correspond to the noise in the x, y, and z axes of the accelerometer, respectively.
- If `AccelerometerBiasNoise` is specified as a scalar, the single element is applied to each axis.

**AccelerometerBiasDecayFactor — Decay factor for accelerometer bias**

0.9999 (default) | scalar in the range [0,1]

Decay factor for accelerometer bias, specified as a scalar in the range [0,1]. A decay factor of 0 models accelerometer bias as a white noise process. A decay factor of 1 models the accelerometer bias as a random walk process.

Data Types: `single` | `double`**State — State vector of extended Kalman filter**

[1;zeros(15,1)] | 16-element column vector

State vector of the extended Kalman filter. The state values represent:

State	Units	Index
Orientation (quaternion parts)	N/A	1:4
Gyroscope Bias (XYZ)	rad/s	5:7
Position (NED or ENU)	m	8:10
Velocity (NED or ENU)	m/s	11:13
Accelerometer Bias (XYZ)	m/s <sup>2</sup>	14:16

Data Types: `single` | `double`**StateCovariance — State error covariance for extended Kalman filter**

eye(16) (default) | 16-by-16 matrix

State error covariance for the extended Kalman filter, specified as a 16-by-16-element matrix, or real numbers.

Data Types: `single` | `double`**ZeroVelocityConstraintNoise — Velocity constraints noise (m/s)<sup>2</sup>**

1e-2 (default) | nonnegative scalar

Velocity constraints noise in (m/s)<sup>2</sup>, specified as a nonnegative scalar.

Data Types: `single` | `double`**Object Functions**correct      Correct states using direct state measurements for `insfilterNonholonomic`

residual	Residuals and residual covariances from direct state measurements for insfilterNonholonomic
fusegps	Correct states using GPS data for insfilterNonholonomic
residualgps	Residuals and residual covariance from GPS measurements for insfilterNonholonomic
pose	Current orientation and position estimate for insfilterNonholonomic
predict	Update states using accelerometer and gyroscope data for insfilterNonholonomic
reset	Reset internal states for insfilterNonholonomic
stateinfo	Display state vector information for insfilterNonholonomic
copy	Create copy of insfilterNonholonomic

## Examples

### Estimate Pose of Ground Vehicle

This example shows how to estimate the pose of a ground vehicle from logged IMU and GPS sensor measurements and ground truth orientation and position.

Load the logged data of a ground vehicle following a circular trajectory.

```
load('loggedGroundVehicleCircle.mat','imuFs','localOrigin','initialState','initialStateCovariance',...
    'gyroData','gpsFs','gpsLLA','Rpos','gpsVel','Rvel','trueOrient','truePos');
```

Initialize the `insfilterNonholonomic` object.

```
filt = insfilterNonholonomic;
filt.IMUSampleRate = imuFs;
filt.ReferenceLocation = localOrigin;
filt.State = initialState;
filt.StateCovariance = initialStateCovariance;

imuSamplesPerGPS = imuFs/gpsFs;
```

Log data for final metric computation. Use the `predict` object function to estimate filter state based on accelerometer and gyroscope data. Then correct the filter state according to GPS data.

```
numIMUSamples = size(accelData,1);
estOrient = quaternion.ones(numIMUSamples,1);
estPos = zeros(numIMUSamples,3);

gpsIdx = 1;

for idx = 1:numIMUSamples
    predict(filt,accelData(idx,:),gyroData(idx,:)); %Predict filter state

    if (mod(idx,imuSamplesPerGPS) == 0) %Correct filter state
        fusegps(filt,gpsLLA(gpsIdx,:),Rpos,gpsVel(gpsIdx,:),Rvel);
        gpsIdx = gpsIdx + 1;
    end

    [estPos(idx,:),estOrient(idx,:)] = pose(filt); %Log estimated pose
end
```

Calculate and display RMS errors.

```
posd = estPos - truePos;
quatd = rad2deg(dist(estOrient,trueOrient));
```

```
msep = sqrt(mean(posd.^2));

fprintf('Position RMS Error\n\tX: %.2f, Y: %.2f, Z: %.2f (meters)\n\n',msep(1),msep(2),msep(3));
Position RMS Error
    X: 0.15, Y: 0.11, Z: 0.01 (meters)

fprintf('Quaternion Distance RMS Error\n\t%.2f (degrees)\n\n',sqrt(mean(quatd.^2)));
Quaternion Distance RMS Error
    0.26 (degrees)
```

## Algorithms

*Note: The following algorithm only applies to an NED reference frame.*

`insfilterNonholonomic` uses a 16-axis error state Kalman filter structure to estimate pose in the NED reference frame. The state is defined as:

$$x = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ gyrobias_X \\ gyrobias_Y \\ gyrobias_Z \\ position_N \\ position_E \\ position_D \\ v_N \\ v_E \\ v_D \\ accelbias_X \\ accelbias_Y \\ accelbias_Z \end{bmatrix}$$

where

- $q_0, q_1, q_2, q_3$  -- Parts of orientation quaternion. The orientation quaternion represents a frame rotation from the platform's current orientation to the local NED coordinate system.
- $gyrobias_X, gyrobias_Y, gyrobias_Z$  -- Bias in the gyroscope reading.
- $position_N, position_E, position_D$  -- Position of the platform in the local NED coordinate system.
- $v_N, v_E, v_D$  -- Velocity of the platform in the local NED coordinate system.
- $accelbias_X, accelbias_Y, accelbias_Z$  -- Bias in the accelerometer reading.

Given the conventional formulation of the state transition function,

$$x_k|k-1 = f(\hat{x}_{k-1}|k-1)$$

the predicted state estimate is:

$x_{k|k-1} =$

$$\begin{aligned}
 & q_0 + \Delta t * q_1(gyrobias_X/2 - gyro_X/2) + \Delta t * q_2 * (gyrobias_Y/2 - gyro_Y/2) + \Delta t * q_3 * (gyrobias_Z/2 - gyro_Z/2) \\
 & q_1 - \Delta t * q_0(gyrobias_X/2 - gyro_X/2) + \Delta t * q_3 * (gyrobias_Y/2 - gyro_Y/2) - \Delta t * q_2 * (gyrobias_Z/2 - gyro_Z/2) \\
 & q_2 - \Delta t * q_3(gyrobias_X/2 - gyro_X/2) - \Delta t * q_0 * (gyrobias_Y/2 - gyro_Y/2) + \Delta t * q_1 * (gyrobias_Z/2 - gyro_Z/2) \\
 & q_3 + \Delta t * q_2(gyrobias_X/2 - gyro_X/2) - \Delta t * q_1 * (gyrobias_Y/2 - gyro_Y/2) - \Delta t * q_0 * (gyrobias_Z/2 - gyro_Z/2) \\
 & \quad - gyrobias_X * (\Delta t * \lambda_{gyro} - 1) \\
 & \quad - gyrobias_Y * (\Delta t * \lambda_{gyro} - 1) \\
 & \quad - gyrobias_Z * (\Delta t * \lambda_{gyro} - 1) \\
 & \quad position_N + \Delta t * v_N \\
 & \quad position_E + \Delta t * v_E \\
 & \quad position_D + \Delta t * v_D \\
 v_N + \Delta t * & \left\{ \begin{array}{l} q_0 * (q_0 * (accelbias_X - accel_X) - q_3 * (accelbias_Y - accel_Y) + q_2 * (accelbias_Z - accel_Z)) - g_N + \\ q_2 * (q_1 * (accelbias_Y - accel_Y) - q_2 * (accelbias_X - accel_X) + q_0 * (accelbias_Z - accel_Z)) + \\ q_1 * (q_1 * (accelbias_X - accel_X) + q_2 * (accelbias_Y - accel_Y) + q_3 * (accelbias_Z - accel_Z)) - \\ q_3 * (q_3 * (accelbias_X - accel_X) + q_0 * (accelbias_Y - accel_Y) - q_1 * (accelbias_Z - accel_Z)) \end{array} \right\} \\
 v_E + \Delta t * & \left\{ \begin{array}{l} q_0 * (q_3 * (accelbias_X - accel_X) + q_0 * (accelbias_Y - accel_Y) - q_1 * (accelbias_Z - accel_Z)) - g_E - \\ q_1 * (q_1 * (accelbias_Y - accel_Y) - q_2 * (accelbias_X - accel_X) + q_0 * (accelbias_Z - accel_Z)) + \\ q_2 * (q_1 * (accelbias_X - accel_X) + q_2 * (accelbias_Y - accel_Y) + q_3 * (accelbias_Z - accel_Z)) + \\ q_3 * (q_3 * (accelbias_X - accel_X) + q_0 * (accelbias_Y - accel_Y) + q_1 * (accelbias_Z - accel_Z)) \end{array} \right\}
 \end{aligned}$$

where

- $\Delta t$  -- IMU sample time.
- $g_N, g_E, g_D$  -- Constant gravity vector in the NED frame.
- $accel_X, accel_Y, accel_Z$  -- Acceleration vector in the body frame.
- $\lambda_{accel}$  -- Accelerometer bias decay factor.
- $\lambda_{gyro}$  -- Gyroscope bias decay factor.

## References

- [1] Munguía, R. "A GPS-Aided Inertial Navigation System in Direct Configuration." *Journal of applied research and technology*. Vol. 12, Number 4, 2014, pp. 803 - 814.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[insfilterAsync](#) | [insfilterErrorState](#) | [insfilterMARG](#)

### Topics

"Estimate Position and Orientation of a Ground Vehicle"

### Introduced in R2018b

# accelparams

Accelerometer sensor parameters

## Description

The `accelparams` class creates an accelerometer sensor parameters object. You can use this object to model an accelerometer when simulating an IMU with `imuSensor`.

## Creation

### Syntax

```
params = accelparams  
params = accelparams(Name,Value)
```

### Description

`params = accelparams` returns an ideal accelerometer sensor parameters object with default values.

`params = accelparams(Name,Value)` configures an accelerometer sensor parameters 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.

## Properties

### MeasurementRange — Maximum sensor reading (m/s<sup>2</sup>)

`inf` (default) | real positive scalar

Maximum sensor reading in m/s<sup>2</sup>, specified as a real positive scalar.

Data Types: `single` | `double`

### Resolution — Resolution of sensor measurements ((m/s<sup>2</sup>)/LSB)

`0` (default) | real nonnegative scalar

Resolution of sensor measurements in (m/s<sup>2</sup>)/LSB, specified as a real nonnegative scalar.

Data Types: `single` | `double`

### ConstantBias — Constant sensor offset bias (m/s<sup>2</sup>)

`[0 0 0]` (default) | real scalar | real 3-element row vector

Constant sensor offset bias in m/s<sup>2</sup>, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

**AxesMisalignment — Sensor axes skew (%)**

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Sensor axes skew in %, specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**NoiseDensity — Power spectral density of sensor noise (m/s<sup>2</sup>/√Hz)**

[0 0 0] (default) | real scalar | real 3-element row vector

Power spectral density of sensor noise in (m/s<sup>2</sup>/√Hz), specified as a real scalar or 3-element row vector. This property corresponds to the velocity random walk (VRW). Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**BiasInstability — Instability of the bias offset (m/s<sup>2</sup>)**

[0 0 0] (default) | real scalar | real 3-element row vector

Instability of the bias offset in m/s<sup>2</sup>, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**RandomWalk — Integrated white noise of sensor ((m/s<sup>2</sup>)(√Hz))**

[0 0 0] (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in (m/s<sup>2</sup>)(√Hz), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**TemperatureBias — Sensor bias from temperature ((m/s<sup>2</sup>)/°C)**

[0 0 0] (default) | real scalar | real 3-element row vector

Sensor bias from temperature in (m/s<sup>2</sup>)/°C, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**TemperatureScaleFactor — Scale factor error from temperature (%/°C)**

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Scale factor error from temperature in %/°C, specified as a real scalar or real 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Examples**

## Generate Accelerometer Data from Stationary Inputs

Generate accelerometer data for an imuSensor object from stationary inputs.

Generate an accelerometer parameter object with a maximum sensor reading of  $19.6 \text{ m/s}^2$  and a resolution of  $0.598 (\text{mm/s}^2)/\text{LSB}$ . The constant offset bias is  $0.49 \text{ m/s}^2$ . The sensor has a power spectral density of  $3920 (\mu\text{m/s}^2)/\sqrt{\text{Hz}}$ . The bias from temperature is  $0.294 (\text{m/s}^2)/0^\circ\text{C}$ . The scale factor error from temperature is  $0.02\%/0^\circ\text{C}$ . The sensor axes are skewed by 2%.

```
params = accelparams('MeasurementRange', 19.6, 'Resolution', 0.598e-3, 'ConstantBias', 0.49, 'NoiseDen...
```

Use a sample rate of 100 Hz spaced out over 1000 samples. Create the imuSensor object using the accelerometer parameter object.

```
Fs = 100;
numSamples = 1000;
t = 0:1/Fs:(numSamples-1)/Fs;

imu = imuSensor('SampleRate', Fs, 'Accelerometer', params);
```

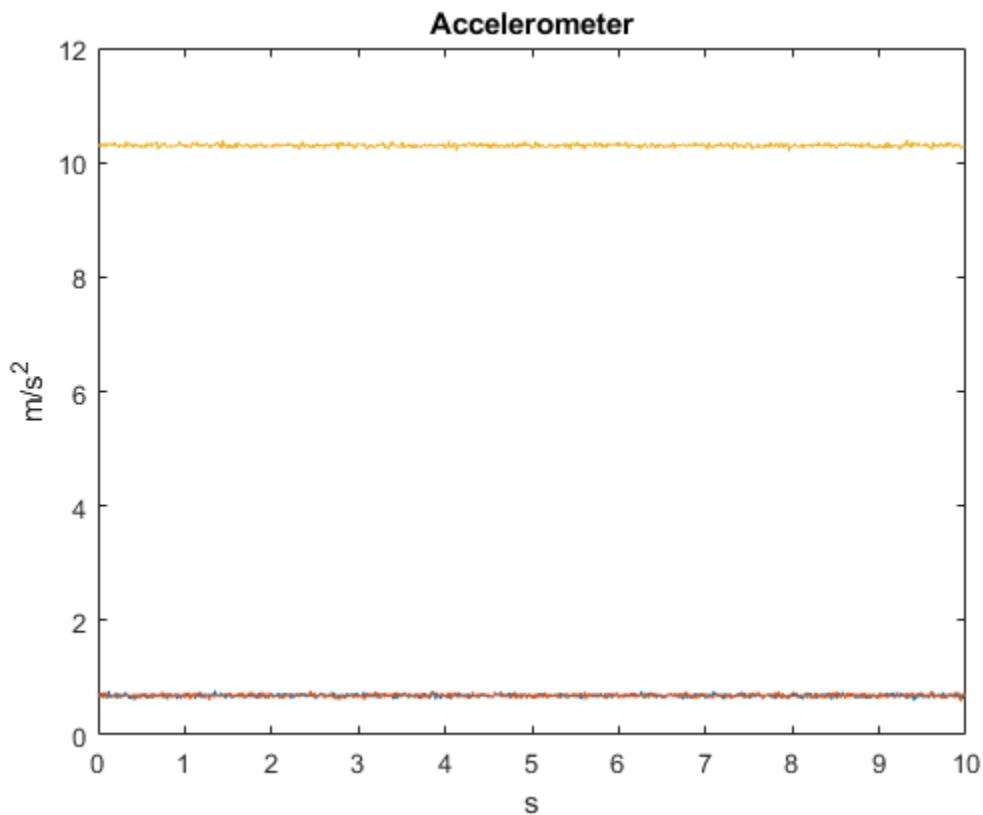
Generate accelerometer data from the imuSensor object.

```
orient = quaternion.ones(numSamples, 1);
acc = zeros(numSamples, 3);
angvel = zeros(numSamples, 3);

accelData = imu(acc, angvel, orient);
```

Plot the resultant accelerometer data.

```
plot(t, accelData)
title('Accelerometer')
xlabel('s')
ylabel('m/s^2')
```



## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[gyroparams](#) | [imuSensor](#) | [magparams](#)

### Topics

"Model IMU, GPS, and INS/GPS"

### Introduced in R2018b

## gyroparams

Gyroscope sensor parameters

### Description

The `gyroparams` class creates a gyroscope sensor parameters object. You can use this object to model a gyroscope when simulating an IMU with `imuSensor`.

### Creation

#### Syntax

```
params = gyroparams  
params = gyroparams(Name,Value)
```

#### Description

`params = gyroparams` returns an ideal gyroscope sensor parameters object with default values.

`params = gyroparams(Name,Value)` configures `gyroparams` 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.

### Properties

#### MeasurementRange — Maximum sensor reading (rad/s)

`Inf` (default) | real positive scalar

Maximum sensor reading in rad/s, specified as a real positive scalar.

Data Types: `single` | `double`

#### Resolution — Resolution of sensor measurements ((rad/s)/LSB)

`0` (default) | real nonnegative scalar

Resolution of sensor measurements in (rad/s)/LSB, specified as a real nonnegative scalar

Data Types: `single` | `double`

#### ConstantBias — Constant sensor offset bias (rad/s)

`[0 0 0]` (default) | real scalar | real 3-element row vector

Constant sensor offset bias in rad/s, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

**AxesMisalignment — Sensor axes skew (%)**

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Sensor axes skew in %, specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**NoiseDensity — Power spectral density of sensor noise ((rad/s)/√Hz)**

[0 0 0] (default) | real scalar | real 3-element row vector

Power spectral density of sensor noise in (rad/s)/√Hz, specified as a real scalar or 3-element row vector. This property corresponds to the angle random walk (ARW). Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**BiasInstability — Instability of the bias offset (rad/s)**

[0 0 0] (default) | real scalar | real 3-element row vector

Instability of the bias offset in rad/s, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**RandomWalk — Integrated white noise of sensor ((rad/s)(√Hz))**

[0 0 0] (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in (rad/s)(√Hz), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**TemperatureBias — Sensor bias from temperature ((rad/s)/°C)**

[0 0 0] (default) | real scalar | real 3-element row vector

Sensor bias from temperature in ((rad/s)/°C), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**TemperatureScaleFactor — Scale factor error from temperature (%/°C)**

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Scale factor error from temperature in (%/°C), specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**AccelerationBias — Sensor bias from linear acceleration (rad/s)/(m/s<sup>2</sup>)**

[0 0 0] (default) | real scalar | real 3-element row vector

Sensor bias from linear acceleration in  $(\text{rad}/\text{s})/(\text{m}/\text{s}^2)$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

## Examples

### Generate Gyroscope Data from Stationary Inputs

Generate gyroscope data for an `imuSensor` object from stationary inputs.

Generate a gyroscope parameter object with a maximum sensor reading of 4.363 rad/s and a resolution of 1.332e-4  $(\text{rad}/\text{s})/\text{LSB}$ . The constant offset bias is 0.349 rad/s. The sensor has a power spectral density of  $8.727\text{e-}4 \frac{\text{rad}/\text{s}}{\sqrt{\text{Hz}}}$ . The bias from temperature is  $0.349 \frac{\text{rad}/\text{s}}{^\circ\text{C}}$ . The scale factor error from temperature is 0.2%/ $^\circ\text{C}$ . The sensor axes are skewed by 2%. The sensor bias from linear acceleration is  $0.178\text{e-}3 (\text{rad}/\text{s})/(\text{m}/\text{s}^2)$

```
params = gyroparams('MeasurementRange', 4.363, 'Resolution', 1.332e-04, 'ConstantBias', 0.349, 'NoiseD
```

Use a sample rate of 100 Hz spaced out over 1000 samples. Create the `imuSensor` object using the gyroscope parameter object.

```
Fs = 100;
numSamples = 1000;
t = 0:1/Fs:(numSamples-1)/Fs;

imu = imuSensor('accel-gyro', 'SampleRate', Fs, 'Gyroscope', params);
```

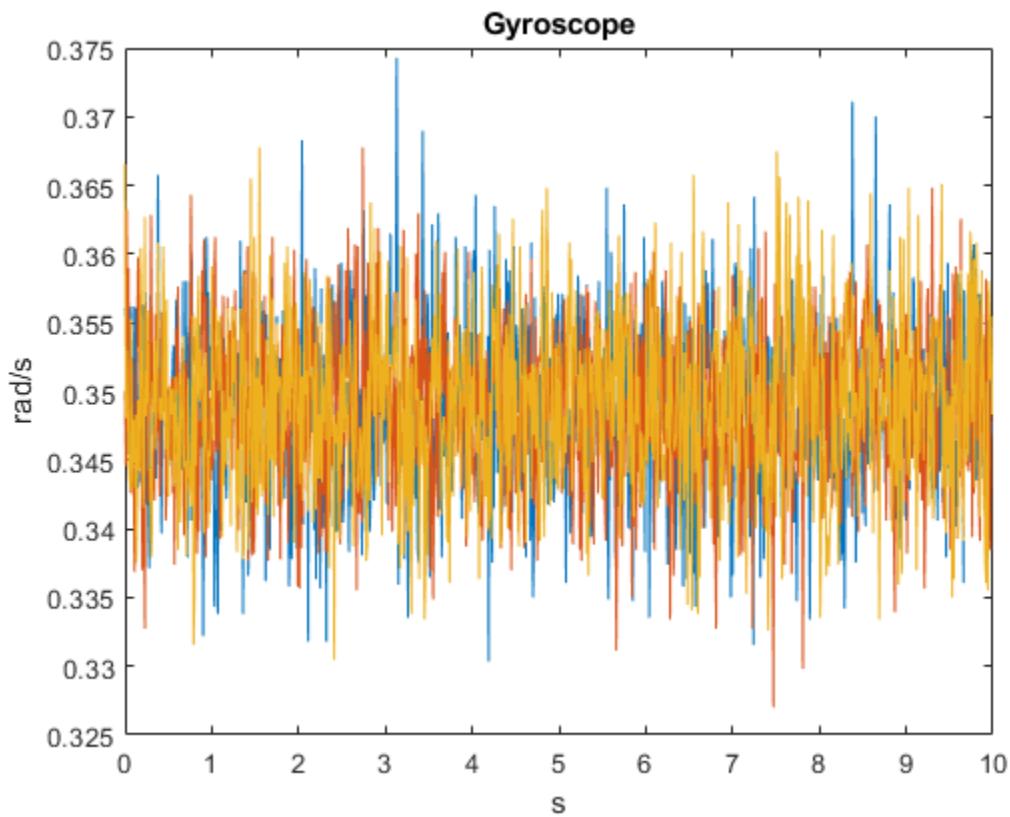
Generate gyroscope data from the `imuSensor` object.

```
orient = quaternion.ones(numSamples, 1);
acc = zeros(numSamples, 3);
angvel = zeros(numSamples, 3);

[~, gyroData] = imu(acc, angvel, orient);
```

Plot the resultant gyroscope data.

```
plot(t, gyroData)
title('Gyroscope')
xlabel('s')
ylabel('rad/s')
```



## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[accelparams](#) | [imuSensor](#) | [magparams](#)

### Topics

"Model IMU, GPS, and INS/GPS"

### Introduced in R2018b

# magparams

Magnetometer sensor parameters

## Description

The `magparams` class creates a magnetometer sensor parameters object. You can use this object to model a magnetometer when simulating an IMU with `imuSensor`.

## Creation

### Syntax

```
params = magparams  
params = magparams(Name,Value)
```

### Description

`params = magparams` returns an ideal magnetometer sensor parameters object with default values.

`params = magparams(Name,Value)` configures `magparams` 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.

## Properties

### MeasurementRange — Maximum sensor reading ( $\mu\text{T}$ )

`Inf` (default) | real positive scalar

Maximum sensor reading in  $\mu\text{T}$ , specified as a real positive scalar.

Data Types: `single` | `double`

### Resolution — Resolution of sensor measurements ( $\mu\text{T}/\text{LSB}$ )

`0` (default) | real nonnegative scalar

Resolution of sensor measurements in  $\mu\text{T}/\text{LSB}$ , specified as a real nonnegative scalar

Data Types: `single` | `double`

### ConstantBias — Constant sensor offset bias ( $\mu\text{T}$ )

`[0 0 0]` (default) | real scalar | real 3-element row vector

Constant sensor offset bias in  $\mu\text{T}$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: `single` | `double`

**AxesMisalignment — Sensor axes skew (%)**

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Sensor axes skew in %, specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**NoiseDensity — Power spectral density of sensor noise ( $\mu\text{T}/\sqrt{\text{Hz}}$ )**

[0 0 0] (default) | real scalar | real 3-element row vector

Power spectral density of sensor noise in  $\mu\text{T}/\sqrt{\text{Hz}}$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**BiasInstability — Instability of the bias offset ( $\mu\text{T}$ )**

[0 0 0] (default) | real scalar | real 3-element row vector

Instability of the bias offset in  $\mu\text{T}$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**RandomWalk — Integrated white noise of sensor ( $\mu\text{T}/\sqrt{\text{Hz}}$ )**

[0 0 0] (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in  $(\mu\text{T}/\sqrt{\text{Hz}})$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**TemperatureBias — Sensor bias from temperature ( $\mu\text{T}/^\circ\text{C}$ )**

[0 0 0] (default) | real scalar | real 3-element row vector

Sensor bias from temperature in  $(\mu\text{T}/^\circ\text{C})$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**TemperatureScaleFactor — Scale factor error from temperature (%/ $^\circ\text{C}$ )**

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Scale factor error from temperature in (%/ $^\circ\text{C}$ ), specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

## Examples

### Generate Magnetometer Data from Stationary Inputs

Generate magnetometer data for an imuSensor object from stationary inputs.

Generate a magnetometer parameter object with a maximum sensor reading of 1200  $\mu\text{T}$  and a resolution of 0.1  $\mu\text{T}/\text{LSB}$ . The constant offset bias is 1  $\mu\text{T}$ . The sensor has a power spectral density of  $\left(\frac{[0.6 \ 0.6 \ 0.9]}{\sqrt{100}}\right) \mu\text{T}/\sqrt{\text{Hz}}$ . The bias from temperature is [0.8 0.8 2.4]  $\mu\text{T}/^{\circ}\text{C}$ . The scale factor error from temperature is 0.1 %/ $^{\circ}\text{C}$ .

```
params = magparams('MeasurementRange',1200,'Resolution',0.1,'ConstantBias',1,'NoiseDensity',[0.6
```

Use a sample rate of 100 Hz spaced out over 1000 samples. Create the imuSensor object using the magnetometer parameter object.

```
Fs = 100;
numSamples = 1000;
t = 0:1/Fs:(numSamples-1)/Fs;

imu = imuSensor('accel-mag','SampleRate', Fs, 'Magnetometer', params);
```

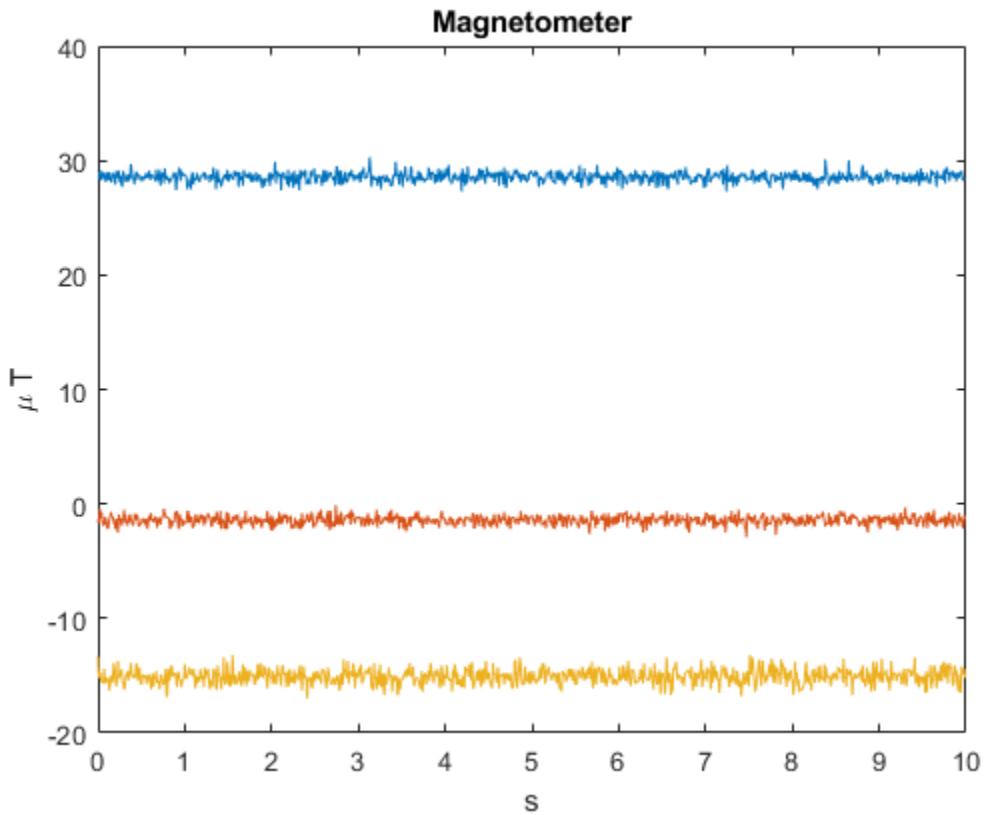
Generate magnetometer data from the imuSensor object.

```
orient = quaternion.ones(numSamples, 1);
acc = zeros(numSamples, 3);
angvel = zeros(numSamples, 3);

[~, magData] = imu(acc, angvel, orient);
```

Plot the resultant magnetometer data.

```
plot(t, magData)
title('Magnetometer')
xlabel('s')
ylabel('\mu T')
```



## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[accelparams](#) | [gyroparams](#) | [imuSensor](#)

### Topics

"Model IMU, GPS, and INS/GPS"

**Introduced in R2018b**

# 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.

## 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, the trackers immediately create 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 <code>HasVelocity</code> is false, the measurements are reported as $[x \ y \ z]$ . If <code>HasVelocity</code> is true, measurements are reported as $[x \ y \ z \ vx \ vy \ vz]$ .	1
IsParentToChild	Logical scalar indicating if <code>Orientation</code> performs a frame rotation from the parent coordinate frame to the child coordinate frame. When <code>IsParentToChild</code> is false, then <code>Orientation</code> 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 trackers but not used by the trackers.

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

`irSensor` | `monostaticRadarSensor` | `radarSensor` | `sonarSensor` | `trackerGNN` | `trackerJPDA` | `trackerPHD` | `trackerTOMHT` | `trackingABF` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

### Introduced in R2018b

# 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

**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

**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<sub>x</sub>; y; v<sub>y</sub>; z; v<sub>z</sub>].

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

**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 scalar | 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 (senors, 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.

**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.

**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

`fuserSourceConfiguration` | `objectDetection` | `trackFuser`

**Introduced in R2019b**

# 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 R2019b

# quaternion

Create a quaternion array

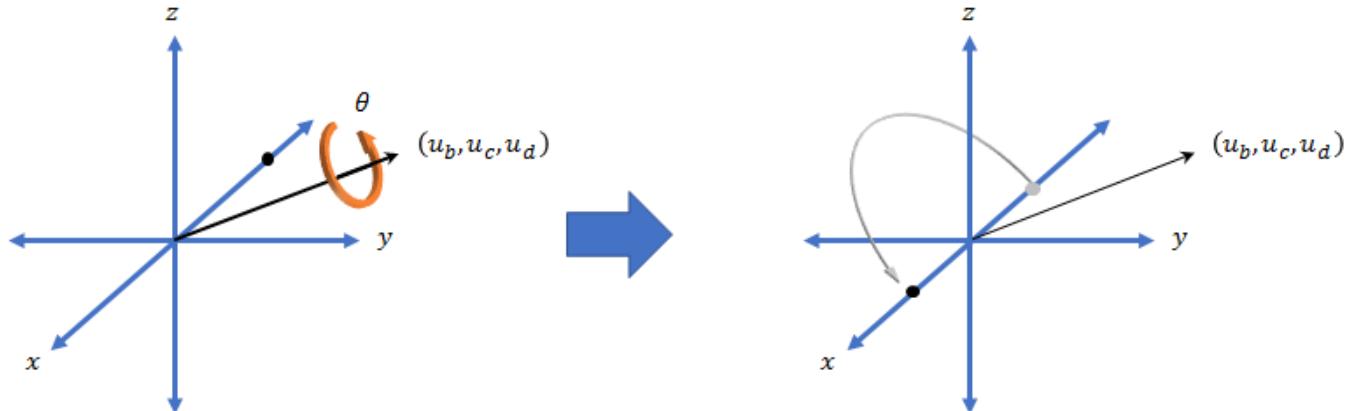
## Description

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  $[u_b, u_c, u_d]$  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'
- 'XYX'
- '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.

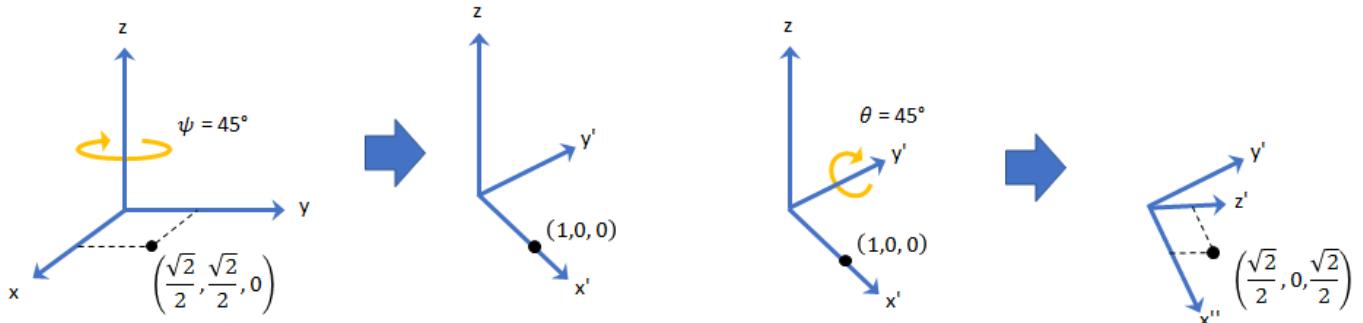
You want to rotate the point using the Euler angle representation [45,45,0]. Rotate the point using two different rotation sequences:

- 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

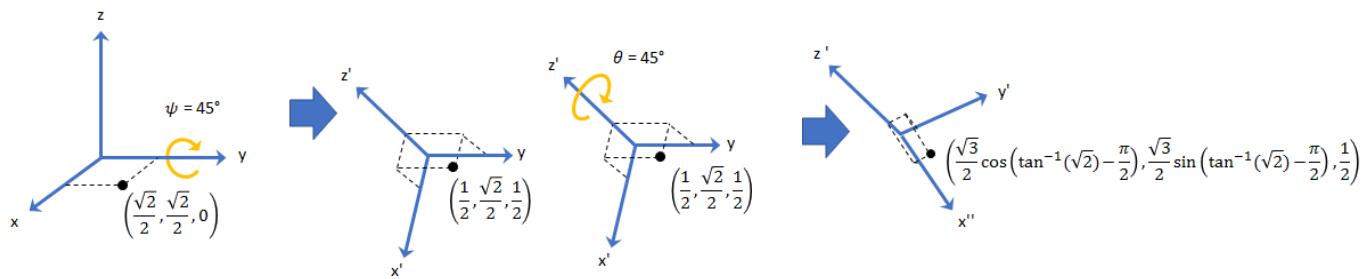


- If you create a quaternion rotator and specify the 'YZX' sequence, the frame is first rotated 45° around the y-axis, then 45° around the new z-axis.

```
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
ctranspose, '	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
minus, -	Quaternion subtraction
mtimes, *	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, .'	Transpose a quaternion array
uminus, -	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=2×1 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=2×2 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 = 3x3

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 = 1x3

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 = 1x3

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
```

```

-1 - 2i - 3j - 4k      -9 - 8i - 7j - 6k

qMultiDimensionalArray(:,:,1) = qMatrix;
qMultiDimensionalArray(:,:,2) = qMatrix

qMultiDimensionalArray = 2x2x2 quaternion array
qMultiDimensionalArray(:,:,1) =

1 + 2i + 3j + 4k      9 + 8i + 7j + 6k
-1 - 2i - 3j - 4k      -9 - 8i - 7j - 6k

qMultiDimensionalArray(:,:,2) =

1 + 2i + 3j + 4k      9 + 8i + 7j + 6k
-1 - 2i - 3j - 4k      -9 - 8i - 7j - 6k

```

## 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) =

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

```

## 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

### Topics

“Rotations, Orientation, and Quaternions”  
“Lowpass Filter Orientation Using Quaternion SLERP”

**Introduced in R2018b**

# trackingScenario

Create tracking scenario

## Description

`trackingScenario` creates a tracking scenario object. A tracking scenario simulates a 3-D arena containing multiple platforms. Platforms represent anything 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 sources of signals or reflect signals. Platforms can also include stationary obstacles that can influence the motion of other platforms. Platforms can be modeled as points or cuboids by specifying the 'Dimension' property when calling `platform`. Platforms can have aspect-dependent properties including radar cross-section or sonar target strength. You can populate a tracking scenario by calling the `platform` method for each platform you want to add. Platforms are `Platform` objects. You can create trajectories for any platform using the `kinematicTrajectory`, `waypointTrajectory`, or `geoTrajectory` System objects. After creating the scenario, run the simulation by calling the `advance` object function.

## Creation

`sc = trackingScenario` creates an empty tracking scenario with default property values. In this case, you can specify platform trajectories in the scenario as Cartesian states using the `kinematicTrajectory` or `waypointTrajectory` objects.

`sc = trackingScenario('IsEarthCentered', true)` creates an empty Earth-centered tracking scenario with default property values. In this case, you can specify platform trajectories in the scenario as geodetic states using the `geoTrajectory` object.

`sc = trackingScenario(Name, Value)` configures a `trackingScenario` object with 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.

## Properties

**IsEarthCentered — Enable Earth-centered reference frame and trajectories**  
`false` (default) | `true`

Enable Earth-centered reference frame and trajectories, specified as `true` or `false`.

- If specified as `false`, you must use the `kinematicTrajectory` or `waypointTrajectory` object to define the trajectories of platforms as Cartesian states in the tracking scenario.
- If specified as `true`, you must use the `geoTrajectory` object to define the trajectories of platforms in the ECEF (Earth-Centered-Earth-Fixed) as geodetic coordinates in the tracking scenario. In this case, you must specify the `IsEarthCentered` property during the tracking scenario creation.

Data Types: `logical`

**StopTime — Stop time of simulation**

Inf (default) | positive scalar

Stop time of simulation, specified as a positive scalar. 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 using waypoints, **waypointTrajectory**.

Units are in seconds.

Example: 60.0

Data Types: double

**SimulationTime — Current time of simulation**

positive scalar

This property is read-only.

Current time of the simulation, defined as a positive scalar. To reset the simulation time to zero and restart the simulation, call the **restart** method. Units are in seconds.

Data Types: double

**UpdateRate — Frequency of simulation updates**

10.0 (default) | positive scalar

Frequency of simulation updates, specified as a positive scalar. This is the rate at which to provide successive updates of the scenario simulation. Units are in Hz.

Example: 2.0

Data Types: double

**IsRunning — Run-state of simulation**

true | false

This property is read-only.

Run-state of simulation, defined as **true** or **false**. If the simulation is running, **IsRunning** is **true**. If the simulation has stopped, **IsRunning** is **false**. 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 that platform **Motion** strategy with waypoints using the **waypointTrajectory** System object.

Units are in seconds.

Data Types: logical

**Platforms — Platforms in the simulation**

cell | cell array

This property is read-only.

Platforms in the scenario, returned as a cell or cell array of `Platform` objects. To add a platform to the scenario, use the `platform` object function.

## Object Functions

platform	Add platform to tracking scenario
advance	Advance tracking scenario simulation by one time step
restart	Restart tracking scenario simulation
record	Run tracking scenario and record platform, sensor, and emitter information
emit	Collect emissions from emitters in tracking scenario
propagate	Propagate emissions in tracking scenario
detect	Collect detections from all the sensors in tracking scenario
lidarDetect	Report point cloud detections from all lidar sensor in trackingScenario
platformPoses	Positions, velocities, and orientations of all platforms in tracking scenario
platformProfiles	Profiles of platforms in tracking scenario
coverageConfig	Sensor and emitter coverage configuration
clone	Create copy of tracking scenario
perturb	Apply perturbations to tracking scenario

## Examples

### Create Tracking Scenario with Two Platforms

Construct a tracking scenario with two platforms that follow different trajectories.

```
sc = trackingScenario('UpdateRate',100.0,'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 fusion.scenario.Platform} {1x1 fusion.scenario.Platform}
```

Run the simulation and plot the current position of each platform. Use an animated line to plot the position of each platform.

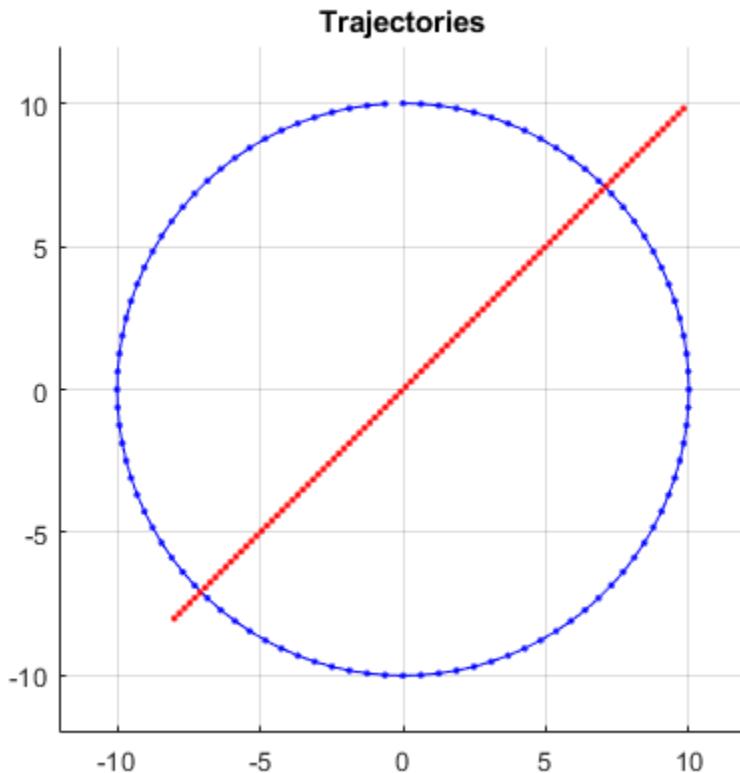
```
figure
grid
```

```

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

```

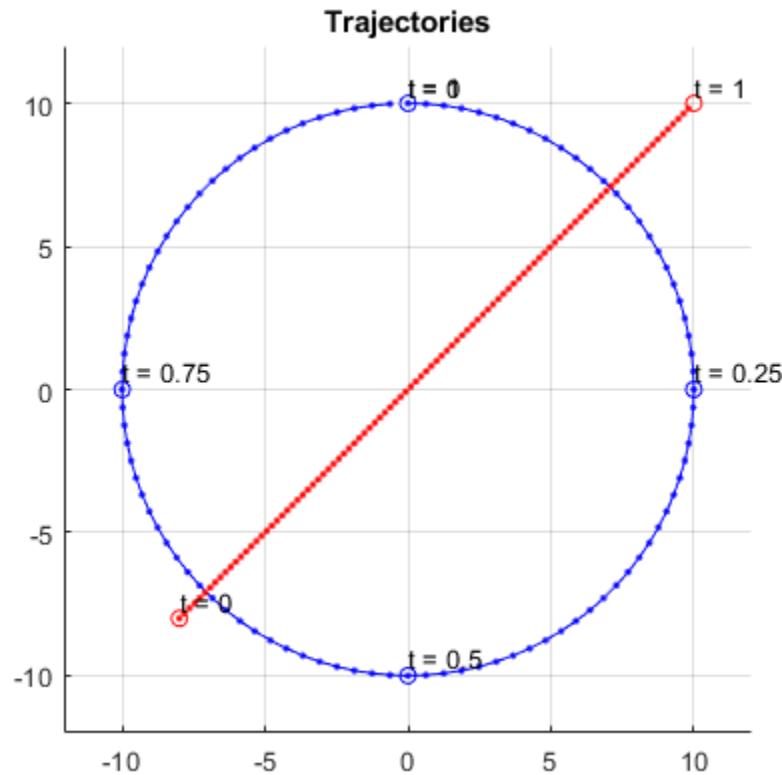


Plot the waypoints for both platforms.

```

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 Scenario

Create a tracking scenario with specified updated rate.

```
scene = trackingScenario('IsEarthCentered',true,'UpdateRate',0.01);
```

Add an airplane in the scenario. The trajectory of the airplane changes in latitude and altitude.

Advance the tracking scenario and record the geodetic and Cartesian positions of the plane target.

```

positions = [];
while advance(scene)
    poseLLA = pose(plane, 'CoordinateSystem', 'Geodetic');
    poseCart = pose(plane, 'CoordinateSystem', 'Cartesian');
    positions = [positions;poseCart.Position];%#ok<AGROW> Allow the buffer to grow.
end

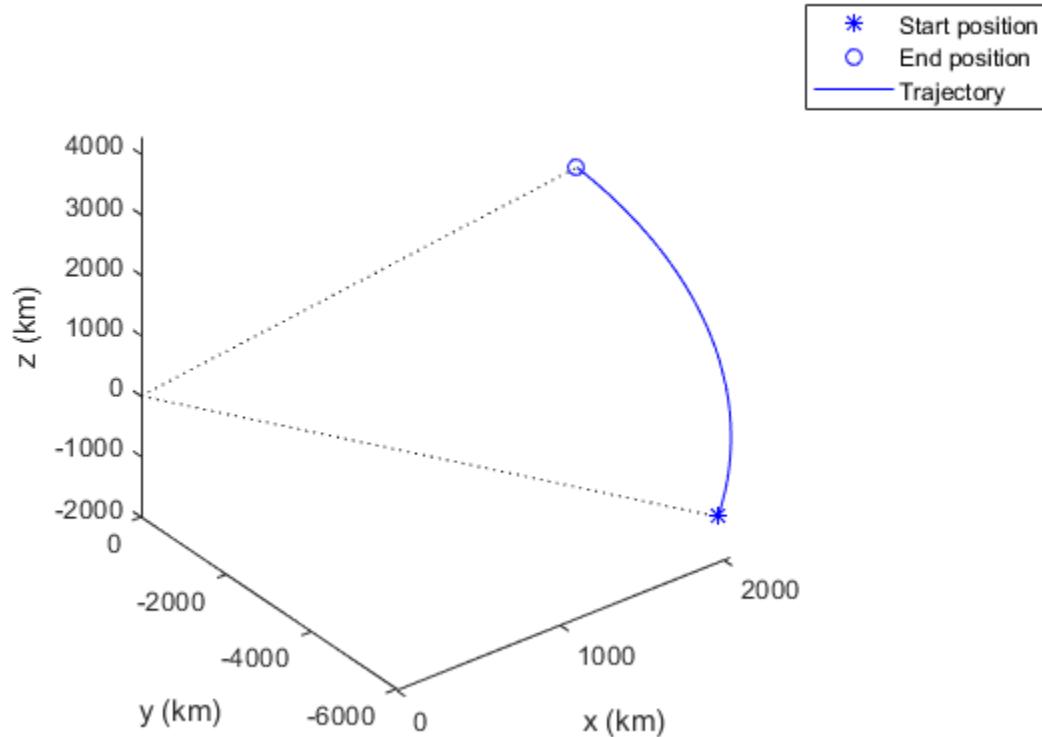
```

Visualize the trajectory in the ECEF frame.

```
figure()
km = 1000;
% Plot the trajectory.
plot3(positions(1,1)/km,positions(1,2)/km,positions(1,3)/km, 'b*' );
```

```
hold on;
plot3(positions(end,1)/km,positions(end,2)/km,positions(end,3)/km, 'bo');
plot3(positions(:,1)/km,positions(:,2)/km,positions(:,3)/km, 'b');

% Plot the Earth radial lines.
plot3([0 positions(1,1)]/km,[0 positions(1,2)]/km,[0 positions(1,3)]/km, 'k:');
plot3([0 positions(end,1)]/km,[0 positions(end,2)]/km,[0 positions(end,3)]/km, 'k:');
xlabel('x (km)'); ylabel('y (km)'); zlabel('z (km)');
legend('Start position','End position','Trajectory')
```



## See Also

### System Objects

[kinematicTrajectory](#) | [waypointTrajectory](#)

**Introduced in R2018b**

# Platform

Platform object belonging to tracking scenario

## Description

`Platform` defines a platform object belonging to a tracking scenario. Platforms represent the moving objects in a scenario and are modeled as points or cuboids with aspect-dependent properties.

## Creation

You can create `Platform` objects using the `platform` method of `trackingScenario`.

## Properties

### **PlatformID — Scenario-defined platform identifier**

1 (default) | 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.

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**

3-element vector of scalar

This property is read-only.

Current position of the platform, specified as a 3-element vector of scalars.

- When the `IsEarthCentered` property of the scenario is set to `false`, the position is expressed 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 expressed as a three element geodetic state: `latitude` in degrees, `longitude` in degrees, and `altitude` in meters.

Data Types: double

**Orientation — Current orientation of platform**

3-element vector of scalar

This property is read-only.

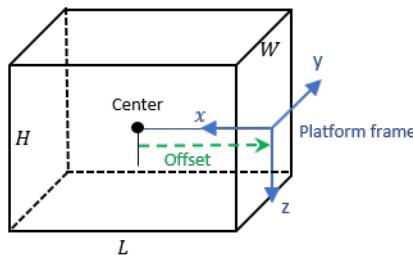
Current orientation of the platform, specified as a 3-element vector of scalars in degrees. The three scalars are the [yaw, pitch, roll] rotation angles from the local reference frame to the platform's body frame.

Data Types: double

**Dimensions — Platform dimensions and origin offset**

struct

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 following 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 either 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`, you can use the `kinematicTrajectory` or the `waypointTrajectory` object. By default, a stationary `kinematicTrajectory` object is used.
- When the `IsEarthCentered` property of the scenario is set to `true`, you can only use the `geoTrajectory` object. By default, a stationary `geoTrajectory` object is used.

### **Signatures — Platform signatures**

{`rcsSignature` `irSignature` `tsSignature`} (default) | cell array of signature objects

Platform signatures, specified as a cell array of `irSignature`, `rcsSignature`, and `tsSignature` objects or an empty cell array. The cell array contains at most only one instance for each type of signature objects listed. 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

A pose estimator, specified as a pose-estimator object such as `insSensor`. The pose estimator determines platform pose with respect to the local NED scenario coordinate. The interface of any pose estimator must match the interface of `insSensor`. By default, pose-estimator accuracy properties are set to zero.

### **Emitters — Emitters mounted on platform**

cell array of emitter objects

Emitters mounted on platform, specified as a cell array of emitter objects, such as `radarEmitter` or `sonarEmitter`.

### **Sensors — Sensors mounted on platform**

cell array of sensor objects

Sensors mounted on platform, specified as a cell array of sensor objects such as `irSensor`, `radarSensor`, `monostaticRadarSensor`, or `sonarSensor`.

### **Mesh — Mesh of platform**

`extendedObjectMesh` object (default)

Mesh of platform, specified as an `extendedObjectMesh` object. The object represents the mesh as vertices and faces. The `monostaticLidarSensor` object uses the platform mesh information to generate cloud data.

## **Object Functions**

<code>detect</code>	Detect signals using platform-mounted sensors
<code>lidarDetect</code>	Report point cloud detections from all lidar sensor on platform
<code>emit</code>	Radiate signals from emitters mounted on platform
<code>pose</code>	Pose of platform
<code>targetPoses</code>	Target positions and orientations as seen from platform
<code>targetMeshes</code>	Target meshes as seen from platform

## Examples

### Platform Follows Circular Trajectory

Create a tracking scenario and a platform following a circular path.

```
scene = trackingScenario('UpdateRate',1/50);

% Create a platform
plat = platform(scene);

% Follow a circular trajectory 1 km in radius completing in 400 hundred seconds.
plat.Trajectory = waypointTrajectory('Waypoints', [0 1000 0; 1000 0 0; 0 -1000 0; -1000 0 0; 0 1000 0], ...
    'TimeOfArrival', [0; 100; 200; 300; 400]);

% Perform the simulation
while scene.advance
    p = pose(plat);
    fprintf('Time = %f ', scene.SimulationTime);
    fprintf('Position = [');
    fprintf('%f ', p.Position);
    fprintf('] Velocity = [');
    fprintf('%f ', p.Velocity);
    fprintf(']\n');
end

Time = 50.000000
Position = [
707.095476 707.100019 0.000000
] Velocity = [
11.107152 -11.107075 0.000000
]
Time = 100.000000
Position = [
1000.000000 0.000000 0.000000
] Velocity = [
0.000476 -15.707961 0.000000
]
Time = 150.000000
Position = [
707.115558 -707.115461 0.000000
] Velocity = [
-11.107346 -11.107341 0.000000
]
```

```
]  
Time = 200.000000  
Position = [  
0.000000 -1000.000000 0.000000  
] Velocity = [  
-15.707963 0.000460 0.000000  
]  
Time = 250.000000  
Position = [  
-707.098004 -707.098102 0.000000  
] Velocity = [  
-11.107069 11.107074 0.000000  
]  
Time = 300.000000  
Position = [  
-1000.000000 0.000000 0.000000  
] Velocity = [  
-0.000476 15.707966 0.000000  
]  
Time = 350.000000  
Position = [  
-707.118086 707.113543 0.000000  
] Velocity = [  
11.107262 11.107340 0.000000  
]  
Time = 400.000000  
Position = [  
-0.000000 1000.000000 0.000000  
] Velocity = [  
15.708226 -0.000493 0.000000  
]
```

## Cuboid Platforms Follow Circular Trajectories

Create a tracking scenario with two cuboid platforms following circular trajectories.

```
sc = trackingScenario;

% Create the platform for a truck with dimension 5 x 2.5 x 3.5 (m).
p1 = platform(sc);
p1.Dimensions = struct('Length',5,'Width',2.5,'Height',3.5,'OriginOffset',[0 0 0]);

% Specify the truck's trajectory as a circle with radius 20 meters.
p1.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 dimension .3 x .3 x .1 (m).
p2 = platform(sc);
p2.Dimensions = struct('Length',.3,'Width',.3,'Height',.1,'OriginOffset',[0 0 0]);

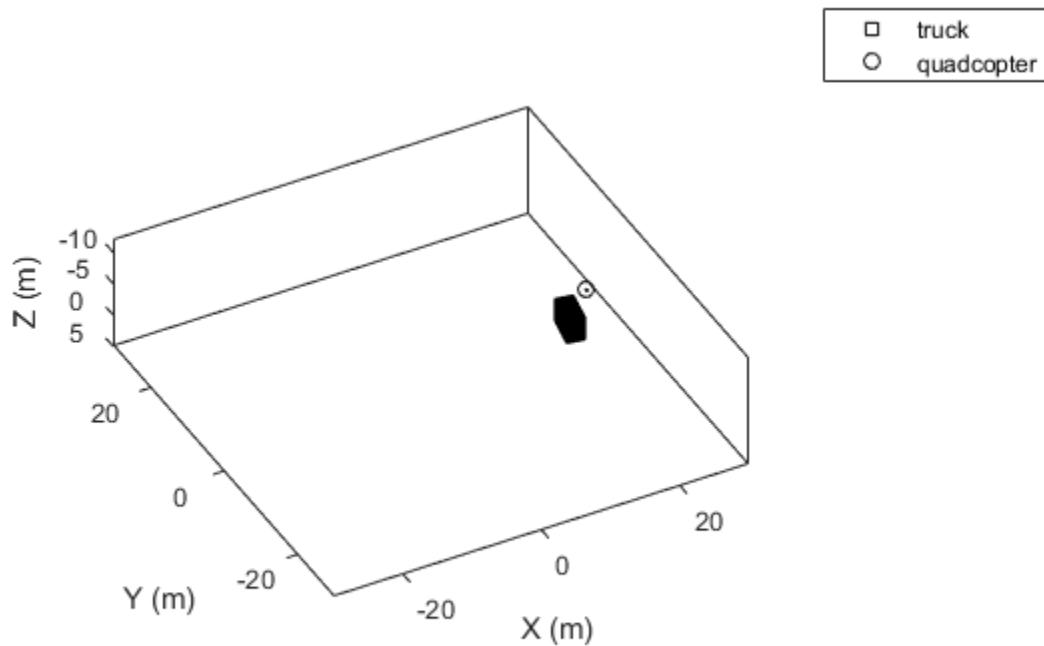
% The quadcopter follows the truck at 10 meters above with small angular delay.
% Note that the negative z coordinates correspond to positive elevation.
p2.Trajectory = waypointTrajectory('Waypoints', [20*cos(2*pi*((0:10)'-.6)/10)...
    20*sin(2*pi*((0:10)'-.6)/10) -11.80*ones(11,1)], ...
    'TimeOfArrival', 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 animate.
view(-28,37);
set(gca,'Zdir','reverse');

while advance(sc)
    poses = platformPoses(sc);
    plotPlatform(pp1, poses(1).Position, p1.Dimensions, poses(1).Orientation);
    plotPlatform(pp2, poses(2).Position, p2.Dimensions, poses(2).Orientation);
end
```



## See Also

### Classes

[rcsSignature](#) | [tsSignature](#)

### System Objects

[insSensor](#) | [irSensor](#) | [kinematicTrajectory](#) | [monostaticRadarSensor](#) | [radarEmitter](#) | [radarSensor](#) | [sonarEmitter](#) | [sonarSensor](#) | [waypointTrajectory](#)

**Introduced in R2018b**

# Platform.emit

Radiate signals from emitters mounted on platform

## Syntax

```
[signals,emitterconfigs] = emit(ptfm,time)
```

## Description

[signals,emitterconfigs] = emit(ptfm,time) returns signals, `signals`, radiated by all the emitters mounted on the platform, `ptfm`, at the specified `time`. The function also returns all emitter configurations, `emitterconfigs`.

## Input Arguments

### `ptfm` — Scenario platform

Platform object

Scenario platform, specified as a `Platform` object. To create platforms, use the `platform` method.

### `time` — Emission time

0 (default) | positive scalar

Emission time, specified as a positive scalar.

Example: 100.5

Data Types: `single` | `double`

## Output Arguments

### `signals` — Signals radiated by emitters on platform

cell array of emission objects

Signals radiated by emitters on platform, returned as a cell array of `radarEmission` and `sonarEmission` objects.

### `emitterconfigs` — Emitter configurations

structure

Emitter configurations, returned as a structure. An emitter configuration has these fields:

Field	Description
<code>EmitterIndex</code>	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.

Data Types: single | double

## See Also

[detect](#) | [pose](#) | [targetPoses](#)

**Introduced in R2018b**

# Platform.detect

Detect signals using platform-mounted sensors

## Syntax

```
dets = detect(ptfm,time)
dets = detect(ptfm,signals,time)
dets = detect(ptfm,signals,emitterconfigs,time)
[dets,numDets] = detect(___)
[dets,numDets,sensorconfigs] = detect(___)
```

## Description

`dets = detect(ptfm,time)` returns detections, `dets`, from all the sensors mounted on the platform, `ptfm`, at the specified `time`. This syntax applies when sensors do not require knowledge of any signals present in the scenario, for example, when the `monostaticRadarSensor` object property `HasInterference` is set to `false`.

`dets = detect(ptfm,signals,time)` also specifies any signals, `signals`, present in the scenario. This syntax applies when sensors require knowledge of these signals, for example, when a `radarSensor` object is configured as an EM sensor.

`dets = detect(ptfm,signals,emitterconfigs,time)` also specifies emitter configurations, `emitterconfigs`. This syntax applies when sensors require knowledge of the configurations of emitters generating signals in the scenario. For example, when an `radarSensor` object is configured as a monostatic radar.

`[dets,numDets] = detect(__)` also returns the number of detections, `numDets`. This output syntax can be used with any of the input syntaxes.

`[dets,numDets,sensorconfigs] = detect(__)` also returns all sensor configurations, `sensorconfigs`. This output syntax can be used with any of the input syntaxes.

## Input Arguments

### **ptfm – Scenario platform**

Platform object

Scenario platform, specified as a `Platform` object. To create platforms, use the `platform` method.

### **time – Simulation time**

0 (default) | positive scalar

Simulation time specified as a positive scalar.

Example: 1.5

Data Types: `single` | `double`

### **signals – Signals in scenario**

cell array of emission objects

Signals in the scenario, specified as a cell array of `radarEmission` and `sonarEmission` emission objects.

**emitterconfigs — Emitter configurations**  
structure

Emitter configurations, specified as a structure. The fields of the emitter configuration are:

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.

Data Types: struct

## Output Arguments

**dets — sensor detections**

cell array of `objectDetection` objects

Sensor detections, returned as a cell array of `objectDetection` objects.

**sensorconfigs — Sensor configurations**

structure

Sensor configurations, returned as a structure. The fields of this structure are:

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.

<b>FieldOfView</b>	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.
<b>MeasurementParameters</b>	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

#### **numDets — Number of detections**

nonnegative integer

Number of detections reported, returned as a nonnegative integer.

Data Types: double

## More About

### **Object Detections**

#### **Measurements**

This section describes the structure of object detections.

The sensor measures the coordinates of the target. The **Measurement** and **MeasurementNoise** values are reported in the coordinate system specified by the **DetectionCoordinates** property of the sensor.

When the **DetectionCoordinates** property is 'Scenario', 'Body', or 'Sensor rectangular', the **Measurement** and **MeasurementNoise** values are reported in rectangular coordinates. Velocities are only reported when the range rate property, **HasRangeRate**, is true.

When the **DetectionCoordinates** property is 'Sensor spherical', the **Measurement** and **MeasurementNoise** values are reported in a spherical coordinate system derived from the sensor rectangular coordinate system. Elevation and range rate are only reported when **HasElevation** and **HasRangeRate** are true.

Measurements are ordered as [azimuth, elevation, range, range rate]. Reporting of elevation and range rate depends on the corresponding **HasElevation** and **HasRangeRate** property values. Angles are in degrees, range is in meters, and range rate is in meters per second.

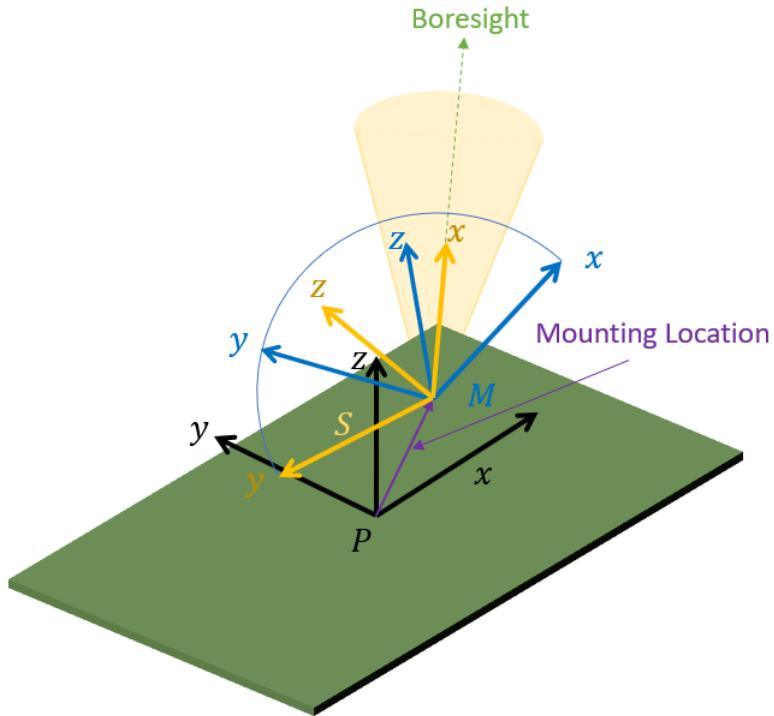
## Measurement Coordinates

DetectionCoordinates	Measurement and Measurement Noise Coordinates	
'Scenario'	Coordinate Dependence on HasRangeRate	
'Body'	HasRangeRate	Coordinates
'Sensor rectangular'	true	[x; y; z; vx; vy; vz]
	false	[x; y; z]
'Sensor spherical'	Coordinate Dependence on HasRangeRate and HasElevation	
HasRangeRate	HasElevation	Coordinates
true	true	[az; el; rng; rr]
true	false	[az; rng; rr]
false	true	[az; el; rng]
false	false	[az; rng]

## Measurement Parameters

The **MeasurementParameters** property consists of an array of structures that describe a sequence of coordinate transformations from a child frame to a parent frame or the inverse transformations (see “Frame Rotation”). 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 the 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 form the scenario frame to the platform frame then from 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.

Trivially, if the detections are reported in platform rectangular coordinates and `HasINS` is set to `false`, the transformation consists only of the identity.

The fields of `MeasurementParameters` are shown here. Not all fields have to be present in the structure. The set of fields and their default values can depend on the type of sensor.

Field	Description
<code>Frame</code>	Enumerated type indicating the frame used to report measurements. When detections are reported using a rectangular coordinate system, <code>Frame</code> is set to ' <code>rectangular</code> '. When detections are reported in spherical coordinates, <code>Frame</code> is set ' <code>spherical</code> ' for the first struct.
<code>OriginPosition</code>	Position offset of the origin of the child frame relative to the parent frame, represented as a 3-by-1 vector.
<code>OriginVelocity</code>	Velocity offset of the origin of the child frame relative to the parent frame, represented as a 3-by-1 vector.

<b>Orientation</b>	3-by-3 real-valued orthonormal frame rotation matrix. The direction of the rotation depends on the <b>IsParentToChild</b> field.
<b>IsParentToChild</b>	A logical scalar indicating if <b>Orientation</b> performs a frame rotation from the parent coordinate frame to the child coordinate frame. If <b>false</b> , <b>Orientation</b> performs a frame rotation from the child coordinate frame to the parent coordinate frame.
<b>HasElevation</b>	A logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if <b>HasElevation</b> is <b>false</b> , the measurements are reported assuming 0 degrees of elevation.
<b>HasAzimuth</b>	A logical scalar indicating if azimuth is included in the measurement.
<b>HasRange</b>	A logical scalar indicating if range is included in the measurement.
<b>HasVelocity</b>	A logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if <b>HasVelocity</b> is <b>false</b> , the measurements are reported as [x y z]. If <b>HasVelocity</b> is <b>true</b> , measurements are reported as [x y z vx vy vz].

### Object Attributes

Object attributes contain additional information about a detection:

Attribute	Description
<b>TargetIndex</b>	Identifier of the platform, <b>PlatformID</b> , that generated the detection. For false alarms, this value is negative.
<b>SNR</b>	Detection signal-to-noise ratio in dB.

### See Also

[emit](#) | [pose](#) | [targetPoses](#)

**Introduced in R2018b**

# lidarDetect

Report point cloud detections from all lidar sensor on platform

## Syntax

```
pointClouds = lidarDetect(plat,time)
pointClouds = lidarDetect(plat,time,includeSelf)
[pointClouds,configs] = lidarDetect(___)
[pointClouds,configs, clusters] = lidarDetect(___)
```

## Description

`pointClouds = lidarDetect(plat,time)` reports the point cloud detections from all `monostaticLidarSensor` object mounted on the ego platform, `plat`.

`pointClouds = lidarDetect(plat,time,includeSelf)` allows you to choose whether the lidar sensor reports detections of the ego platform, `plat`, on which the sensors are mounted. Specify `includeSelf` as `true` or `false`.

`[pointClouds,configs] = lidarDetect(__)` also returns the configurations of the sensors, `configs`, at the current simulation time. You can use these output arguments with any of the previous input syntaxes.

`[pointClouds,configs, clusters] = lidarDetect(__)` also returns `clusters`, the cluster labels for each point in the point cloud.

## Examples

### Generate Lidar Detection for Platform

Create a tracking scenario.

```
sc = trackingScenario;
rng(2020)% for repeatable results
```

Add an ego platform to the tracking scenario.

```
ego = platform(sc);
```

Add a target platform to the tracking scenario.

```
target = platform(sc);
```

Define a simple waypoint trajectory for the target.

```
traj = waypointTrajectory("Waypoints",[1 1 1; 2 2 2],"TimeOfArrival",[0,1]);
target.Trajectory = traj;
```

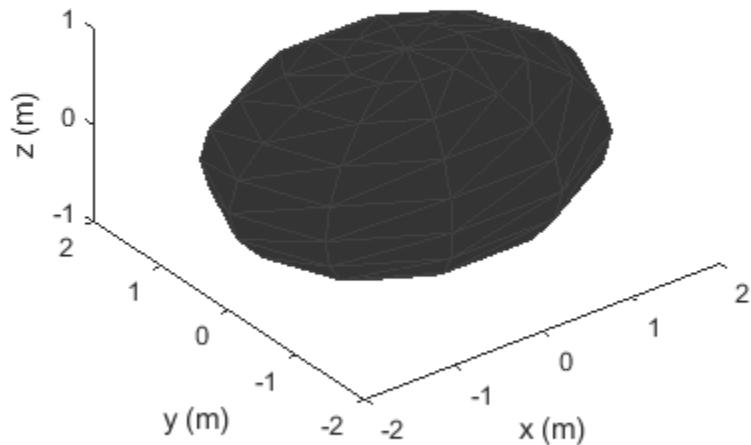
Define a sphere mesh for the target.

```
target.Mesh = extendedObjectMesh("Sphere");
target.Dimensions = struct("Length",4,"Width",3,"Height",2,"OriginOffset",[0 0 0]);
```

Show the mesh of the target.

```
figure()
show(target.Mesh);
legend("Target Mesh")
xlabel('x (m)'); ylabel('y (m)'); zlabel('z (m)');
```

 Target Mesh



Create two lidar sensors with different range accuracy. Mount them on the ego platform.

```
sensor1 = monostaticLidarSensor(1,"RangeAccuracy",0.01);
sensor2 = monostaticLidarSensor(2,"RangeAccuracy",0.1);
ego.Sensors = {sensor1;sensor2};
```

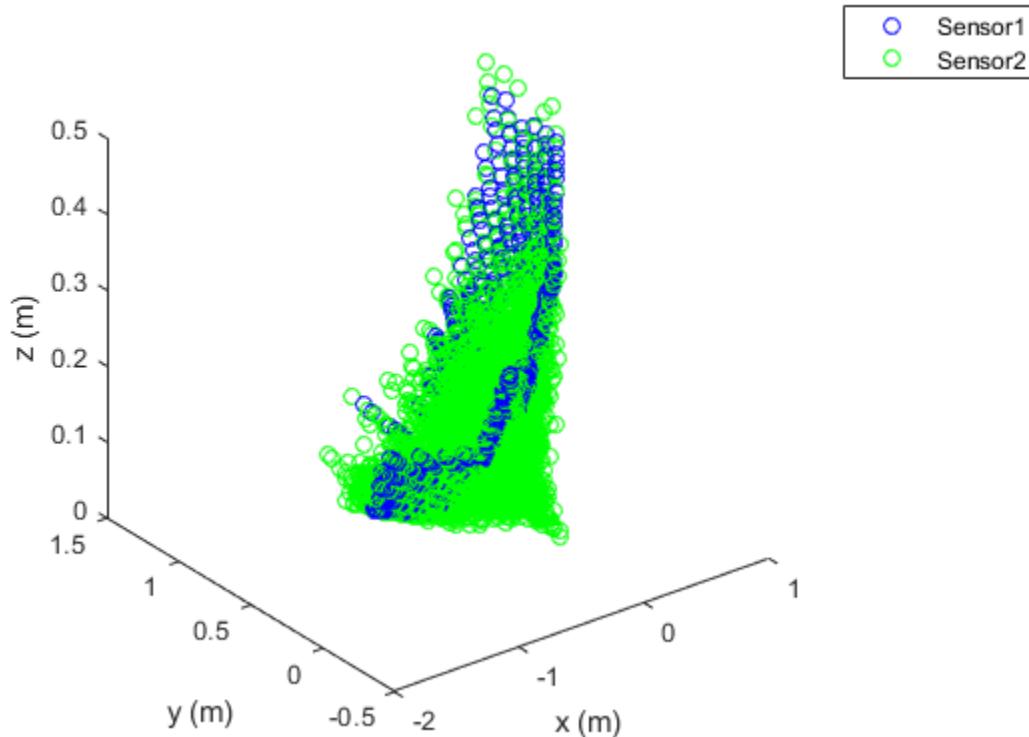
Generate detections from the two lidar sensor using `lidarDetect`.

```
[pointClouds,configs,clusters] = lidarDetect(ego,0);
```

Visualize the results.

```
cloud1 = pointClouds{1};
cloud2 = pointClouds{2};
figure()
plot3(cloud1(:,1),cloud1(:,2),cloud1(:,3), 'bo')
hold on
plot3(cloud2(:,1),cloud2(:,2),cloud2(:,3), 'go')
```

```
legend('Sensor1','Sensor2')
xlabel('x (m)'); ylabel('y (m)'); zlabel('z (m)')
```



## Input Arguments

### **plat — Ego platform**

Platform object

Ego platform, specified as a Platform object.

### **time — Current simulation time**

nonnegative scalar

Current simulation time, specified as a positive scalar in seconds.

Data Types: double

### **includeSelf — Enable sensors reporting ego platform's mesh detection**

false (default) | true

Enable reporting ego platform's mesh detection in the output, specified as true or false.

## Output Arguments

### **pointClouds — Detection point clouds**

*K*-element cell array

Point cloud detections generated by the sensors, returned as a *K*-element cell array. *K* is the number of `monostaticLidarSensor` objects mounted on the platform, `plat`. Each cell element is an array representing the point cloud generated by the corresponding sensor. The dimension of the array is determined by the `HasOrganizedOutput` property of the sensor.

- When this property is set as `true`, the cell element is returned an *N*-by-*M*-by-3 array of scalars, where *N* is the number of elevation channels, and *M* is the number of azimuth channels.
- When this property is set as `false`, the cell element is returned as an *P*-by-3 matrix of scalars, where *P* is the product of the numbers of elevation and azimuth channels.

The coordinate frame in which the point cloud locations are reported is determined by the `DetectionCoordinates` property of the sensor.

### **configs — Current sensor configurations**

*K*-element array of structure

Current sensor configurations, returned as a *K*-element array of structures. *K* is the number of `monostaticLidarSensor` objects mounted on the platform, `plat`. Each structure has these fields:

Field	Description
<code>SensorIndex</code>	Unique sensor index, returned as a positive integer.
<code>IsValidTime</code>	<code>IsValidTime</code> is <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-2 matrix of positive real values. The first row elements are the lower and upper azimuth limits; the second row elements are the lower and upper elevation limits.
<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.

Data Types: `struct`

### **clusters — Cluster labels of points**

*K*-element cell array

Cluster labels of points in the `pointClouds` output, returned as a *K*-element cell array. *K* is the number of `monostaticLidarSensor` objects mounted on the platform, `plat`. Each cell element is

an array representing cluster labels of points in the point cloud generated by the corresponding sensor. The dimension of the array is determined by the `HasOrganizedOutput` of the sensor.

- When this property is set as `true`, the cell element is returned as an  $N$ -by- $M$ -by-2 array of scalars, where  $N$  is the number of elevation channels, and  $M$  is the number of azimuth channels. On the third dimension, the first element represents the `PlatformID` of the target generating the point whereas the second element represents the `ClassID` of the target.
- When this property is set as `false`, the cell element is returned as a  $P$ -by-2 matrix of scalars, where  $P$  is the product of the numbers of elevation and azimuth channels. For each column of the matrix, the first element represents the `PlatformID` of the target generating the point whereas the second element represents the `ClassID` of the target.

## See Also

`extendedObjectMesh` | `monostaticLidarSensor` | `targetMeshes`

**Introduced in R2020b**

## Platform.targetPoses

Target positions and orientations as seen from platform

### Syntax

```
poses = targetPoses(ptfm)
```

### Description

`poses = targetPoses(ptfm)` returns the poses of all targets in a scenario with respect to the platform `ptfm`. Targets are defined as platforms as seen by another platform and are located with respect to the coordinate system of that platform. Pose represents the position, velocity, and orientation of a target with respect to the coordinate system belonging to the platform, `ptfm`. The targets must already exist in the tracking scenario. Add targets using the `platform` method.

### Input Arguments

#### `ptfm` — Scenario platform

Platform object

Scenario platform, specified as a `Platform` object. To create platforms, use the `platform` method.

### 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 input platform, `ptfm`, is not included. Pose consists of the position, velocity, orientation, and signature of a target in platform coordinates. The returned structure has these fields:

Field	Description
PlatformID	Unique identifier for the platform, specified as a scalar 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 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 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].

Field	Description
Acceleration	Acceleration of 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 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**

detect | emit | pose

**Introduced in R2018b**

# targetMeshes

Target meshes as seen from platform

## Syntax

```
tgtMeshes = targetMeshes(plat)
tgtMeshes = targetMeshes(plat,reportSelf)
tgtMeshes = targetMeshes(plat,reportSelf,format)
```

## Description

`tgtMeshes = targetMeshes(plat)` returns the relative pose and meshes of all other platforms seen from the ego platform, `plat`.

`tgtMeshes = targetMeshes(plat,reportSelf)` allows you to choose whether report mesh information about the ego platform, `plat`. Specify `reportSelf` as `true` or `false`.

`tgtMeshes = targetMeshes(plat,reportSelf,format)` allows you to specify the `format` of orientation as quaternion or rotation matrix in the `tgtMeshes` output.

## Examples

### Obtain Mesh of Target

Create a tracking scenario. Create an ego platform and a target platform.

```
scenario = trackingScenario;
ego = platform(scenario,'Position',[1 1 1]);
target = platform(scenario,'Trajectory',kinematicTrajectory('Position',[10 -3 0], 'Velocity',[5 0 0]),'Dimensions',[1 1 1]);
```

Define the target's mesh and adjust its dimensions.

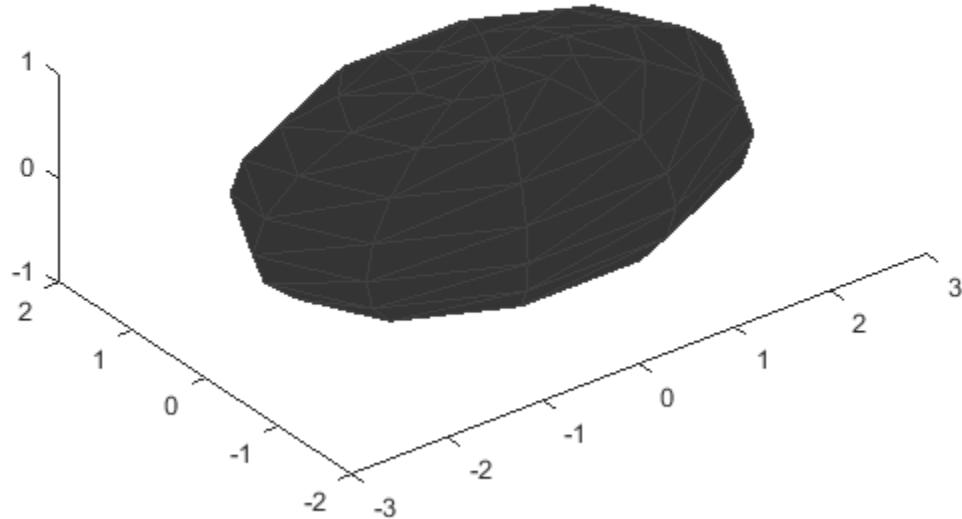
```
target.Mesh = extendedObjectMesh('sphere');
target.Dimensions.Length = 5;
target.Dimensions.Width = 3;
target.Dimensions.Height = 2;
```

Get the mesh of the target viewed from the ego platform.

```
tgtmeshes = targetMeshes(ego);
```

Show the derived mesh of the target from the view of the ego platform.

```
show(tgtmeshes.Mesh);
```



## Input Arguments

### plat — Ego platform

Platform object

Ego platform, specified as a Platform object.

### reportSelf — Enable reporting ego platform's mesh information

false (default) | true

Enable reporting ego platform's mesh information, specified as true or false.

### format — Orientation output format

'quaternion' (default) | 'rotmat'

Orientation output format, specified as 'quaternion' or 'rotmat'. When specified as 'quaternion', the orientation is given by quaternion. When specified as 'rotmat', the orientation output is given by rotation matrix.

## Output Arguments

### tgtMeshes — Pose and meshes of target platforms

array of structure

Pose and meshes of target platforms relative to the ego platform `plat`, returned as an array of structures. Each structure contains these fields:

Field Name	Description
PlatformID	Unique identifier of the target, specified as a nonnegative integer.
ClassID	Unique identifier of the class of the target, specified as a nonnegative integer.
Position	Position of the target with respect to the sensor mounting platform's body frame, specified as a 3-element vector of scalars.
Orientation	Orientation of the target with respect to the sensor mounting platform's body frame, specified as a <code>quaternion</code> object or a rotation matrix.
Mesh	Geometric mesh of the target, specified as an <code>extendedObjectMesh</code> object with respect to the target's body frame.

## See Also

`lidarDetect` | `monostaticLidarSensor`

**Introduced in R2020b**

# Platform.pose

Pose of platform

## Syntax

```
pse = pose(ptfm)
pse = pose(ptfm,type)
pse = pose(___, 'CoordinateSystem', coordinate)
```

## Description

`pse = pose(ptfm)` returns the estimated pose, `pse`, of the platform `ptfm`, in scenario coordinates. The platform must already exist in the tracking scenario. Add platforms to a scenario using the `platform` method. The pose is estimated by a pose estimator specified in the `PoseEstimator` property of the platform.

`pse = pose(ptfm,type)` specifies the type of the pose estimation as '`estimated`' or '`true`'.

`pse = pose(___, 'CoordinateSystem', coordinate)` specifies the coordinate system of the `pse` output. You can only use this syntax when the `IsEarthCentered` property of the tracking scenario is set to `true`.

## Examples

### Create Earth Centered Scenario

Create a tracking scenario with specified updated rate.

```
scene = trackingScenario('IsEarthCentered', true, 'UpdateRate', 0.01);
```

Add an airplane in the scenario. The trajectory of the airplane changes in latitude and altitude.

```
plane = platform(scene, 'Trajectory', geoTrajectory([-12.338, -71.349, 10600; 42.390, -71.349, 0], [0 360 0 360 0 360]));
```

Advance the tracking scenario and record the geodetic and Cartesian positions of the plane target.

```
positions = [];
while advance(scene)
    poseLLA = pose(plane, 'CoordinateSystem', 'Geodetic');
    poseCart = pose(plane, 'CoordinateSystem', 'Cartesian');
    positions = [positions;poseCart.Position]; %#ok<AGROW> Allow the buffer to grow.
end
```

Visualize the trajectory in the ECEF frame.

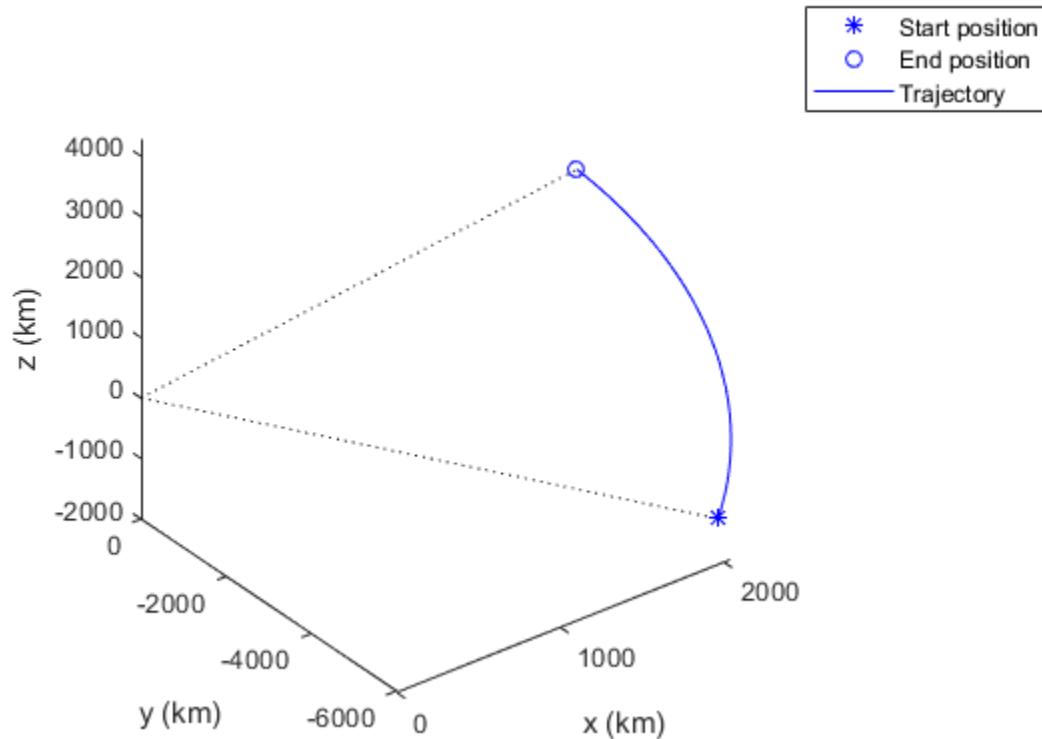
```
figure()
km = 1000;
% Plot the trajectory.
plot3(positions(1,1)/km,positions(1,2)/km,positions(1,3)/km, 'b*');
hold on;
plot3(positions(end,1)/km,positions(end,2)/km,positions(end,3)/km, 'bo');
```

```

plot3(positions(:,1)/km,positions(:,2)/km,positions(:,3)/km,'b');

% Plot the Earth radial lines.
plot3([0 positions(1,1)]/km,[0 positions(1,2)]/km,[0 positions(1,3)]/km,'k:');
plot3([0 positions(end,1)]/km,[0 positions(end,2)]/km,[0 positions(end,3)]/km,'k:');
xlabel('x (km)'); ylabel('y (km)'); zlabel('z (km)');
legend('Start position','End position','Trajectory')

```



## Input Arguments

### **ptfm – Scenario platform**

Platform object

Scenario platform, specified as a Platform object. To create platforms, use the `platform` method.

### **type – Source of platform pose information**

'estimated' (default) | 'true'

Source of platform pose information, specified as 'estimated' or 'true'. When set to 'estimated', the pose is estimated using the pose estimator specified in the `PoseEstimator` property of the tracking scenario. When 'true' is selected, the true pose of the platform is returned.

Example: 'true'

Data Types: char

**coordinate — Coordinate system to report pose**

'Cartesian' (default) | 'Geodetic'

Coordinate system to report pose, specified as:

- '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 (North-East-Down by default) corresponding to the current waypoint.

You can only use this argument when the `IsEarthCentered` property of the tracking scenario is set to `true`.

## Output Arguments

**pse — Pose of platform**

structure

Pose of platform, returned as a structure. Pose consists of the position, velocity, orientation, and angular velocity of the platform with respect to scenario coordinates. The returned 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	<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', the <code>Position</code> is the 3-element Cartesian position coordinates in meters.</li> <li>• If the <code>coordinateSystem</code> argument is specified as 'Geodetic', the <code>Position</code> is the 3-element 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].

Field	Description
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 quaternion(1, 0, 0, 0).
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].

**See Also**

[detect](#) | [emit](#) | [targetPoses](#)

**Introduced in R2018b**

# platform

Add platform to tracking scenario

## Syntax

```
ptfm = platform(sc)
ptfm = platform(sc,Name,Value)
```

## Description

`ptfm = platform(sc)` adds a `Platform` object, `ptfm`, to the tracking scenario, `sc`. The function creates a platform with default property values. Platforms are defined as points or cuboids with aspect-dependent properties. Each platform is automatically assigned a unique ID specified in the `platformID` field of the `Platform` object.

`ptfm = platform(sc,Name,Value)` adds a platform with additional properties specified by 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.

## Input Arguments

### sc — Tracking scenario

`trackingScenario` object

Tracking scenario, specified as a `trackingScenario` object.

### 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`.

### 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`

### Position — Position of platform

3-element vector of scalar

This property is read-only.

Current position of the platform, specified as a 3-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.

You should only specify position when creating a stationary platform. If you choose to specify the trajectory of the platform, do not use `Position`. Instead, use the `Trajectory` argument.

Data Types: double

### **Orientation — Orientation of platform**

3-element vector of scalar

This property is read-only.

Orientation of the platform, specified as a 3-element vector of scalars in degrees. The three scalars are the `[yaw, pitch, roll]` rotation angles from the local reference frame to the platform's body frame.

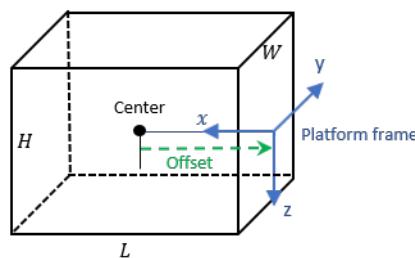
You should only specify `Orientation` when creating a stationary platform. If you choose to specify the orientation over time, use the `Trajectory` argument.

Data Types: double

### **Dimensions — Platform dimensions and origin offset**

struct

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 following 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 either 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`, you can use the `kinematicTrajectory` or the `waypointTrajectory` object. By default, a stationary `kinematicTrajectory` object is used.
- When the `IsEarthCentered` property of the scenario is set to `true`, you can only use the `geoTrajectory` object. By default, a stationary `geoTrajectory` object is used.

## Signatures — Platform signatures

`{rcsSignature irSignature tsSignature}` (default) | cell array of signature objects

Platform signatures, specified as a cell array of `irSignature`, `rcsSignature`, and `tsSignature` objects or an empty cell array. The cell array contains at most only one instance for each type of signature objects listed. 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` System object (default) | pose estimator object

A pose estimator, specified as a pose estimator object. The pose estimator determines platform pose with respect to the local NED scenario coordinate. The interface of any pose estimator must match the interface of `insSensor`. By default, pose estimator accuracy properties are set 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` or `sonarEmitter`.

**Sensors — Sensors mounted on platform**

cell array of sensor objects

Sensors mounted on platform, specified as a cell array of sensor objects such as `irSensor`, `radarSensor`, `monostaticRadarSensor`, or `sonarSensor`.

## Output Arguments

**ptfm — Scenario platform**

Platform object

Scenario platform, returned as a `Platform` object.

## Examples

**Platform Follows Circular Trajectory**

Create a tracking scenario and a platform following a circular path.

```
scene = trackingScenario('UpdateRate',1/50);

% Create a platform
plat = platform(scene);

% Follow a circular trajectory 1 km in radius completing in 400 hundred seconds.
plat.Trajectory = waypointTrajectory('Waypoints', [0 1000 0; 1000 0 0; 0 -1000 0; -1000 0 0; 0 1000 0]);
plat.Trajectory = waypointsSmooth(plat.Trajectory);
plat.Trajectory = waypointsInterpolate(plat.Trajectory, 100);

% Perform the simulation
while scene.advance
    p = pose(plat);
    fprintf('Time = %f ', scene.SimulationTime);
    fprintf('Position = [');
    fprintf('%f ', p.Position);
    fprintf('] Velocity = [');
    fprintf('%f ', p.Velocity);
    fprintf(']\n');
end

Time = 50.000000
Position = [
707.095476 707.100019 0.000000
] Velocity = [
11.107152 -11.107075 0.000000
]
Time = 100.000000
Position = [
1000.000000 0.000000 0.000000
]
```

```
] Velocity = [  
0.000476 -15.707961 0.000000  
]  
Time = 150.000000  
Position = [  
707.115558 -707.115461 0.000000  
] Velocity = [  
-11.107346 -11.107341 0.000000  
]  
Time = 200.000000  
Position = [  
0.000000 -1000.000000 0.000000  
] Velocity = [  
-15.707963 0.000460 0.000000  
]  
Time = 250.000000  
Position = [  
-707.098004 -707.098102 0.000000  
] Velocity = [  
-11.107069 11.107074 0.000000  
]  
Time = 300.000000  
Position = [  
-1000.000000 0.000000 0.000000  
] Velocity = [  
-0.000476 15.707966 0.000000  
]  
Time = 350.000000  
Position = [  
-707.118086 707.113543 0.000000  
] Velocity = [  
11.107262 11.107340 0.000000
```

```

]

Time = 400.000000

Position = [
-0.000000 1000.000000 0.000000

] Velocity = [
15.708226 -0.000493 0.000000

]

```

### Cuboid Platforms Follow Circular Trajectories

Create a tracking scenario with two cuboid platforms following circular trajectories.

```

sc = trackingScenario;

% Create the platform for a truck with dimension 5 x 2.5 x 3.5 (m).
p1 = platform(sc);
p1.Dimensions = struct('Length',5,'Width',2.5,'Height',3.5,'OriginOffset',[0 0 0]);

% Specify the truck's trajectory as a circle with radius 20 meters.
p1.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 dimension .3 x .3 x .1 (m).
p2 = platform(sc);
p2.Dimensions = struct('Length',.3,'Width',.3,'Height',.1,'OriginOffset',[0 0 0]);

% The quadcopter follows the truck at 10 meteres above with small angular delay.
% Note that the negative z coordinates correspond to positive elevation.
p2.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`.

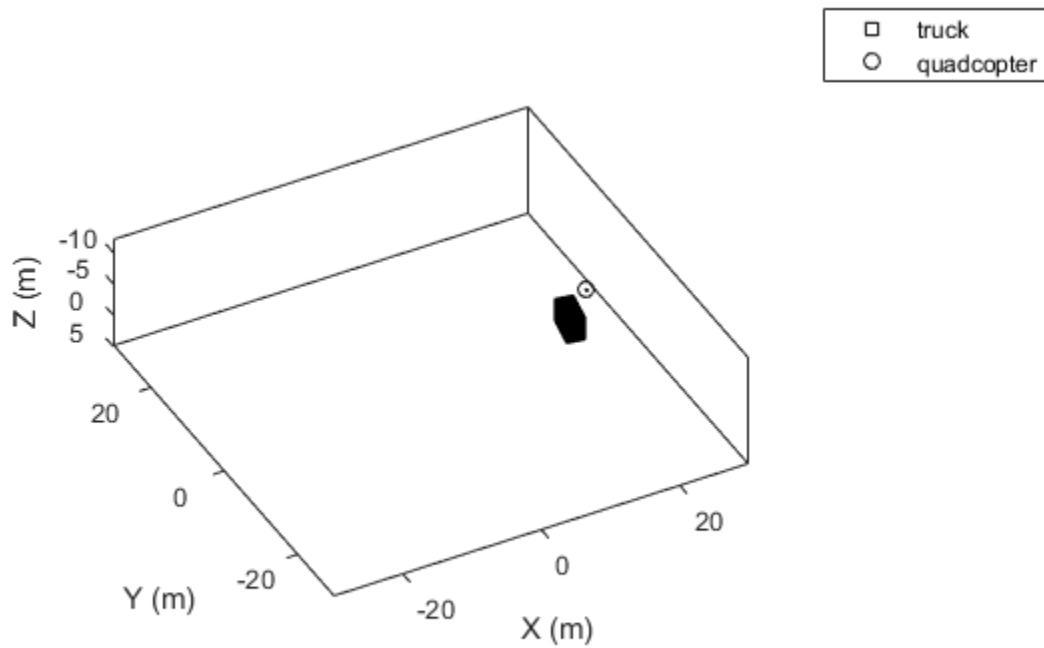
```

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 animate.
view(-28,37);
set(gca, 'Zdir', 'reverse');

while advance(sc)
    poses = platformPoses(sc);
    plotPlatform(pp1, poses(1).Position, p1.Dimensions, poses(1).Orientation);
    plotPlatform(pp2, poses(2).Position, p2.Dimensions, poses(2).Orientation);
end

```



## See Also

**Objects**

Platform

**System Objects**

kinematicTrajectory | waypointTrajectory

**Introduced in R2018b**

# advance

Advance tracking scenario simulation by one time step

## Syntax

```
isrunning = advance(sc)
```

## Description

`isrunning = advance(sc)` advances the tracking scenario simulation, `sc`, by one time step. To specify the step time, set the `UpdateRate` property of the `trackingScenario` object. The function returns the status, `isrunning`, of the simulation. `advance` updates a platform location only if the platform has an assigned path. You can generate assigned paths using the `Motion` property of a platform. To update platforms that have no assigned paths, you can set the `Position`, `Velocity`, `Orientation`, or `AngularVelocity` properties at any time during the simulation.

## Input Arguments

### **sc — Tracking scenario**

`trackingScenario` object

Tracking scenario, specified as a `trackingScenario` object.

## Output Arguments

### **isrunning — Run-state of simulation**

`0 | 1`

The run-state of the simulation, returned as `0` or `1`. If `isrunning` is `1`, the simulation is running. If `isrunning` is `0`, 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 using waypoints (with a `waypointTrajectory` object).

Units are in seconds.

### Introduced in R2018b

# detect

Collect detections from all the sensors in tracking scenario

## Syntax

```
detections = detect(sc)
detections = detect(sc,signals)
detections = detect(sc,signals,configs)
[detections,sensorConfigs] = detect(___)
[detections,sensorConfigs,configIDS] = detect(___)
```

## Description

`detections = detect(sc)` reports the detections from all sensors mounted on every platform in the tracking scenario, `sc`.

---

**Tip** Use this syntax only when none of the sensors requires knowledge of the signals present in the scenario. For example, the `HasInterference` property of `monostaticRadarSensor` is set to `false`.

---

`detections = detect(sc,signals)` reports the detections from all sensors when at least one sensor requires the knowledge of signals in the scenario. For example, when a `radarSensor` is operating in an ESM mode.

`detections = detect(sc,signals,configs)` reports the detections from all sensors when at least one sensor also requires the knowledge of emitter configurations in the scenario. For example, when a `radarSensor` is configured as a monostatic radar.

`[detections,sensorConfigs] = detect(__)` additionally returns the configurations of each sensor at the detection time.

`[detections,sensorConfigs,configIDS] = detect(__)` additionally returns all platform IDs corresponding to the sensor configurations, `sensorConfigs`.

## Examples

### Obtain Detections from Two Platforms in Tracking Scenario

Create a tracking scenario.

```
s = rng(0); % For repeatable result
ts = trackingScenario('UpdateRate',1);
```

Create the first platform and mount one emitter and one sensor on it.

```
plat1 = platform(ts);
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(ts);
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 tracking scenario, transmit and propagate emissions, and collect signals using the `detect` function.

```
advance(ts);
[emtx,emitterConfs,emitterConfPIDs] = emit(ts); % Transmitted emissions
emprop = propagate(ts,emtx,'HasOcclusion',true); % Propagate emissions
[dets,sensorConfs,sensorConfPIDs] = detect(ts,emprop,emitterConfs);
```

Display the detection results: Sensor 1 on platform 1 detected platform 2.

```
disp(dets{1})
objectDetection with properties:

    Time: 1
    Measurement: [3x1 double]
    MeasurementNoise: [3x3 double]
    SensorIndex: 1
    ObjectClassID: 0
    MeasurementParameters: [1x1 struct]
    ObjectAttributes: {[1x1 struct]}

sensor = dets{1}.SensorIndex
sensor = 1
detectedPlatform = dets{1}.ObjectAttributes{1}.TargetIndex
detectedPlatform = 2
rng(s) % Return the random number generator to its previous state
```

## Input Arguments

### **sc — Tracking scenario**

trackingScenario object

Tracking scenario, specified as a `trackingScenario` object.

### **signals — Signal emissions**

cell array of signal emission object

Signal emissions, specified as a cell array of signal emission objects, such as `radarEmission` and `sonarEmission`.

**configs — Emitter configurations**

array of emitter configuration structures

Emitter configurations, specified as an array of emitter configuration structures. The fields of each structure are:

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.

## 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 structure

Sensor configurations, return as an array of sensor configuration structures. The fields of each structure are:

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, [azfov;elfov]. azfov and elfov 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.

**configIDs — Platform IDs for sensor configurations**

array of positive integer

Platform IDs for sensor configurations in the sensorConfigs output, returned as an array of positive integers.

**See Also**

[emit](#) | [objectDetection](#) | [propagate](#) | [record](#) | [trackingScenario](#)

**Introduced in R2020a**

# emit

Collect emissions from emitters in tracking scenario

## Syntax

```
emissions = emit(sc)
[emissions,configs] = emit(sc)
[emissions,configs,platformIDs] = emit(sc)
```

## Description

`emissions = emit(sc)` reports signals emitted from all the emitters mounted on platforms in the tracking scenario `sc`.

`[emissions,configs] = emit(sc)` also returns the configurations of all the emitters at the emission time.

`[emissions,configs,platformIDs] = emit(sc)` also returns the IDs of platforms on which the emitters are mounted.

## Examples

### Obtain Emissions from Platforms in Tracking Scenario

Create a tracking scenario and add two platforms. Set the position of each platform and add an emitter.

```
ts = trackingScenario('UpdateRate',1);
plat1 = platform(ts);
plat1.Trajectory.Position = [0,0,0];
emitter1 = radarEmitter(1,'UpdateRate',1);
plat1.Emitters = emitter1;
plat2 = platform(ts);
plat2.Trajectory.Position = [100,0,0];
emitter2 = radarEmitter(2,'UpdateRate',1);
plat2.Emitters = emitter2;
```

Advance the tracking scenario and generate emissions.

```
advance(ts);
[emissions, configs, sensorConfigPIDs] = emit(ts);
```

Print the results.

```
disp("There are " + numel(emissions) + " emissions.");
```

There are 2 emissions.

The first emission is:

```
disp(emissions{1});
```

```
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
```

The second emission is:

```
disp(emissions{2});
```

```
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
```

The emitter configuration associated with the first emission is:

```
disp(configs(1));
```

```
    EmitterIndex: 1
    IsValidTime: 1
    IsScanDone: 0
    FieldOfView: [1 5]
    MeasurementParameters: [1x1 struct]
```

The emitter configuration associated with the second emission is:

```
disp(configs(2));
```

```
    EmitterIndex: 2
    IsValidTime: 1
    IsScanDone: 0
    FieldOfView: [1 5]
    MeasurementParameters: [1x1 struct]
```

The emitter configurations are connected with platform IDs:

```
1 disp(sensorConfigPIIDs');
2
```

## Input Arguments

### sc — Tracking scenario

trackingScenario object

Tracking scenario, specified as a `trackingScenario` object.

## Output Arguments

### emissions — Emissions of all emitters

cell array of emission objects

Emissions of all emitters in the tracking scenario, returned as a cell array of emission objects, such as `radarEmission` and `sonarEmission` objects.

### configs — Configuration of emitters

array of emitter configuration structures

Configuration of all the emitters in the tracking scenario, returned as an array of emitter configuration structures. The fields of each structure are:

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.

### platformIDs — Platform IDs

vector of positive integers

Platform IDs, returned as a vector of positive integers. The order of platform IDs output is the same as that of the `configs` output.

**See Also**

`radarEmitter` | `radarSensor` | `sonarEmitter` | `sonarSensor` | `trackingScenario`

**Introduced in R2020a**

# clone

Create copy of tracking scenario

## Syntax

```
newScenario = clone(scenario)
```

## Description

`newScenario = clone(scenario)` creates a copy of the `trackingScenario` scenario.

## Examples

### Create Copy of Tracking Scenario

Create a tracking scenario object.

```
scene = trackingScenario;
```

Create a copy of the scenario, `scene`.

```
newScene = clone(scene)
```

```
newScene =
    trackingScenario with properties:
```

```
    IsEarthCentered: 0
        UpdateRate: 10
    SimulationTime: 0
        StopTime: Inf
    IsRunning: 1
    Platforms: {}
```

## Input Arguments

**scenario — Tracking scenario**  
`trackingScenario` object

Tracking scenario, specified as a `trackingScenario` object.

## Output Arguments

**newScenario — Copy of tracking scenario**  
`trackingScenario` object

Copy of tracking scenario, returned as a `trackingScenario` object.

**See Also**

`trackingScenario`

**Introduced in R2020b**

# perturb

Apply perturbations to tracking scenario

## Syntax

```
perturb(scene)
```

## Description

`perturb(scene)` perturbs the baseline tracking scenario, `scene`, according to the perturbations defined on objects (such as trajectories, sensors, and platforms) in the `scene`. Use the `perturbations` function to define property perturbations on each object.

## Examples

### Tracking Scenario Perturbation

Create a tracking scenario and add a platform.

```
scenario = trackingScenario;
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.

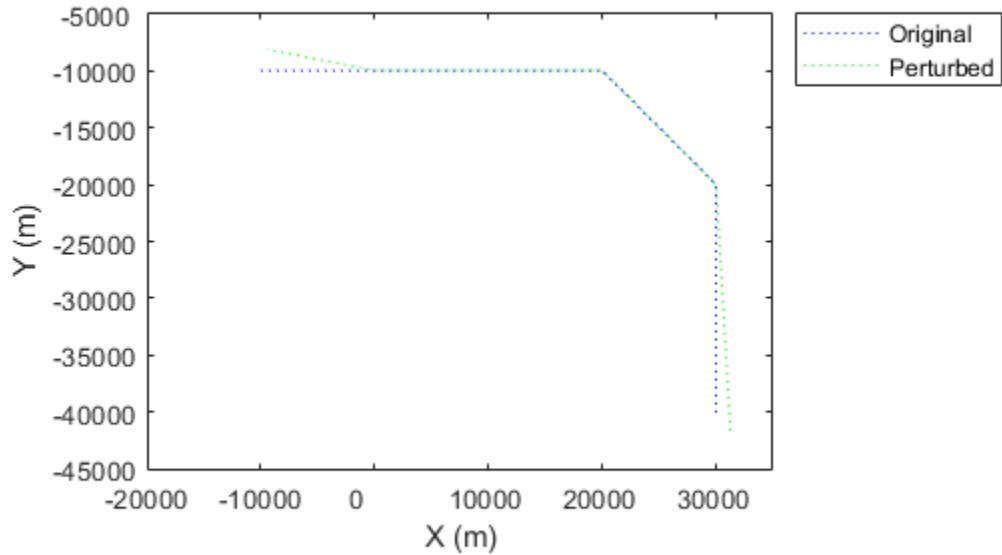
```
tp = theaterPlot("XLimits", [-20 35]*1e3, "YLimits", [-45 -5]*1e3);
trajPlotter1 = trajectoryPlotter(tp, 'DisplayName', 'Original', 'Color', 'b');
plotTrajectory(trajPlotter1, {p.Trajectory.Waypoints});
```

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

**scene — Tracking scenario**  
trackingScenario object

Tracking scenario, specified as a trackingScenario object.

## See Also

[perturbations](#)

**Introduced in R2020b**

# platformPoses

Positions, velocities, and orientations of all platforms in tracking scenario

## Syntax

```
poses = platformPoses(sc)
poses = platformPoses(sc,fmt)
poses = platformPoses(___, 'CoordinateSystem', coordinate)
```

## Description

`poses = platformPoses(sc)` returns the current poses for all platforms in the tracking scenario, `sc`. Pose is the position, velocity, and orientation of a platform relative to scenario coordinates. Platforms are `Platform` objects.

`poses = platformPoses(sc,fmt)` also specifies the format, `fmt`, of the returned platform orientation.

`poses = platformPoses(___, 'CoordinateSystem', coordinate)` specifies the coordinate system of the poses output. You can only use this syntax only when the `IsEarthCentered` property of the tracking scenario, `sc`, is set to `true`.

## Input Arguments

### sc — Tracking scenario

`trackingScenario` object

Tracking scenario, specified as a `trackingScenario` object.

### fmt — 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.

Example: `'rotmat'`

Data Types: `char`

### coordinate — Coordinate system to report poses

`'Cartesian'` (default) | `'Geodetic'`

Coordinate system to report poses, specified as:

- `'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.

You can only use this argument when the `IsEarthCentered` property of the tracking scenario, `sc`, is set to `true`.

## Output Arguments

### **poses — Platform poses in scenario coordinates**

structures | array of structures

Poses of all platforms in the tracking scenario, returned as a structure or array of structures. The pose 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 '<code>Cartesian</code>', the <code>Position</code> is the 3-element Cartesian position coordinates in meters.</li> <li>If the <code>coordinateSystem</code> argument is specified as '<code>Geodetic</code>', the <code>Position</code> is the 3-element 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

**Introduced in R2018b**

# platformProfiles

Profiles of platforms in tracking scenario

## Syntax

```
profiles = platformProfiles(sc)
```

## Description

`profiles = platformProfiles(sc)` returns the profiles of all platforms in the tracking scenario, `sc`.

## Input Arguments

### **sc — Tracking scenario**

trackingScenario object

Tracking scenario, specified as a `trackingScenario` object.

## Output Arguments

### **profiles — Platform profiles**

array of structures

Profiles of all platforms in the tracking scenario, returned as an array of structures. The number of structures in the array is equal to the number platforms. Each profile contains the signatures of a platform and identifying information. The 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
Signatures	Platform signatures defined as a cell array of radar cross-section ( <code>rcsSignature</code> ), IR emission pattern ( <code>irSignature</code> ), and sonar target strength ( <code>tsSignature</code> ) objects.

See `Platform` for more completed definitions of the fields.

## Examples

### Generate Platform Profiles from Tracking Scenario

Create a tracking scenario.

```
scene = trackingScenario;
```

Add two platforms to the tracking scenario. Specify the `ClassID` of the second platform as 3.

```
p1 = platform(scene);  
p2 = platform(scene);  
p2.ClassID = 3;
```

Extract the profiles for platforms in the scene.

```
profiles = platformProfiles(scene)  
  
profiles=1x2 struct array with fields:  
    PlatformID  
    ClassID  
    Signatures
```

## See Also

[Platform](#) | [trackingScenario](#)

**Introduced in R2018b**

# propagate

Propagate emissions in tracking scenario

## Syntax

```
propEmissions = propagate(sc,emissions)
propEmissions = propagate(sc,emissions,'HasOcclusion',tfOcclusion)
```

## Description

`propEmissions = propagate(sc,emissions)` returns propagated emissions that are a combination of the input emissions and the reflections of these input emissions from the platforms in the tracking scenario `sc`.

`propEmissions = propagate(sc,emissions,'HasOcclusion',tfOcclusion)` specifies whether the radar channel models occlusion or not. By default, `tfOcclusion` is set to `true`.

## Examples

### Propagate Emissions from Platforms in Tracking Scenario

Create a tracking scenario and add two platforms. Set the position of each platform and add an emitter.

```
ts = trackingScenario('UpdateRate',1);
plat1 = platform(ts);
plat1.Trajectory.Position = [0,0,0];
emitter1 = radarEmitter(1,'UpdateRate',1);
plat1.Emitters = emitter1;
plat2 = platform(ts);
plat2.Trajectory.Position = [100,0,0];
emitter2 = radarEmitter(2,'UpdateRate',1);
plat2.Emitters = emitter2;
```

Advance the tracking scenario, generate emissions, and obtain propagated emissions.

```
advance(ts);
emtx = emit(ts); % Get emissions
emprop = propagate(ts, emtx, 'HasOcclusion', true)

emprop=3x1 cell array
    {1x1 radarEmission}
    {1x1 radarEmission}
    {1x1 radarEmission}
```

The last emission was 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

### **sc — Tracking scenario**

trackingScenario object

Tracking scenario, specified as a trackingScenario object.

### **emissions — Emissions in the tracking scenario**

cell array of emission objects

Emissions in the tracking scenario, specified as a cell array of emission objects, such as radarEmission and sonarEmission objects. You can obtain emissions from a tracking scenario using the emit function.

### **tfOcclusion — Indicate HasOcclusion status**

true (default) | false

Indicate HasOcclusion status, specified as true or false.

## Output Arguments

### **propEmissions — Propagated emissions**

cell array of emission objects

Propagated emissions in the tracking scenario, specified as a cell array of emission objects, such as radarEmission and sonarEmission objects. The propagated emissions contain the source emissions and the emissions reflected from the platforms.

## See Also

[emit](#) | [radarChannel](#) | [trackingScenario](#) | [underwaterChannel](#)

## Introduced in R2020a

## lidarDetect

Report point cloud detections from all lidar sensor in `trackingScenario`

### Syntax

```
pointClouds = lidarDetect(scene)
[pointClouds,configs] = lidarDetect(scene)
[pointClouds,configs, clusters] = lidarDetect(____)
```

### Description

`pointClouds = lidarDetect(scene)` reports point cloud detections from all `monostaticLidarSensor` objects mounted on every platform in the `trackingScenario`, `scene`.

`[pointClouds,configs] = lidarDetect(scene)` also returns the configurations of the sensors, `configs`, in the tracking scenario.

`[pointClouds,configs, clusters] = lidarDetect(____)` also returns `clusters`, the cluster labels for each point in the point cloud detections.

### Examples

#### Generate Lidar Detection in Tracking Scenario

Create a tracking scenario.

```
sc = trackingScenario;
rng(2020) % for repeatable results
```

Add two platforms to the tracking scenario.

```
plat1 = platform(sc);
plat2 = platform(sc);
```

Add a target platform to the tracking scenario.

```
target = platform(sc);
```

Define a simple waypoint trajectory for the target.

```
traj = waypointTrajectory("Waypoints",[1 1 1; 2 2 2],"TimeOfArrival",[0,1]);
target.Trajectory = traj;
```

Define a sphere mesh for the target.

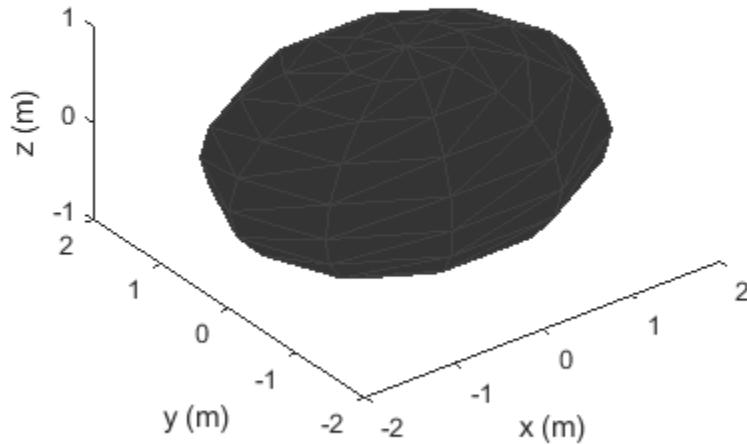
```
target.Mesh = extendedObjectMesh("Sphere");
target.Dimensions = struct("Length",4,"Width",3,"Height",2,"OriginOffset",[0 0 0]);
```

Show the mesh of the target.

```
figure()
show(target.Mesh);
```

```
legend("Target Mesh")
xlabel('x (m)'); ylabel('y (m)'); zlabel('z (m)');
```

 Target Mesh



Create two lidar sensors with different range accuracy. Mount them on the two platforms.

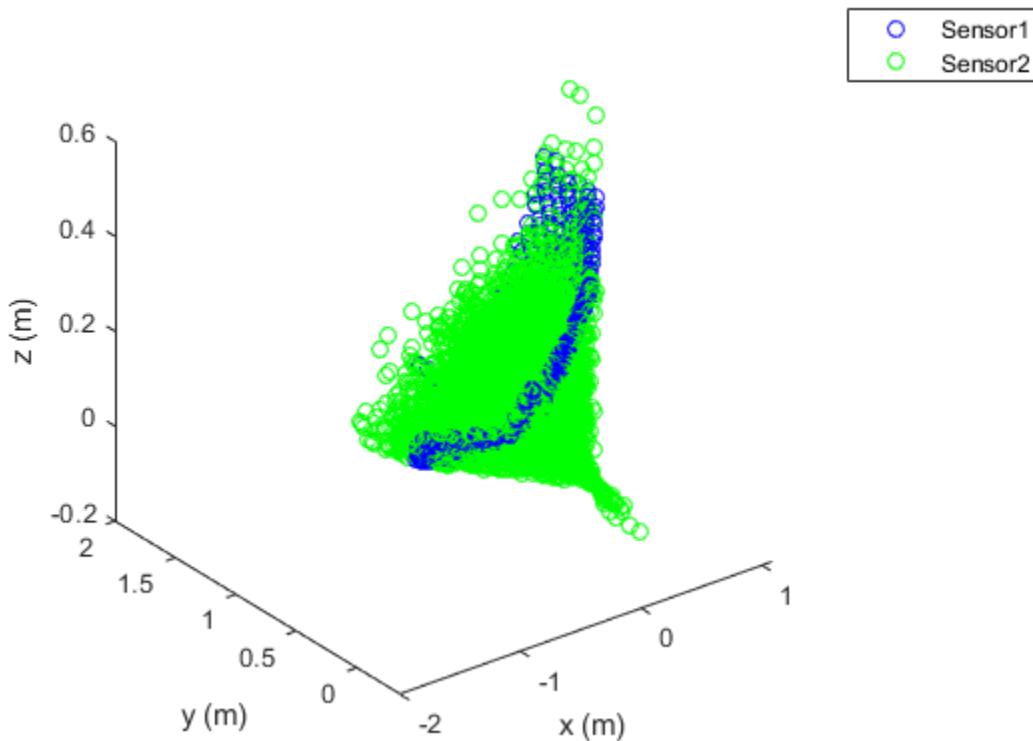
```
sensor1 = monostaticLidarSensor(1,"RangeAccuracy",0.01);
sensor2 = monostaticLidarSensor(2,"RangeAccuracy",0.2);
plat1.Sensors = {sensor1};
plat2.Sensors = {sensor2};
```

Generate detections from the two lidar sensor using `lidarDetect`.

```
[pointClouds,configs,clusters] = lidarDetect(sc);
```

Visualize the results.

```
cloud1 = pointClouds{1};
cloud2 = pointClouds{2};
figure()
plot3(cloud1(:,1),cloud1(:,2),cloud1(:,3), 'bo')
hold on
plot3(cloud2(:,1),cloud2(:,2),cloud2(:,3), 'go')
legend('Sensor1','Sensor2')
xlabel('x (m)'); ylabel('y (m)'); zlabel('z (m)')
```



## Input Arguments

### **scene — Tracking scenario**

trackingScenario object

Tracking scenario, specified as a trackingScenario object.

## Output Arguments

### **pointClouds — Detection point clouds**

K-element cell array

Point cloud detections generated by the sensors, returned as a  $K$ -element cell array.  $K$  is the number of monostaticLidarSensor objects in the tracking scenario, `scene`. Each cell element is an array representing the point cloud generated by the corresponding sensor. The dimension of the array is determined by the `HasOrganizedOutput` property of the sensor.

- When this property is set as `true`, the cell element is returned an  $N$ -by- $M$ -by-3 array of scalars, where  $N$  is the number of elevation channels, and  $M$  is the number of azimuth channels.
- When this property is set as `false`, the cell element is returned as an  $P$ -by-3 matrix of scalars, where  $P$  is the product of the numbers of elevation and azimuth channels.

The coordinate frame in which the point cloud locations are reported is determined by the `DetectionCoordinates` property of the sensor.

**configs — Current sensor configurations**

K-element array of structure

Current sensor configurations, returned as a  $K$ -element array of structures.  $K$  is the number of `monostaticLidarSensor` objects in the tracking scenario, `scene`. Each 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-2 matrix of positive real values. The first row elements are the lower and upper azimuth limits; the second row elements are the lower and upper elevation limits.
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`**clusters — Cluster labels of points**

K-element cell array

Cluster labels of points in the `pointClouds` output, returned as a  $K$ -element cell array.  $K$  is the number of `monostaticLidarSensor` in the tracking scenario, `scene`. Each cell element is an array representing cluster labels of points in the point cloud generated by the corresponding sensor. The dimension of the array is determined by the `HasOrganizedOutput` of the sensor.

- When this property is set as `true`, the cell element is returned as an  $N$ -by- $M$ -by-2 array of scalars, where  $N$  is the number of elevation channels, and  $M$  is the number of azimuth channels. On the third dimension, the first element represents the `PlatformID` of the target generating the point, and the second element represents the `ClassID` of the target.
- When this property is set as `false`, the cell element is returned as a  $P$ -by-2 matrix of scalars, where  $P$  is the product of the numbers of elevation and azimuth channels. For each column of the matrix, the first element represents the `PlatformID` of the target generating the point whereas the second element represents the `ClassID` of the target.

**See Also**`extendedObjectMesh` | `monostaticLidarSensor` | `targetMeshes`**Introduced in R2020b**

## record

Run tracking scenario and record platform, sensor, and emitter information

### Syntax

```
rec = record(sc)
rec = record(sc,format)
rec = record(___,Name,Value)
```

### Description

`rec = record(sc)` returns a record, `rec`, of the evolution of the tracking scenario simulation, `sc`. 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 scenario's `StopTime` is reached or any platform in the scenario has finished its trajectory specified by the `Trajectory` property.

`rec = record(sc,format)` also specifies the format, `format`, of the returned platform orientation.

`rec = record(___,Name,Value)` specifies additional recording quantities using name-value pairs. Enclose each `Name` in quotes.

### Input Arguments

#### **sc — Tracking scenario**

`trackingScenario` object

Tracking scenario, specified as a `trackingScenario` 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.

Example: `'rotmat'`

Data Types: char

#### **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`.

#### **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 `trackingScenarioRecording` object.

**CoordinateSystem — Coordinate system to report recorded poses**

`'Cartesian'` (default) | `'Geodetic'`

Coordinate system to report recorded positions, specified as:

- `'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.

You can only use this argument when the `IsEarthCentered` property of the tracking scenario, `sc`, is set to `true`.

## Output Arguments

**rec — Records of platform states during simulation**

`M`-by-1 array of structures | `trackingScenarioRecording` 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 `trackingScenarioRecording` 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 input.

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 `Poses` structure 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 'Cartesian', the <code>Position</code> is the 3-element Cartesian position coordinates in meters.</li> <li>If the <code>coordinateSystem</code> argument is specified as 'Geodetic', the <code>Position</code> is the 3-element 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 [0 0 0].
<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 [0 0 0].
<code>Orientation</code>	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> .
<code>AngularVelocity</code>	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].

The optional fields in the `rec` output are:

Field	Description
Emissions	a cell array of emissions (such as <code>radarEmission</code> and <code>sonarEmission</code> ) in the scenario
EmitterConfigurations	a struct array of emitter configurations for each emitter
EmitterPlatformIDs	a numeric array of platform IDs for each emitter
Detections	a cell array of <code>objectDetection</code> objects generated by the sensors in the scenario
SensorConfigurations	a struct array of sensor configurations for each sensor
SensorPlatformIDs	a numeric array of platform IDs for each sensor
CoverageConfig	a struct array of coverage configurations for each sensor or emitter

Each emitter configuration structure contains the following 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 the following 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.

<b>IsScanDone</b>	<b>IsScanDone</b> is <code>true</code> when the sensor has completed a scan.
<b>FieldOfView</b>	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.
<b>MeasurementParameters</b>	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:

## Fields of configurations

Field	Description
Index	A unique integer to distinguish sensors or emitters. In practice, you can use <code>SensorIndex</code> or <code>EmitterIndex</code> of the sensor or emitter objects, respectively.
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 [<code>azimuth</code>; <code>elevation</code>] 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 [ <code>azimuth</code> ; <code>elevation</code> ] in degrees.
ScanLimits	The minimum and maximum angles the sensor or emitter can scan from its <code>Orientation</code> . <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 [<code>minAz</code>, <code>maxAz</code>] 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 [<code>minAz</code>, <code>maxAz</code>; <code>minEl</code>, <code>maxEl</code>] 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.

## Examples

### Record a Tracking Scenario

Create a new scenario and add a platform.

```
scene = trackingScenario;
plat = platform(scene);
```

Specify the platform trajectory. The distance of the trajectory is 25 meters. The trajectory velocity is 20 m/s in the x-direction.

```
plat.Trajectory = waypointTrajectory('Waypoints',[0 0 0; 25 0 0], ...
    'TimeOfArrival', [0 25/20]);
```

Run simulation and record results.

```
r = record(scene)

r=13x1 struct array with fields:
    SimulationTime
    Poses
```

Show the record 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 record 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]
```

## Load and Record Tracking Scenario

Load an air traffic control tracking scenario.

```
load ATCScenario scenario
```

Run simulation and record results.

```
r = record(scenario, 'quaternion', 'IncludeEmitters', true, ...
    'IncludeSensors', true, 'InitialSeed', 2019)
```

```
r=3215×1 struct array with fields:
    SimulationTime
    Poses
    Emissions
    EmitterConfigurations
    EmitterPlatformIDs
    Detections
    PointClouds
    PointCloudClusters
    SensorConfigurations
    SensorPlatformIDs
    CoverageConfig
```

Show the record at the initial time.

```
r(1)
```

```
ans = struct with fields:
    SimulationTime: 0
        Poses: [4x1 struct]
        Emissions: {0x1 cell}
    EmitterConfigurations: [0x1 struct]
    EmitterPlatformIDs: [0x1 double]
        Detections: {}
        PointClouds: {0x1 cell}
        PointCloudClusters: {0x1 cell}
    SensorConfigurations: [1x1 struct]
    SensorPlatformIDs: 1
        CoverageConfig: [1x1 struct]
```

Show the record at the final time.

```
r(end)
```

```
ans = struct with fields:
    SimulationTime: 59.9947
        Poses: [4x1 struct]
        Emissions: {0x1 cell}
    EmitterConfigurations: [0x1 struct]
    EmitterPlatformIDs: [0x1 double]
        Detections: {}
        PointClouds: {0x1 cell}
        PointCloudClusters: {0x1 cell}
    SensorConfigurations: [1x1 struct]
    SensorPlatformIDs: 1
```

CoverageConfig: [1x1 struct]

**See Also**

[trackingScenario](#) | [trackingScenarioRecording](#)

**Introduced in R2018b**

# restart

Restart tracking scenario simulation

## Syntax

```
restart(sc)
```

## Description

`restart(sc)` restarts the simulation of the tracking scenario, `sc`, from the beginning and sets the `SimulationTime` property of `sc` to zero.

## Input Arguments

### **sc — Tracking scenario**

`trackingScenario` object

Tracking scenario, specified as a `trackingScenario` object.

## Introduced in R2018b

# trackingScenarioRecording

Tracking scenario recording

## Description

Use the `trackingScenarioRecording` object to record a tracking scenario.

## Creation

### Syntax

```
TSR = trackingScenarioRecording(recordedData)
TSR = trackingScenarioRecording(recordedData,Name,Value)
```

### Description

`TSR = trackingScenarioRecording(recordedData)` returns a `trackingScenarioRecording` object `TSR` using the recorded data. `recordedData` sets the value of the `RecordedData` property.

`TSR = trackingScenarioRecording(recordedData,Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes.

### Input Arguments

**recordedData — Recorded data**  
structure

Recorded data, specified as a structure. The fields of the structure are the same as the fields of the output of the `record` method of `trackingScenario`.

## Properties

**RecordedData — Recorded data stored in the 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` method of `trackingScenario`.

**CurrentTime — Timestamp of latest read data**  
0 | nonnegative scalar

Timestamp of the latest read data, specified as a nonnegative scalar. When you use the `read` method on the object, the method reads the recorded dataset that has `SimulationTime` larger than the `CurrentTime`.

**CurrentStep — Step index of the latest read data**

$\theta$  | nonnegative integer

Step index of the latest read data, specified as a nonnegative integer. When you use the `read` method on the object, the method reads the next-step dataset.

## Object Functions

`read` Read recorded data  
`isDone` End-of-data status

## Examples

### Run a Recorded Scenario

Load recorded data from a prerecorded scenario called `recordedScenario`. Construct a `trackingScenarioRecording` object using the recorded data.

```
load recordedScenario recordedData
recording = trackingScenarioRecording(recordedData);
```

Construct a theater plot to display the recorded data using multiple plotters.

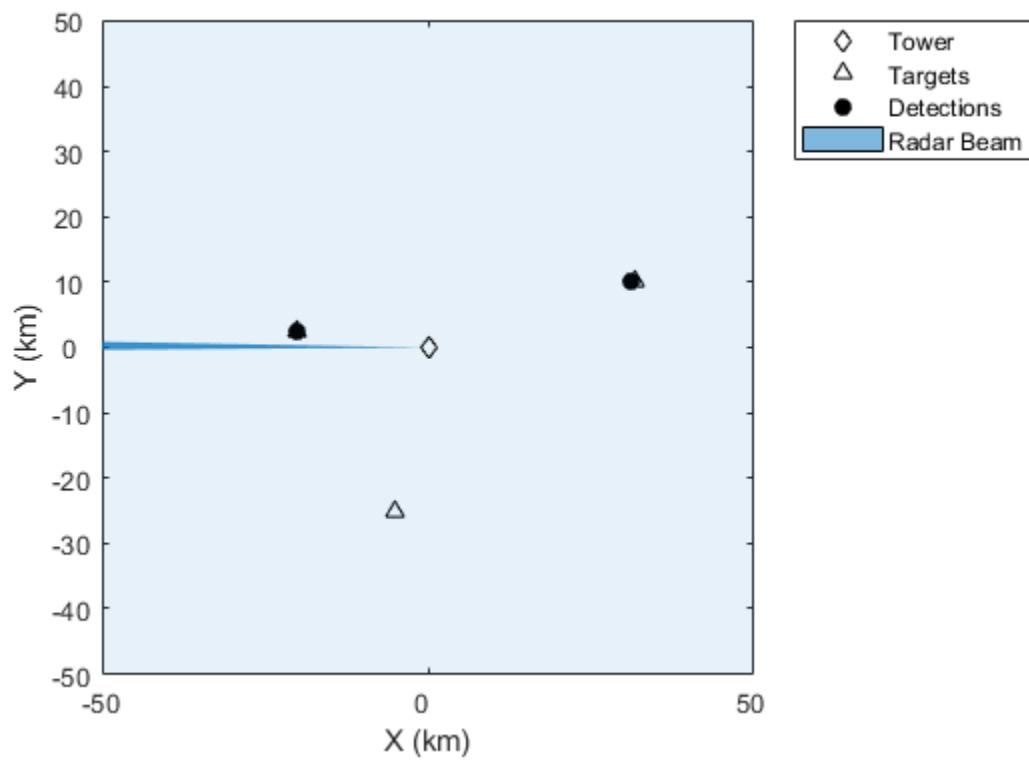
```
tp = theaterPlot('AxesUnits', ["km" "km" "km"], 'XLimits', [-50 50]*1e3, ...
    'YLimits', [-50 50]*1e3, 'ZLimits', [-20 20]*1e3);
to = platformPlotter(tp,'DisplayName','Tower','Marker','d');
pp = platformPlotter(tp,'DisplayName','Targets');
dp = detectionPlotter(tp,'DisplayName','Detections','MarkerFaceColor','black');
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,[1]));
    plotCoverage(cp,covcon);
    if ~isempty(dets)
        plotDetection(dp,cell2mat(cellfun(@(c) c.Measurement(:)', scanBuffer, 'UniformOutput', ...
            end

    % Clear the buffer when a 360 degree scan is complete
    if senconfig.IsScanDone
        scanBuffer = {};
        dp.clearData;
    end
end
```



**See Also**

[record](#) | [trackingScenario](#)

**Introduced in R2020a**

# read

Read recorded data

## Syntax

```
[simTime,poses,detections,sensorConfigs,sensorPlatformIDs,emissions,
emitterConfigs,emitterPlatformIDs] = read(TSR)
```

## Description

[simTime,poses,detections,sensorConfigs,sensorPlatformIDs,emissions,emitterConfigs,emitterPlatformIDs] = read(TSR) returns one recorded dataset at the simulation time, simTime, from a tracking scenario recording TSR.

## Examples

### Run a Recorded Scenario

Load recorded data from a prerecorded scenario called recordedScenario. Construct a trackingScenarioRecording object using the recorded data.

```
load recordedScenario recordedData
recording = trackingScenarioRecording(recordedData);
```

Construct a theater plot to display the recorded data using multiple plotters.

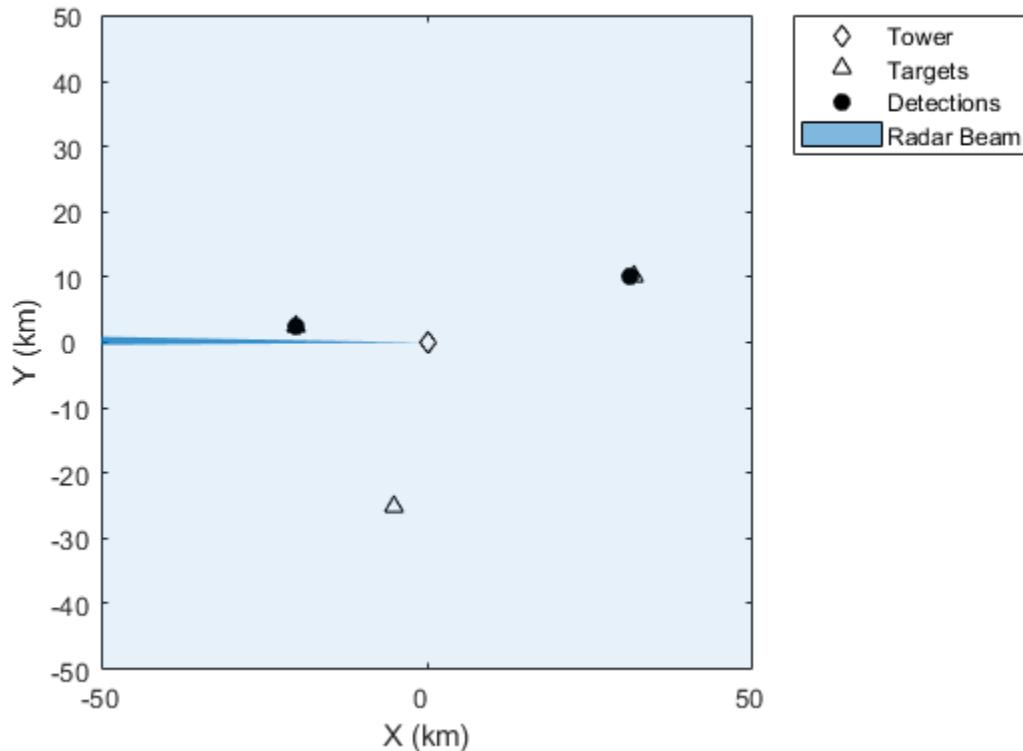
```
tp = theaterPlot('AxesUnits', ["km" "km" "km"], 'XLimits', [-50 50]*1e3, ...
    'YLimits', [-50 50]*1e3, 'ZLimits', [-20 20]*1e3);
to = platformPlotter(tp,'DisplayName','Tower','Marker','d');
pp = platformPlotter(tp,'DisplayName','Targets');
dp = detectionPlotter(tp,'DisplayName','Detections','MarkerFaceColor','black');
cp = coveragePlotter(tp,'DisplayName','Radar Beam');

coverage = struct('Index',1,'LookAngle',[0;-7],'FieldOfView',[1;10],...
    'ScanLimits',[0 365;-12 -21],'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<AGR0W>
    plotPlatform(to,poses(1).Position);
    plotPlatform(pp,reshape([poses(2:4).Position]',3,[1]));
    plotCoverage(cp,covcon);
    if ~isempty(dets)
        plotDetection(dp,cell2mat(cellfun(@(c) c.Measurement(:)', scanBuffer, 'UniformOutput', false)));
    end
end
```

```
% Clear the buffer when a 360 degree scan is complete
if senconfig.IsScanDone
    scanBuffer = {};
    dp.clearData;
end
end
```



## Input Arguments

**TSR — Tracking scenario recording**  
trackingScenarioRecording object

Tracking scenario recording, specified as a trackingScenarioRecording 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. The fields of each structure are:

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', the <code>Position</code> is the 3-element Cartesian position coordinates in meters.</li> <li>If the <code>coordinateSystem</code> argument is specified as 'Geodetic', the <code>Position</code> is the 3-element 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. The fields of each structure are:

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.
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.

#### **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 `radarEmission` or `sonarEmission` objects

Emissions, returned as a cell array of `radarEmission` or `sonarEmission` objects.

#### **emitterConfigs — Emitter configurations**

array of structures

Emitter configurations, returned as an array of structures. The fields of each structure are:

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.
-----------------------	---

**emitterPlatformIDs — Platform IDs of emitters**

array of nonnegative integers

Platform IDs of emitters, returned as an array of nonnegative integers.

**See Also**

[record](#) | [trackingScenario](#) | [trackingScenarioRecording](#)

**Introduced in R2020a**

# 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 tracking 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 = monostaticRadarSensor(1,'Rotator');
emitter = radarEmitter(2);
```

Obtain coverage configurations based on sensor's position information.

```
cfg = coverageConfig({radar, emitter})

cfg=2x1 struct array with fields:
    Index
    LookAngle
    FieldOfView
    ScanLimits
    Range
    Position
    Orientation

cfg2 = coverageConfig({radar, emitter},[1000 0 0 ; 0 1000 0])

cfg2=2x1 struct array with fields:
    Index
    LookAngle
    FieldOfView
    ScanLimits
    Range
```

Position  
Orientation

## Input Arguments

### **sc — Tracking scenario**

trackingScenario object

Tracking scenario, specified as a trackingScenario object.

### **sensors — Sensors or emitters**

sensor or emitter object |  $N$ -element cell array of sensor or emitter object

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 monostaticRadarSensor, radarSensor, radarEmitter, sonarSensor, sonarEmitter, irSensor, and monostaticLidarSensor.

### **positions — Position of sensor or emitter's platform**

$N$ -by-3 matrix of scalar

Position of sensor or emitter's platform, specified as an  $N$ -by-3 matrix of scalars. 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 quaternion

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 structure

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. Each configuration structure contains seven fields:

## Fields of configurations

Field	Description
Index	A unique integer to distinguish sensors or emitters. In practice, you can use <code>SensorIndex</code> or <code>EmitterIndex</code> of the sensor or emitter objects, respectively.
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 [<code>azimuth</code>; <code>elevation</code>] 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 [ <code>azimuth</code> ; <code>elevation</code> ] in degrees.
ScanLimits	The minimum and maximum angles the sensor or emitter can scan from its <code>Orientation</code> . <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 [<code>minAz</code>, <code>maxAz</code>] 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 [<code>minAz</code>, <code>maxAz</code>; <code>minEl</code>, <code>maxEl</code>] 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.

## See Also

`coveragePlotter` | `plotCoverage` | `trackingScenario`

**Introduced in R2020a**

# rcsSignature

Radar cross-section pattern

## Description

`rcsSignature` 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.

## Creation

### Syntax

```
rcssig = rcsSignature
rcssig = rcsSignature(Name,Value)
```

### Description

`rcssig = rcsSignature` creates an `rcsSignature` object with default property values.

`rcssig = rcsSignature(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 `rcsSignature` 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.

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.

Example: [-30:0.5:30]

Data Types: double

#### **Frequency — Pattern frequencies**

[ -90 90] (default) | length- $K$  real-valued vector

Frequencies used to define the applicable RCS for each page of the **Pattern** property. Specify the frequencies as a length- $K$  vector.  $K$  must be greater than two. Frequency units are in hertz.

Example: [-30:0.1:30]

Data Types: double

## **Object Functions**

perturb Apply perturbations to object

## **Examples**

### **Radar Cross-Section of Ellipsoid**

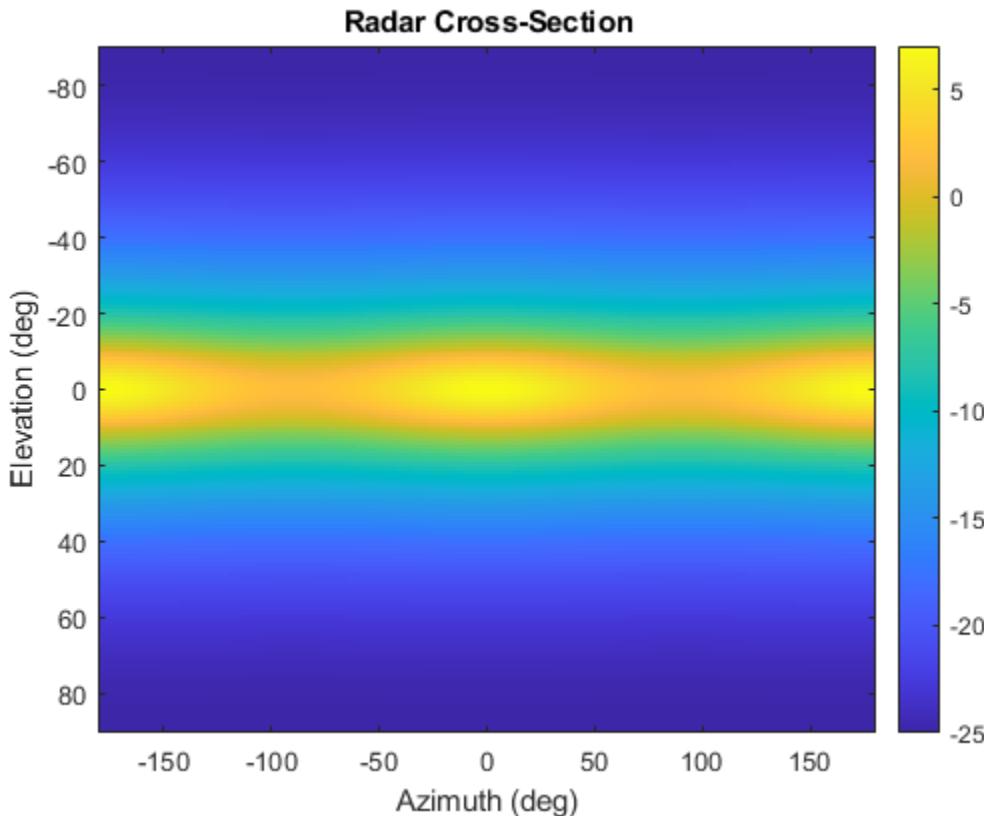
Specify the radar cross-section (RCS) of a triaxial ellipsoid and plot RCS values along an azimuth cut.

Specify the lengths of the axes of the ellipsoid. Units are in meters.

```
a = 0.15;
b = 0.20;
c = 0.95;
```

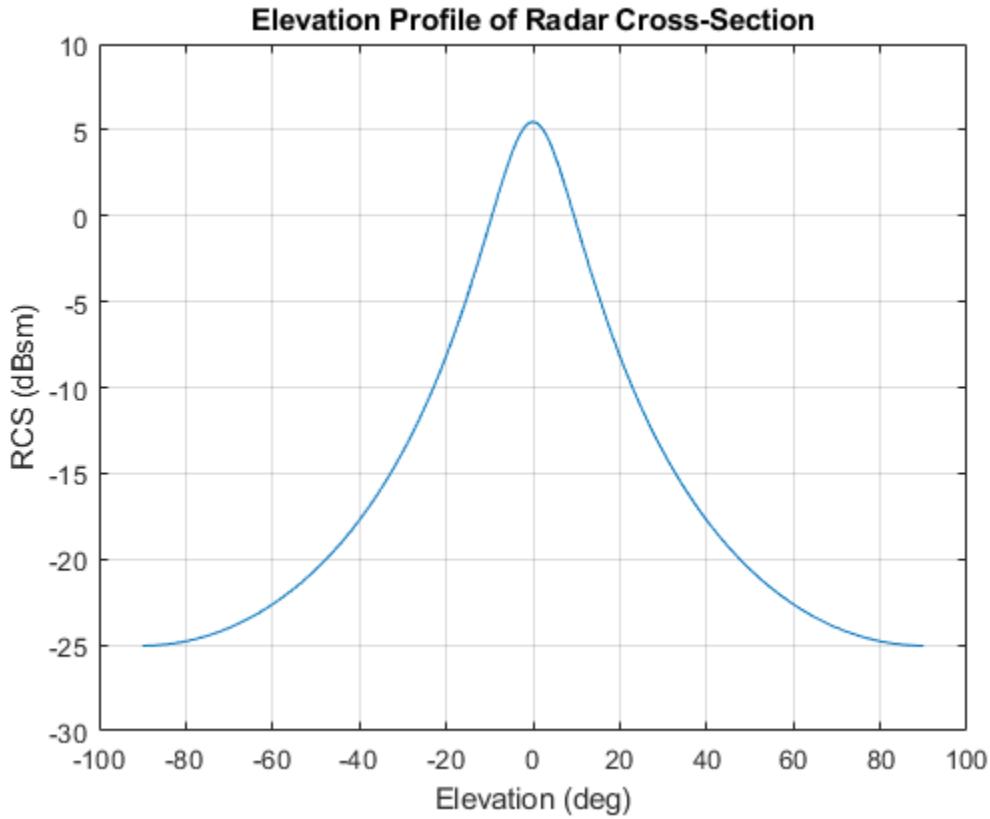
Create an RCS array. Specify the range of azimuth and elevation angles over which RCS is defined. Then, use an analytical model to compute the radar cross-section of the ellipsoid. Create an image of the RCS.

```
az = [-180:1:180];
el = [-90:1:90];
rcs = rcs_ellipsoid(a,b,c,az,el);
rcsdb = 10*log10(rcs);
imagesc(az,el,rcsdb)
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',rcsdb,'Azimuth',az,'Elevation',el,'Frequency',[300e6 300e6]);
rcsdb1 = value(rcssig,30,el,300e6);
plot(el,rcsdb1)
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
rcs = (pi*a^2*b^2*c^2)./denom;
end
```

## 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

### Classes

tsSignature

**Introduced in R2018b**

## value

Radar cross-section at specified angle and frequency

### Syntax

```
rcsval = 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.

### 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

### **rcsval — 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.

## Examples

### **Radar Cross-Section of Ellipsoid**

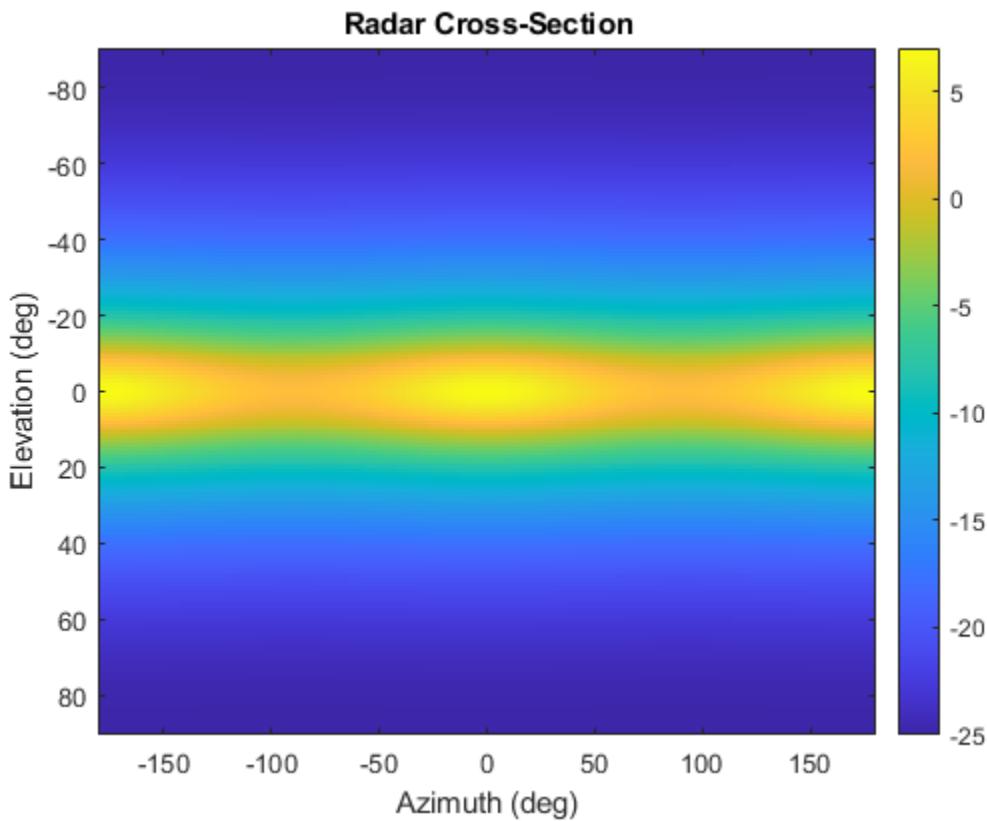
Specify the radar cross-section (RCS) of a triaxial ellipsoid and plot RCS values along an azimuth cut.

Specify the lengths of the axes of the ellipsoid. Units are in meters.

```
a = 0.15;  
b = 0.20;  
c = 0.95;
```

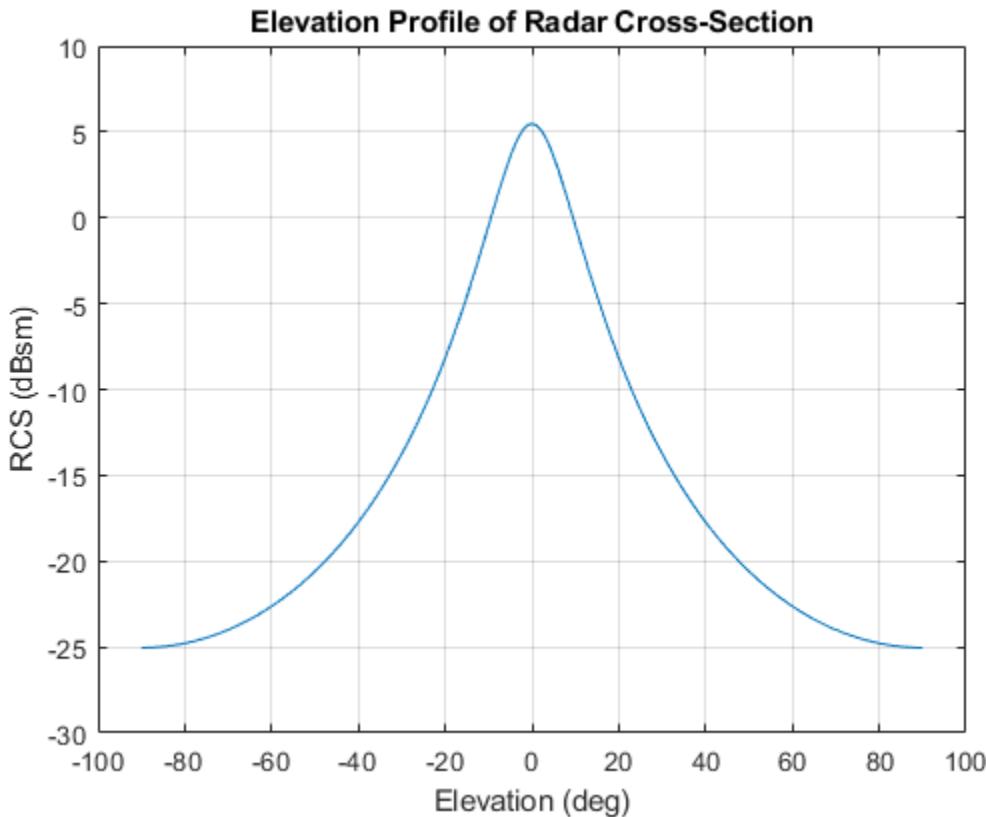
Create an RCS array. Specify the range of azimuth and elevation angles over which RCS is defined. Then, use an analytical model to compute the radar cross-section of the ellipsoid. Create an image of the RCS.

```
az = [-180:1:180];  
el = [-90:1:90];  
rcs = rcs_ellipsoid(a,b,c,az,el);  
rcsdb = 10*log10(rcs);  
imagesc(az,el,rcsdb)  
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', rcsdb, 'Azimuth', az, 'Elevation', el, 'Frequency', [300e6 300e6]);
rcsdb1 = value(rcssig, 30, el, 300e6);
plot(el, rcsdb1)
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(s));
rcs = (pi*a^2*b^2*c^2)./denom;
end
```

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

**Introduced in R2018b**

# tsSignature

Target strength pattern

## Description

`tsSignature` creates a sonar target strength (TS) signature object. You can use this object to model an angle-dependent and frequency-dependent target strength pattern. Target strength determines the intensity of reflected sound signal power from a target.

## Creation

### Syntax

```
tssig = tsSignature  
tssig = tsSignature(Name,Value)
```

### Description

`tssig = tsSignature` creates a `tsSignature` object with default property values.

`tssig = tsSignature(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 `tsSignature` when constructing the object. The property values are not changeable after construction.

---

## Properties

### Pattern — Target strength pattern

`[-30 -30; -30 -30]` (default) |  $Q$ -by- $P$  real-valued matrix |  $Q$ -by- $P$ -by- $K$  real-valued array

Sampled target strength 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 TS 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 TS samples in elevation.
- $P$  is the number of TS samples in azimuth.
- $K$  is the number of TS 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 a TS pattern for an elevation cut (constant azimuth), you can specify the TS 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 a TS pattern for an azimuth cut (constant elevation), you can specify the TS 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 a TS pattern for one frequency, you can specify the TS 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.

Example: [-45:0.1:45]

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.

Example: [-30:0.1:30]

Data Types: double

#### **Frequency — Pattern frequencies**

[0 1e8] (default) | length- $K$  real-valued vector

Frequencies used to define the applicable RCS for each page of the **Pattern** property. Specify the frequencies as a length- $K$  vector.  $K$  must be no less than two. Frequency units are in hertz.

Example: [-30:0.1:30]

Data Types: double

## **Object Functions**

value Target strength at specified angle and frequency

## **Examples**

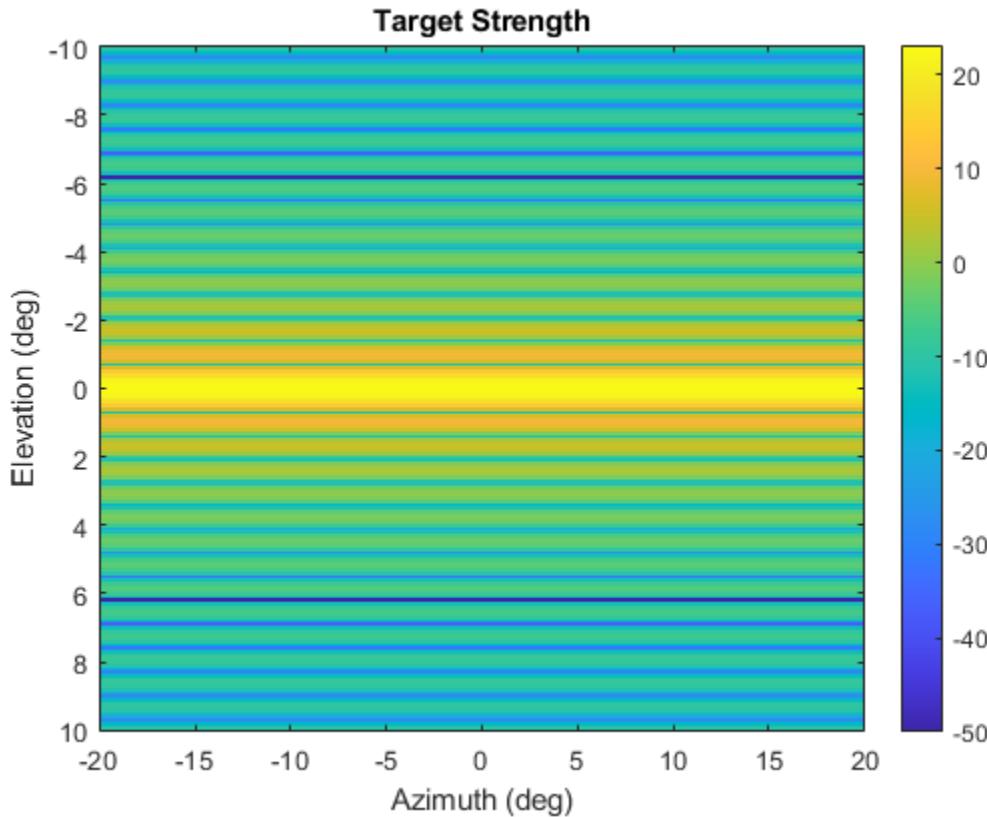
### **Target Strength of Rigid Cylinder**

Specify the target strength (TS) of a 5m long rigid cylinder immersed in water and plot TS values along an azimuth cut. Assume the short-wavelength approximation. The cylinder radius is 2m. The speed of sound is 1520 m/s.

```
L = 5;
a = 2;
```

Create an array of target strengths at two wavelengths. First, specify the range of azimuth and elevation angles over which TS is defined. Then, use an analytical model to compute the target strength. Create an image of the TS.

```
lambda = [0.12, .1];
c = 1520.0;
az = [-20:0.1:20];
el = [-10:0.1:10];
ts1 = ts_cylinder(L,a,az,el,lambda(1));
ts2 = ts_cylinder(L,a,az,el,lambda(2));
tsdb1 = 10*log10(ts1);
tsdb2 = 10*log10(ts2);
imagesc(az,el,tsdb1)
title('Target Strength')
xlabel('Azimuth (deg)')
ylabel('Elevation (deg)')
colorbar
```



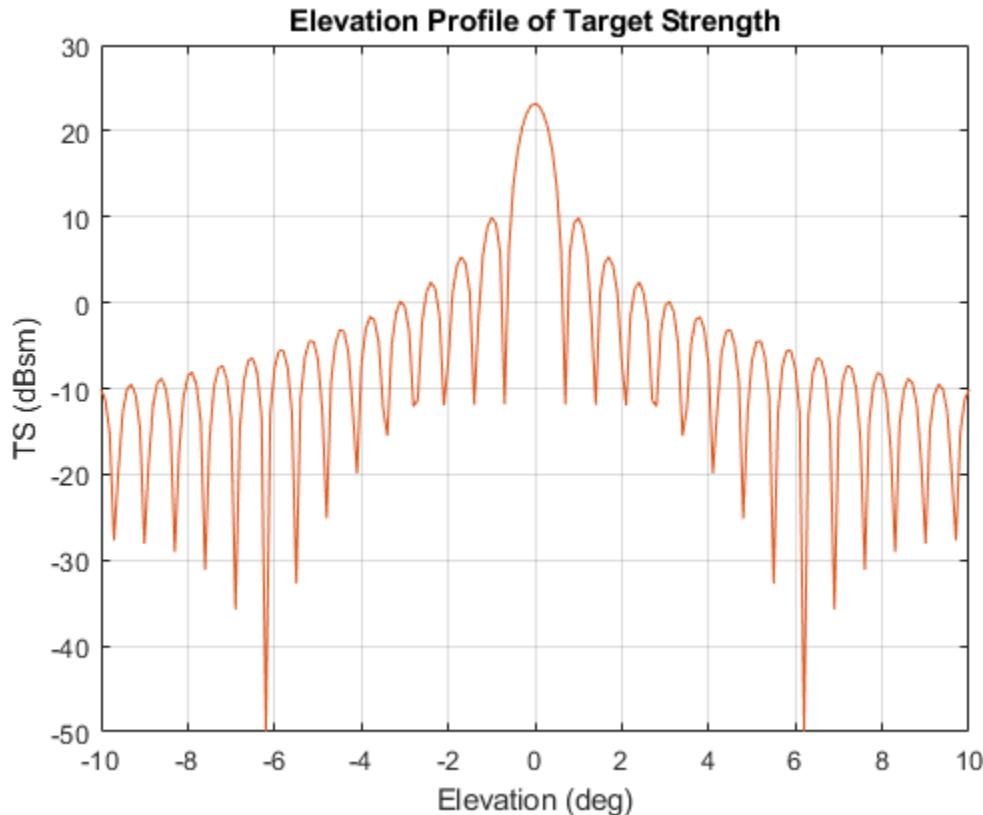
Create a `tsSignature` object and plot an elevation cut at 30° azimuth.

```
tsdb(:,:,1) = tsdb1;
tsdb(:,:,2) = tsdb2;
freq = c./lambda;
tssig = tsSignature('Pattern',tsdb,'Azimuth',az,'Elevation',el,'Frequency',freq);
```

```

ts = value(tssig,30,el,freq(1));
plot(el,tsdb1)
grid
title('Elevation Profile of Target Strength')
xlabel('Elevation (deg)')
ylabel('TS (dBsm)')

```



```

function ts = ts_cylinder(L,a,az,el,lambda)
k = 2*pi/lambda;
beta = k*L*sind(el')*ones(size(az));
gamma = cosd(el')*ones(size(az));
ts = a*L^2*(sinc(beta).^2).*gamma.^2/2/lambda;
ts = max(ts,10^(-5));
end

function s = sinc(theta)
s = ones(size(theta));
idx = (abs(theta) <= 1e-2);
s(idx) = 1 - 1/6*(theta(idx)).^2;
s(~idx) = sin(theta(~idx))./theta(~idx);
end

```

## References

- [1] Urich, Robert J. *Principles of Underwater Sound*, 3rd ed. New York: McGraw-Hill, Inc. 2005.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

**Classes**

`rcsSignature`

**Introduced in R2018b**

## value

Target strength at specified angle and frequency

### Syntax

```
tsval = value(tssig,az,el,freq)
```

### Description

`tsval = value(tssig,az,el,freq)` returns the value, `tsval`, of the target strength specified by the target strength signature object, `tssig`, computed at azimuth `az`, elevation `el`, and frequency `freq`. If the specified azimuth and elevation is outside of the region in which the target strength signature is defined, the target strength value, `tsval`, is returned as `-Inf` in dBsm.

### Input Arguments

#### **tssig — Target strength signature**

tsSignature object

Target strength signature, specified as a `tsSignature` 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 — TS frequency**

positive scalar | length- $M$  vector with positive, real elements

TS 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- $M$ . Units are in Hertz.

Example: `20e3`

Data Types: double

## Output Arguments

### tssval — Target strength

scalar | real-valued length- $M$  vector

Target strength, returned as a scalar or real-valued length- $M$  vector. Units are in dBsm.

## Object Functions

perturbations    Perturbation defined on object  
perturb           Apply perturbations to object

## Examples

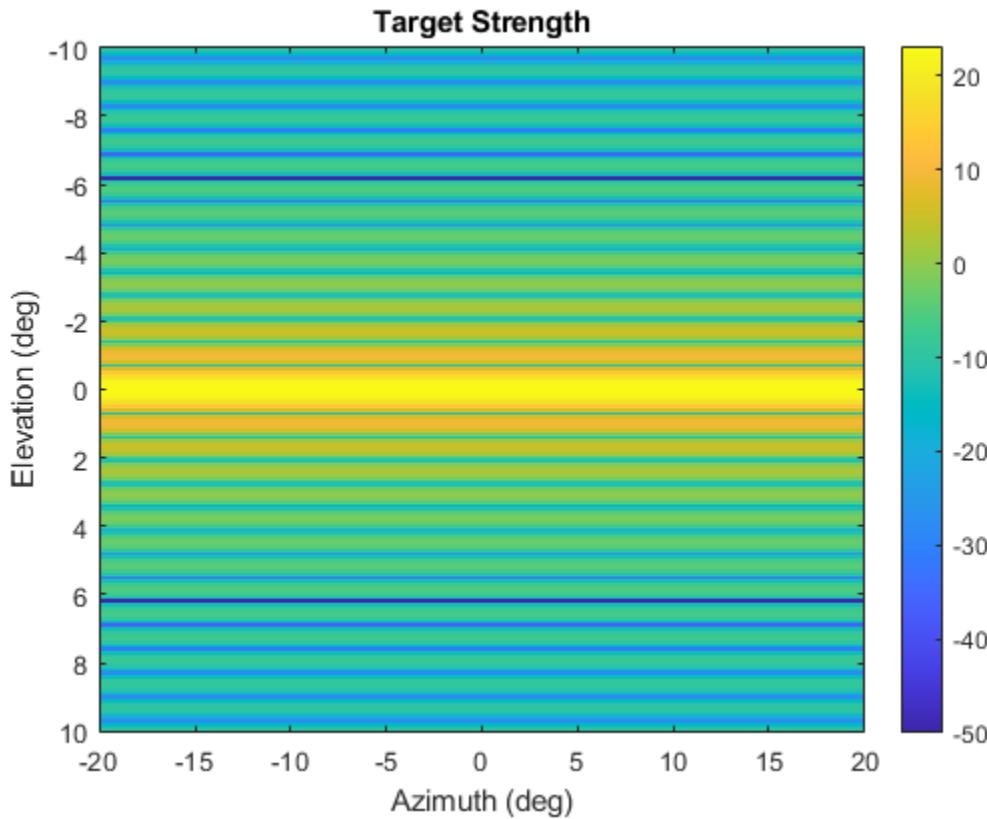
### Target Strength of Rigid Cylinder

Specify the target strength (TS) of a 5m long rigid cylinder immersed in water and plot TS values along an azimuth cut. Assume the short-wavelength approximation. The cylinder radius is 2m. The speed of sound is 1520 m/s.

```
L = 5;  
a = 2;
```

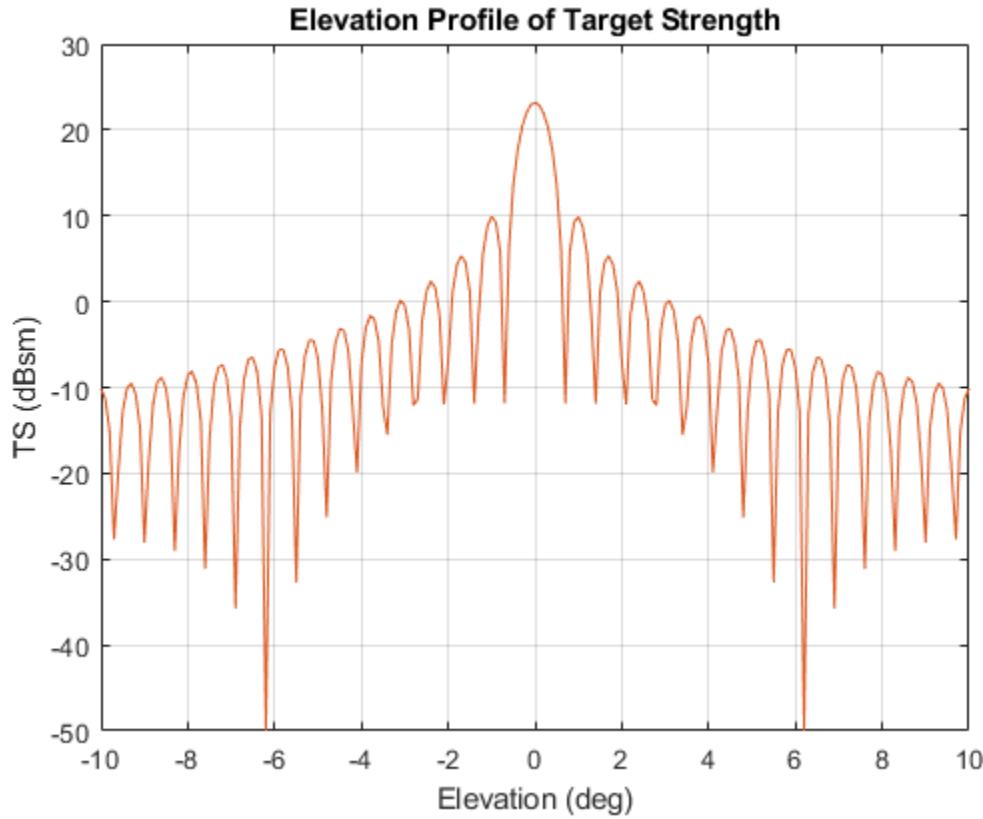
Create an array of target strengths at two wavelengths. First, specify the range of azimuth and elevation angles over which TS is defined. Then, use an analytical model to compute the target strength. Create an image of the TS.

```
lambda = [0.12, .1];  
c = 1520.0;  
az = [-20:0.1:20];  
el = [-10:0.1:10];  
ts1 = ts_cylinder(L,a,az,el,lambda(1));  
ts2 = ts_cylinder(L,a,az,el,lambda(2));  
tsdb1 = 10*log10(ts1);  
tsdb2 = 10*log10(ts2);  
imagesc(az,el,tsdb1)  
title('Target Strength')  
xlabel('Azimuth (deg)')  
ylabel('Elevation (deg)')  
colorbar
```



Create a `tsSignature` object and plot an elevation cut at 30° azimuth.

```
tsdb(:,:,1) = tsdb1;
tsdb(:,:,2) = tsdb2;
freq = c./lambda;
tssig = tsSignature('Pattern',tsdb,'Azimuth',az,'Elevation',el,'Frequency',freq);
ts = value(tssig,30,el,freq(1));
plot(el,tsdb1)
grid
title('Elevation Profile of Target Strength')
xlabel('Elevation (deg)')
ylabel('TS (dBsm)')
```



```

function ts = ts_cylinder(L,a,az,el,lambda)
k = 2*pi/lambda;
beta = k*L*sind(el')*ones(size(az));
gamma = cosd(el')*ones(size(az));
ts = a*L^2*(sinc(beta).^2).*gamma.^2/2/lambda;
ts = max(ts,10^(-5));
end

function s = sinc(theta)
s = ones(size(theta));
idx = (abs(theta) <= 1e-2);
s(idx) = 1 - 1/6*(theta(idx)).^2;
s(~idx) = sin(theta(~idx))./theta(~idx);
end

```

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

**Introduced in R2018b**

# irSignature

Infrared platform signature

## Description

The `irSignature` creates an infrared (IR) signature object. You can use this object to model an angle-dependent contrast radiant intensity of a platform. The radiant intensity is with respect to the background.

## Creation

### Syntax

```
irsig = irSignature
irsig = irSignature(Name,Value)
```

#### Description

`irsig = irSignature` creates an `irSignature` object with default property values.

`irsig = irSignature(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 `irSignature` when constructing the object. The property values are not changeable after construction.

---

## Properties

### Pattern — Sampled IR intensity pattern

[50 50; 50 50] (default) |  $Q$ -by- $P$  real-valued matrix

Sampled contrast IR intensity pattern, specified as a scalar, or a  $Q$ -by- $P$  real-valued matrix. The pattern is an array of IR values defined on a grid of elevation angles and azimuth angles. Azimuth and elevation are defined in the body frame of the target. Units are dBw/sr.

- $Q$  is the number of IR samples in elevation.
- $P$  is the number of IR samples in azimuth.

$Q$  and  $P$  usually match the length of the vectors defined in the `Elevation` and `Azimuth` properties, respectively, with these exceptions:

- If you want to model an IR pattern for an elevation cut (constant azimuth), you can specify the IR pattern as a  $Q$ -by-1 vector. Then, the elevation vector specified in the `Elevation` property must have length-2.

- If you want to model an IR pattern for an azimuth cut (constant elevation), you can specify the IR pattern as a 1-by- $P$  vector. Then, the azimuth vector specified in the **Azimuth** 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.

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.

Example: [-30:0.5:30]

Data Types: double

## **Object Functions**

**value** Infrared intensity at specified angle and frequency

## **Examples**

### **Create Direction-Dependent IR Signature**

Create and display an IR intensity signature. The signature depends on azimuth and elevation.

Define the azimuth and elevation angle sample points.

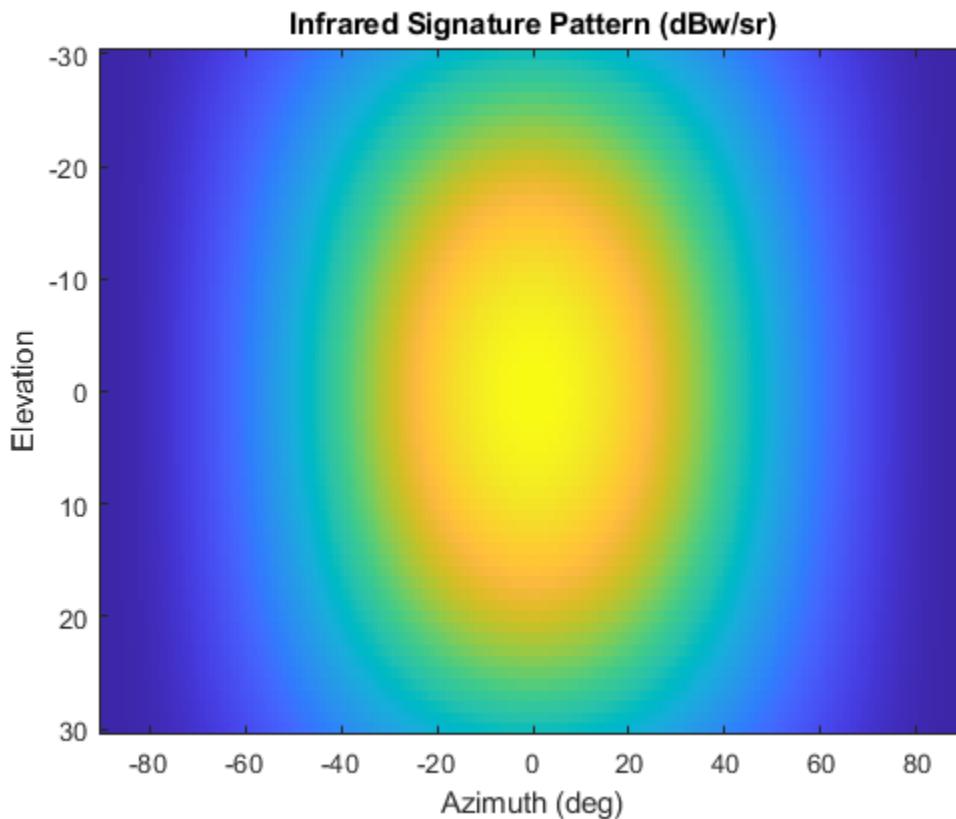
```
az = -90:90;
el = [-30:30];
```

Create the IR intensity signature pattern.

```
pat = 50*cosd(2*el.)*cosd(az).^2;
irsig = irSignature('Pattern', pat, 'Azimuth', az, 'Elevation', el);
```

Display the IR pattern.

```
imagesc(irsig.Azimuth, irsig.Elevation, irsig.Pattern)
xlabel('Azimuth (deg)')
ylabel('Elevation')
title('Infrared Signature Pattern (dBw/sr)')
```



Get the IR intensity value at 25 degrees azimuth and 10 degrees elevation.

```
value(irsig,25,10)
```

```
ans = 38.5929
```

Get IR intensity value outside of the valid elevation span.

```
value(irsig,25,35)
```

```
ans = -Inf
```

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Classes

[rcsSignature](#) | [tsSignature](#)

**Introduced in R2018b**

## value

Infrared intensity at specified angle and frequency

### Syntax

```
irval = value(irsig,az,el)
```

### Description

`irval = value(irsig,az,el)` returns the value of the IR intensity, `irval`, specified by the IR signature object, `irsig`, computed at the azimuth, `az`, and elevation, `el`. If the specified azimuth and elevation is outside of the region in which the IR signature is defined, the IR intensity is returned as -Inf in dBw/sr.

### Input Arguments

#### **irsig — IR signature object**

irSignature object

Radar cross-section signature, specified as an `irSignature` object.

#### **az — Azimuth angle**

scalar | real-valued length- $M$  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- $M$ .

Example: `30`

Data Types: `double`

#### **el — Elevation angle**

scalar | real-valued length- $M$  vector

Elevation angle, specified as scalar or real-valued length- $M$  vector. The `az` and `el` 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.

Example: `-4`

Data Types: `double`

### Output Arguments

#### **irval — Infrared intensity**

scalar | real-valued length- $M$  vector

Infrared intensity, returned as a scalar or real-valued length- $M$  vector. Units are in dBw/sr.

## Examples

### Create Direction-Dependent IR Signature

Create and display an IR intensity signature. The signature depends on azimuth and elevation.

Define the azimuth and elevation angle sample points.

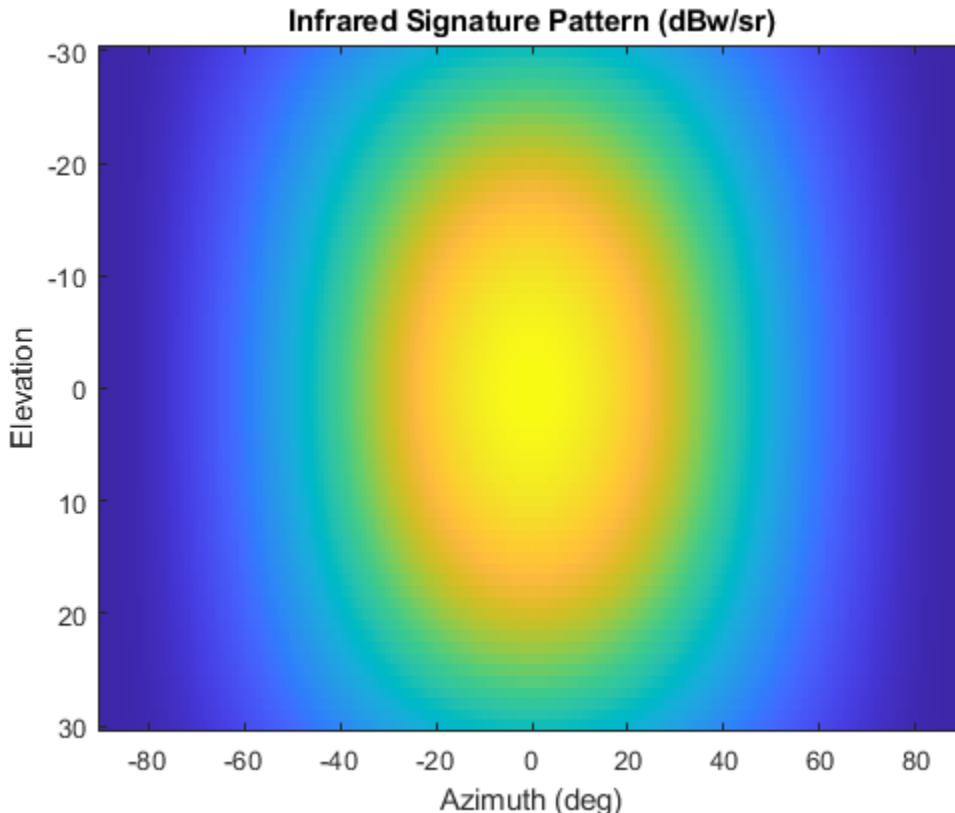
```
az = -90:90;
el = [-30:30];
```

Create the IR intensity signature pattern.

```
pat = 50*cosd(2*el.'){\}cosd(az).{^}2;
irsig = irSignature('Pattern',pat,'Azimuth',az,'Elevation',el);
```

Display the IR pattern.

```
imagesc(irsig.Azimuth,irsig.Elevation,irsig.Pattern)
xlabel('Azimuth (deg)')
ylabel('Elevation')
title('Infrared Signature Pattern (dBw/sr)')
```



Get the IR intensity value at 25 degrees azimuth and 10 degrees elevation.

```
value(irsig,25,10)
```

```
ans = 38.5929  
Get IR intensity value outside of the valid elevation span.  
value(irsig,25,35)  
ans = -Inf
```

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

**Introduced in R2018b**

# tunerconfig

Fusion filter tuner configuration options

## Description

The tunerconfig object creates a tuner configuration for a fusion filter (`insfilterAsync`, `ahrsfilter`, or `imufilter`) used to tune the filter for reduced estimation error.

## Creation

### Syntax

```
config = tunnerconfig(filterName)
config = tunnerconfig(fitlerName,Name,Value)
```

### Description

`config = tunnerconfig(filterName)` creates a `tunerconfig` object controlling the optimization algorithm of the `tune` function of the fusion filter. The `filterName` input can be '`imufilter`', '`ahrsfilter`', or '`insfilterAsync`' corresponding to the three types of fusion filters, respectively.

`config = tunnerconfig(fitlerName,Name,Value)` configures the created `tunerconfig` 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 quotes. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Any unspecified properties take default values.

For example, `tunnerconfig('imufilter','MaxIterations',3)` create a `tunerconfig` object for the `imufilter` object with the maximum number of allowed iterations equal to 3.

### Inputs Arguments

#### **filterName — Fusion filter name**

`'imufilter' | 'ahrsfilter' | 'insfilterAsync'`

Fusion filter names, specified as '`imufilter`', '`ahrsfilter`', or '`insfilterAsync`'.

## Properties

#### **TunableParameters — Tunable parameters**

array of string

Tunable parameters, specified as an array of strings. Each string is a tunable property name of the fusion filter. By default, the property contains all the tunable parameters of a fusion filter.

Example: `["AccelerometerNoise" "GyroscopeNoise"]`

**StepForward — Factor of forward step**

1.1 (default) | scalar larger than 1

Factor of a forward step, specified as a scalar larger than 1. During the tuning process, the tuner increases or decreases the noise parameters to achieve smaller estimation errors. This property specifies the ratio of parameter increase during a parameter increase step.

**StepBackward — Factor of backward step**

0.5 (default) | scalar in range (0,1)

Factor of a backward step, specified as a scalar in the range of (0,1). During the tuning process, the tuner increases or decreases the noise parameters to achieve smaller estimation errors. This property specifies the factor of parameter decrease during a parameter decrease step.

**MaxIterations — Maximum number of iterations**

20 (default) | positive integer

Maximum number of iterations allowed by the tuning algorithm, specified as a positive integer.

**ObjectiveLimit — Cost at which to stop the tuning process**

0.1 (default) | positive scalar

Cost at which to stop the tuning process, specified as a positive scalar. The tuning process stops if the **ObjectiveLimit** is achieved before the program reaches the maximum number of iterations.

**Display — Enable showing the iteration details**

"iter" (default) | "none"

Enable showing the iteration details, specified as "iter" or "none". When specified as:

- "iter" — The program shows the tuned parameter details in each iteration in the Command Window.
- "none" — The program does not show any tuning information.

**Cost — Metric for evaluating filter performance**

"RMS" (default) | "Custom"

Metric for evaluating filter performance, specified as "RMS" or "Custom". When specified as:

- "RMS" — The program optimizes the root-mean-squared (RMS) error between the estimate and the truth.
- "Custom" — The program optimizes the filter performance by using a customized cost function specified by the **CustomCostFcn** property.

**CustomCostFcn — Customized cost function**

function handle

Customized cost function, specified as a function handle.

## Examples

## Create Tunerconfig Object and Show Tunable Parameters

Create a tunerconfig object for the `insfilterAsync` object.

```
config = tunerconfig('insfilterAsync')

config =
    tunerconfig with properties:

        TunableParameters: [1x14 string]
            StepForward: 1.1000
            StepBackward: 0.5000
            MaxIterations: 20
            OptimalityTolerance: 0.1000
            Display: iter
            Cost: RMS
```

Display the default tunable parameters.

```
config.TunableParameters

ans = 1x14 string
    "AccelerometerNoise"      "GyroscopeNoise"      "MagnetometerNoise"      "GPSPositionNoise"      "GPSVelocity"      "Orientation"      "Position"      "Velocity"      "Bias"      "BiasRate"      "BiasRateRate"      "Cost"      "StepForward"      "StepBackward"
```

## Tune `insfilterAsync` to Optimize Pose Estimate

Load the recorded sensor data and ground truth data.

```
load('insfilterAsyncTuneData.mat');
```

Create timetables for the sensor data and the truth data.

```
sensorData = timetable(Accelerometer, Gyroscope, ...
    Magnetometer, GPSPosition, GPSVelocity, 'SampleRate', 100);
groundTruth = timetable(Orientation, Position, ...
    'SampleRate', 100);
```

Create an `insfilterAsync` filter object that has a few noise properties.

```
filter = insfilterAsync('State', initialState, ...
    'StateCovariance', initialStateCovariance, ...
    'AccelerometerBiasNoise', 1e-7, ...
    'GyroscopeBiasNoise', 1e-7, ...
    'MagnetometerBiasNoise', 1e-7, ...
    'GeomagneticVectorNoise', 1e-7);
```

Create a tuner configuration object for the filter. Set the maximum iterations to two. Also, set the tunable parameters as the unspecified properties.

```
config = tunerconfig('insfilterAsync', 'MaxIterations', 8);
config.TunableParameters = setdiff(config.TunableParameters, ...
    {'GeomagneticVectorNoise', 'AccelerometerBiasNoise', ...
    'GyroscopeBiasNoise', 'MagnetometerBiasNoise'});
config.TunableParameters
```

```
ans = 1×10 string
    "AccelerationNoise"      "AccelerometerNoise"      "AngularVelocityNoise"      "GPSPositionNoise"
```

Use the tuner noise function to obtain a set of initial sensor noises used in the filter.

```
measNoise = tunernoise('insfilterAsync')

measNoise = struct with fields:
    AccelerometerNoise: 1
        GyroscopeNoise: 1
    MagnetometerNoise: 1
        GPSPositionNoise: 1
        GPSVelocityNoise: 1
```

Tune the filter and obtain the tuned parameters.

```
tunedParams = tune(filter,measNoise,sensorData,groundTruth,config);
```

Iteration	Parameter	Metric
1	AccelerationNoise	2.1345
1	AccelerometerNoise	2.1264
1	AngularVelocityNoise	1.9659
1	GPSPositionNoise	1.9341
1	GPSVelocityNoise	1.8420
1	GyroscopeNoise	1.7589
1	MagnetometerNoise	1.7362
1	PositionNoise	1.7362
1	QuaternionNoise	1.7218
1	VelocityNoise	1.7218
2	AccelerationNoise	1.7190
2	AccelerometerNoise	1.7170
2	AngularVelocityNoise	1.6045
2	GPSPositionNoise	1.5948
2	GPSVelocityNoise	1.5323
2	GyroscopeNoise	1.4803
2	MagnetometerNoise	1.4703
2	PositionNoise	1.4703
2	QuaternionNoise	1.4632
2	VelocityNoise	1.4632
3	AccelerationNoise	1.4596
3	AccelerometerNoise	1.4548
3	AngularVelocityNoise	1.3923
3	GPSPositionNoise	1.3810
3	GPSVelocityNoise	1.3322
3	GyroscopeNoise	1.2998
3	MagnetometerNoise	1.2976
3	PositionNoise	1.2976
3	QuaternionNoise	1.2943
3	VelocityNoise	1.2943
4	AccelerationNoise	1.2906
4	AccelerometerNoise	1.2836
4	AngularVelocityNoise	1.2491
4	GPSPositionNoise	1.2258
4	GPSVelocityNoise	1.1880
4	GyroscopeNoise	1.1701
4	MagnetometerNoise	1.1698

4	PositionNoise	1.1698
4	QuaternionNoise	1.1688
4	VelocityNoise	1.1688
5	AccelerationNoise	1.1650
5	AccelerometerNoise	1.1569
5	AngularVelocityNoise	1.1454
5	GPSPositionNoise	1.1100
5	GPSVelocityNoise	1.0778
5	GyroscopeNoise	1.0709
5	MagnetometerNoise	1.0675
5	PositionNoise	1.0675
5	QuaternionNoise	1.0669
5	VelocityNoise	1.0669
6	AccelerationNoise	1.0634
6	AccelerometerNoise	1.0549
6	AngularVelocityNoise	1.0549
6	GPSPositionNoise	1.0180
6	GPSVelocityNoise	0.9866
6	GyroscopeNoise	0.9810
6	MagnetometerNoise	0.9775
6	PositionNoise	0.9775
6	QuaternionNoise	0.9768
6	VelocityNoise	0.9768
7	AccelerationNoise	0.9735
7	AccelerometerNoise	0.9652
7	AngularVelocityNoise	0.9652
7	GPSPositionNoise	0.9283
7	GPSVelocityNoise	0.8997
7	GyroscopeNoise	0.8947
7	MagnetometerNoise	0.8920
7	PositionNoise	0.8920
7	QuaternionNoise	0.8912
7	VelocityNoise	0.8912
8	AccelerationNoise	0.8885
8	AccelerometerNoise	0.8811
8	AngularVelocityNoise	0.8807
8	GPSPositionNoise	0.8479
8	GPSVelocityNoise	0.8238
8	GyroscopeNoise	0.8165
8	MagnetometerNoise	0.8165
8	PositionNoise	0.8165
8	QuaternionNoise	0.8159
8	VelocityNoise	0.8159

Fuse the sensor data using the tuned filter.

```
dt = seconds(diff(groundTruth.Time));
N = size(sensorData,1);
qEst = quaternion.zeros(N,1);
posEst = zeros(N,3);
% Iterate the filter for prediction and correction using sensor data.
for ii=1:N
    if ii ~= 1
        predict(filter, dt(ii-1));
    end
    if all(~isnan(Accelerometer(ii,:)))
        fuseaccel(filter, Accelerometer(ii,:), ...
            tunedParams.AccelerometerNoise);
    end
end
```

```
end
if all(~isnan(Gyroscope(ii,:)))
    fusegyro(filter, Gyroscope(ii,:), ...
        tunedParams.GyroscopeNoise);
end
if all(~isnan(Magnetometer(ii,1)))
    fusemag(filter, Magnetometer(ii,:), ...
        tunedParams.MagnetometerNoise);
end
if all(~isnan(GPSPosition(ii,1)))
    fusegps(filter, GPSPosition(ii,:), ...
        tunedParams.GPSPositionNoise, GPSVelocity(ii,:), ...
        tunedParams.GPSVelocityNoise);
end
[posEst(ii,:), qEst(ii,:)] = pose(filter);
end
```

Compute the RMS errors.

```
orientationError = rad2deg(dist(qEst, Orientation));
rmsorientationError = sqrt(mean(orientationError.^2))

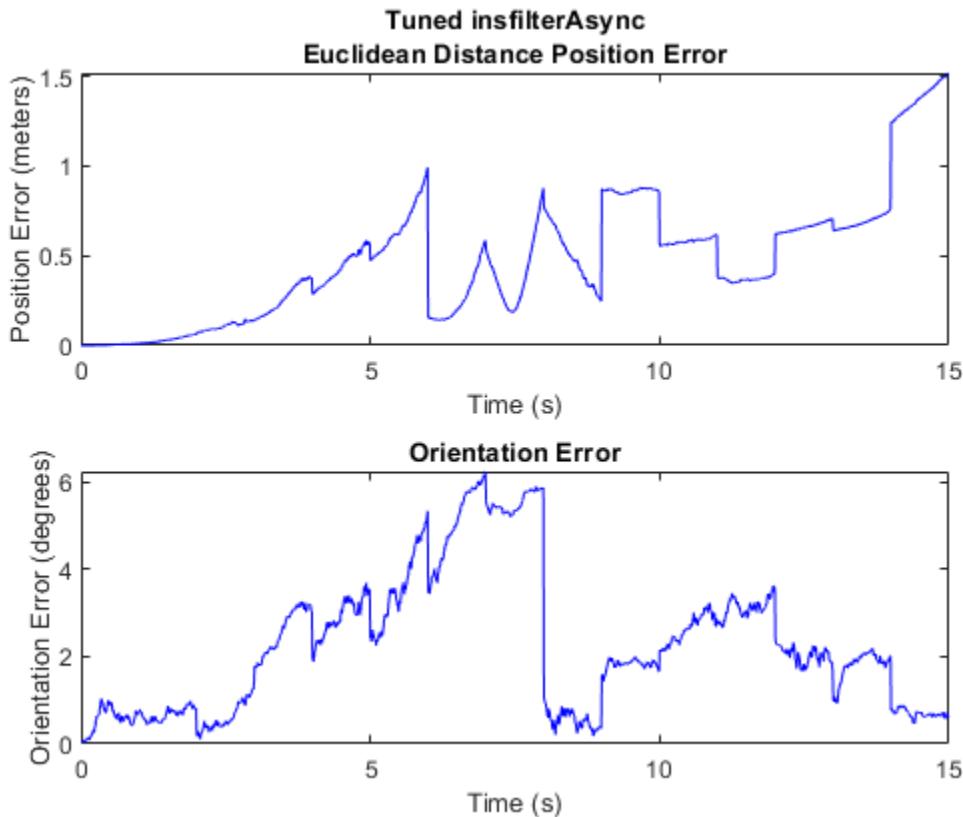
rmsorientationError = 2.7801

positionError = sqrt(sum((posEst - Position).^2, 2));
rmspositionError = sqrt(mean(positionError.^2))

rmspositionError = 0.5966
```

Visualize the results.

```
figure();
t = (0:N-1)./ groundTruth.Properties.SampleRate;
subplot(2,1,1)
plot(t, positionError, 'b');
title("Tuned insfilterAsync" + newline + "Euclidean Distance Position Error")
xlabel('Time (s)');
ylabel('Position Error (meters)')
subplot(2,1,2)
plot(t, orientationError, 'b');
title("Orientation Error")
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



### Tune imufilter to Optimize Orientation Estimate

Load recorded sensor data and ground truth data.

```
ld = load('imufilterTuneData.mat');
qTrue = ld.groundTruth.Orientation; % true orientation
```

Create an `imufilter` object and fuse the filter with the sensor data.

```
fuse = imufilter;
qEstUntuned = fuse(ld.sensorData.Accelerometer, ...
    ld.sensorData.Gyroscope);
```

Create a `tunerconfig` object and tune the imufilter to improve the orientation estimate.

```
cfg = tunerconfig('imufilter');
tune(fuse, ld.sensorData, ld.groundTruth, cfg);
```

Iteration	Parameter	Metric
1	AccelerometerNoise	0.1149
1	GyroscopeNoise	0.1146
1	GyroscopeDriftNoise	0.1146
1	LinearAccelerationNoise	0.1122
1	LinearAccelerationDecayFactor	0.1103

2	AccelerometerNoise	0.1102
2	GyroscopeNoise	0.1098
2	GyroscopeDriftNoise	0.1098
2	LinearAccelerationNoise	0.1070
2	LinearAccelerationDecayFactor	0.1053
3	AccelerometerNoise	0.1053
3	GyroscopeNoise	0.1048
3	GyroscopeDriftNoise	0.1048
3	LinearAccelerationNoise	0.1016
3	LinearAccelerationDecayFactor	0.1002
4	AccelerometerNoise	0.1001
4	GyroscopeNoise	0.0996
4	GyroscopeDriftNoise	0.0996
4	LinearAccelerationNoise	0.0962
4	LinearAccelerationDecayFactor	0.0950
5	AccelerometerNoise	0.0950
5	GyroscopeNoise	0.0943
5	GyroscopeDriftNoise	0.0943
5	LinearAccelerationNoise	0.0910
5	LinearAccelerationDecayFactor	0.0901
6	AccelerometerNoise	0.0900
6	GyroscopeNoise	0.0893
6	GyroscopeDriftNoise	0.0893
6	LinearAccelerationNoise	0.0862
6	LinearAccelerationDecayFactor	0.0855
7	AccelerometerNoise	0.0855
7	GyroscopeNoise	0.0848
7	GyroscopeDriftNoise	0.0848
7	LinearAccelerationNoise	0.0822
7	LinearAccelerationDecayFactor	0.0818
8	AccelerometerNoise	0.0817
8	GyroscopeNoise	0.0811
8	GyroscopeDriftNoise	0.0811
8	LinearAccelerationNoise	0.0791
8	LinearAccelerationDecayFactor	0.0789
9	AccelerometerNoise	0.0788
9	GyroscopeNoise	0.0782
9	GyroscopeDriftNoise	0.0782
9	LinearAccelerationNoise	0.0769
9	LinearAccelerationDecayFactor	0.0768
10	AccelerometerNoise	0.0768
10	GyroscopeNoise	0.0762
10	GyroscopeDriftNoise	0.0762
10	LinearAccelerationNoise	0.0754
10	LinearAccelerationDecayFactor	0.0753
11	AccelerometerNoise	0.0753
11	GyroscopeNoise	0.0747
11	GyroscopeDriftNoise	0.0747
11	LinearAccelerationNoise	0.0741
11	LinearAccelerationDecayFactor	0.0740
12	AccelerometerNoise	0.0740
12	GyroscopeNoise	0.0734
12	GyroscopeDriftNoise	0.0734
12	LinearAccelerationNoise	0.0728
12	LinearAccelerationDecayFactor	0.0728
13	AccelerometerNoise	0.0728
13	GyroscopeNoise	0.0721
13	GyroscopeDriftNoise	0.0721

```

13      LinearAccelerationNoise    0.0715
13      LinearAccelerationDecayFactor 0.0715
14      AccelerometerNoise        0.0715
14      GyroscopeNoise           0.0706
14      GyroscopeDriftNoise       0.0706
14      LinearAccelerationNoise    0.0700
14      LinearAccelerationDecayFactor 0.0700
15      AccelerometerNoise        0.0700
15      GyroscopeNoise           0.0690
15      GyroscopeDriftNoise       0.0690
15      LinearAccelerationNoise    0.0684
15      LinearAccelerationDecayFactor 0.0684
16      AccelerometerNoise        0.0684
16      GyroscopeNoise           0.0672
16      GyroscopeDriftNoise       0.0672
16      LinearAccelerationNoise    0.0668
16      LinearAccelerationDecayFactor 0.0667
17      AccelerometerNoise        0.0667
17      GyroscopeNoise           0.0655
17      GyroscopeDriftNoise       0.0655
17      LinearAccelerationNoise    0.0654
17      LinearAccelerationDecayFactor 0.0654
18      AccelerometerNoise        0.0654
18      GyroscopeNoise           0.0641
18      GyroscopeDriftNoise       0.0641
18      LinearAccelerationNoise    0.0640
18      LinearAccelerationDecayFactor 0.0639
19      AccelerometerNoise        0.0639
19      GyroscopeNoise           0.0627
19      GyroscopeDriftNoise       0.0627
19      LinearAccelerationNoise    0.0627
19      LinearAccelerationDecayFactor 0.0624
20      AccelerometerNoise        0.0624
20      GyroscopeNoise           0.0614
20      GyroscopeDriftNoise       0.0614
20      LinearAccelerationNoise    0.0613
20      LinearAccelerationDecayFactor 0.0613

```

Fuse the sensor data again using the tuned filter.

```
qEstTuned = fuse(ld.sensorData.Accelerometer, ...
    ld.sensorData.Gyroscope);
```

Compare the tuned and untuned filter RMS error performances.

```
dUntuned = rad2deg(dist(qEstUntuned, qTrue));
dTuned = rad2deg(dist(qEstTuned, qTrue));
rmsUntuned = sqrt(mean(dUntuned.^2))

rmsUntuned = 6.5864

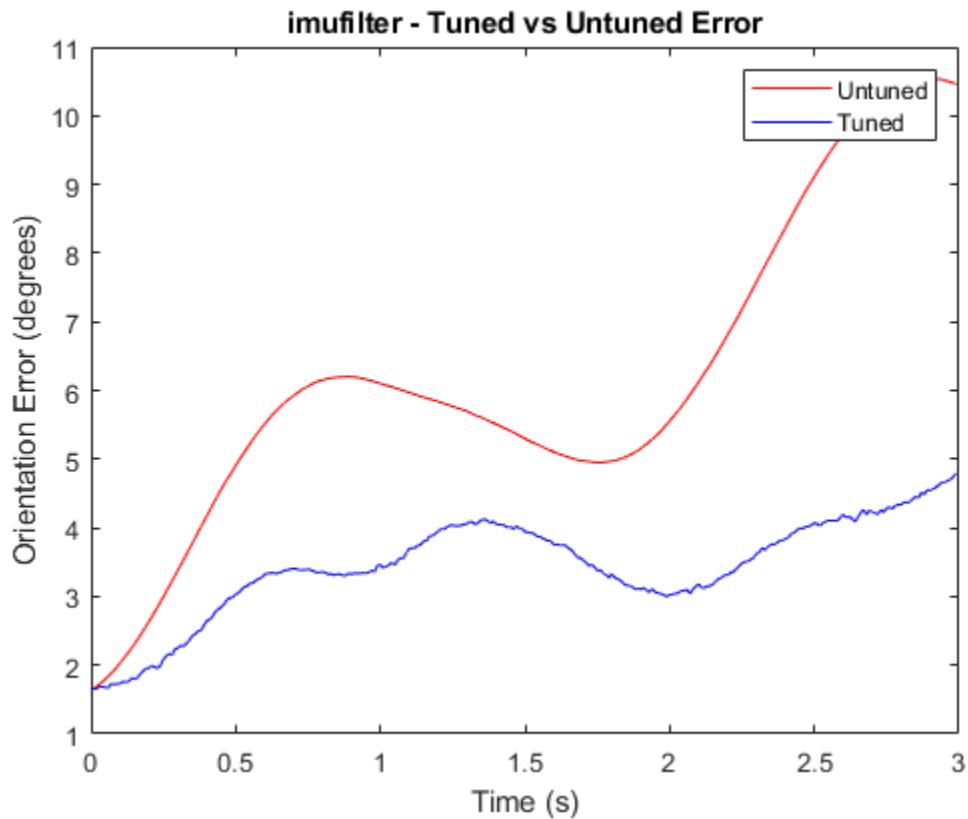
rmsTuned = sqrt(mean(dTuned.^2))

rmsTuned = 3.5098
```

Visualize the results.

```
N = numel(dUntuned);
t = (0:N-1)./. fuse.SampleRate;
```

```
plot(t, dUntuned, 'r', t, dTuned, 'b');
legend('Untuned', 'Tuned');
title('imufilter - Tuned vs Untuned Error')
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



## See Also

Introduced in R2020b

# extendedObjectMesh

Mesh representation of extended object

## Description

The `extendedObjectMesh` represents the 3-D geometry of an object. The 3-D geometry is represented by faces and vertices. Use these object meshes to specify the geometry of a `Platform` for simulating lidar sensor data using `monostaticLidarSensor`.

## Creation

### Syntax

```
mesh = extendedObjectMesh('cuboid')
mesh = extendedObjectMesh('cylinder')
mesh = extendedObjectMesh('cylinder',n)
mesh = extendedObjectMesh('sphere')
mesh = extendedObjectMesh('sphere',n)
mesh = extendedObjectMesh(vertices,faces)
```

### Description

`mesh = extendedObjectMesh('cuboid')` returns an `extendedObjectMesh` object, that defines a cuboid with unit dimensions. The origin of the cuboid is located at its geometric center.

`mesh = extendedObjectMesh('cylinder')` returns a hollow cylinder mesh with unit dimensions. The cylinder mesh has 20 equally spaced vertices around its circumference. The origin of the cylinder is located at its geometric center. The height is aligned with the z-axis.

`mesh = extendedObjectMesh('cylinder',n)` returns a cylinder mesh with n equally spaced vertices around its circumference.

`mesh = extendedObjectMesh('sphere')` returns a sphere mesh with unit dimensions. The sphere mesh has 119 vertices and 180 faces. The origin of the sphere is located at its center.

`mesh = extendedObjectMesh('sphere',n)` additionally allows you to specify the resolution, n, of the spherical mesh. The sphere mesh has  $(n + 1)^2 - 2$  vertices and  $2n(n - 1)$  faces.

`mesh = extendedObjectMesh(vertices,faces)` returns a mesh from faces and vertices. `vertices` and `faces` set the `Vertices` and `Faces` properties respectively.

## Properties

### Vertices — Vertices of defined object

N-by-3 matrix of real scalar

Vertices of the defined object, specified as an  $N$ -by-3 matrix of real scalars.  $N$  is the number of vertices. The first, second, and third element of each row represents the x-, y-, and z-position of each vertex, respectively.

### Faces — Faces of defined object

$M$ -by-3 matrix of positive integer

Faces of the defined object, specified as a  $M$ -by-3 array of positive integers.  $M$  is the number of faces. The three elements in each row are the vertex IDs of the three vertices forming the triangle face. The ID of the vertex is its corresponding row number specified in the `Vertices` property.

## Object Functions

Use the object functions to develop new meshes.

translate	Translate mesh along coordinate axes
rotate	Rotate mesh about coordinate axes
scale	Scale mesh in each dimension
applyTransform	Apply forward transformation to mesh vertices
join	Join two object meshes
scaleToFit	Auto-scale object mesh to match specified cuboid dimensions
show	Display the mesh as a patch on the current axes

## Examples

### Create and Translate Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

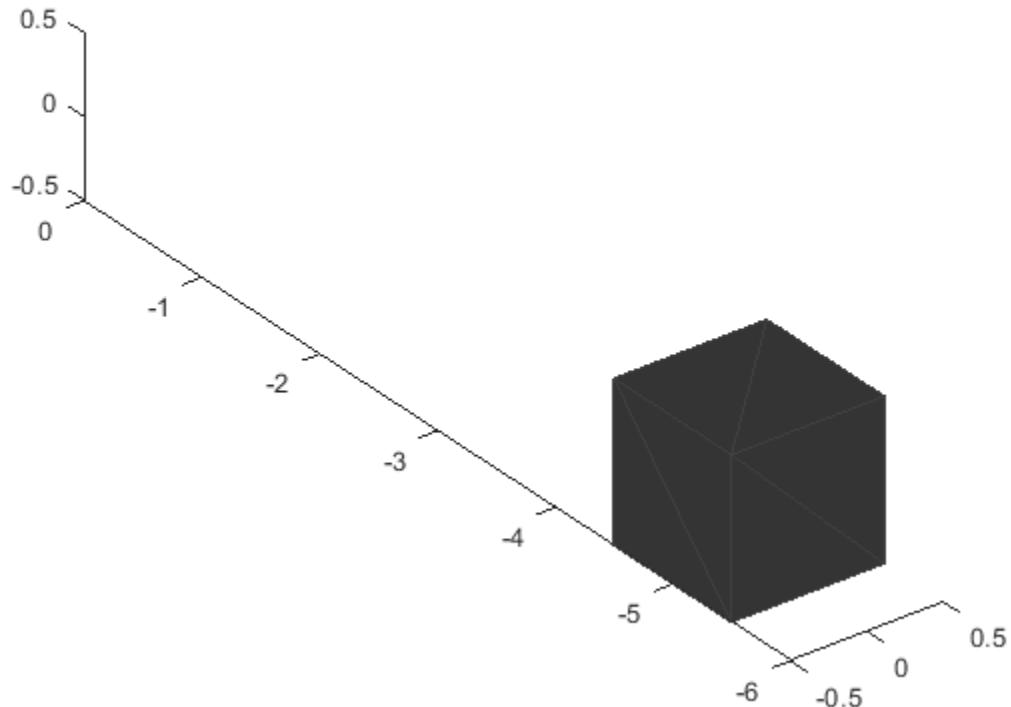
```
mesh = extendedObjectMesh('cuboid');
```

Translate the mesh by 5 units along the negative y axis.

```
mesh = translate(mesh,[0 -5 0]);
```

Visualize the mesh.

```
ax = show(mesh);
ax.YLim = [-6 0];
```



### Create and Visualize Cylinder Mesh

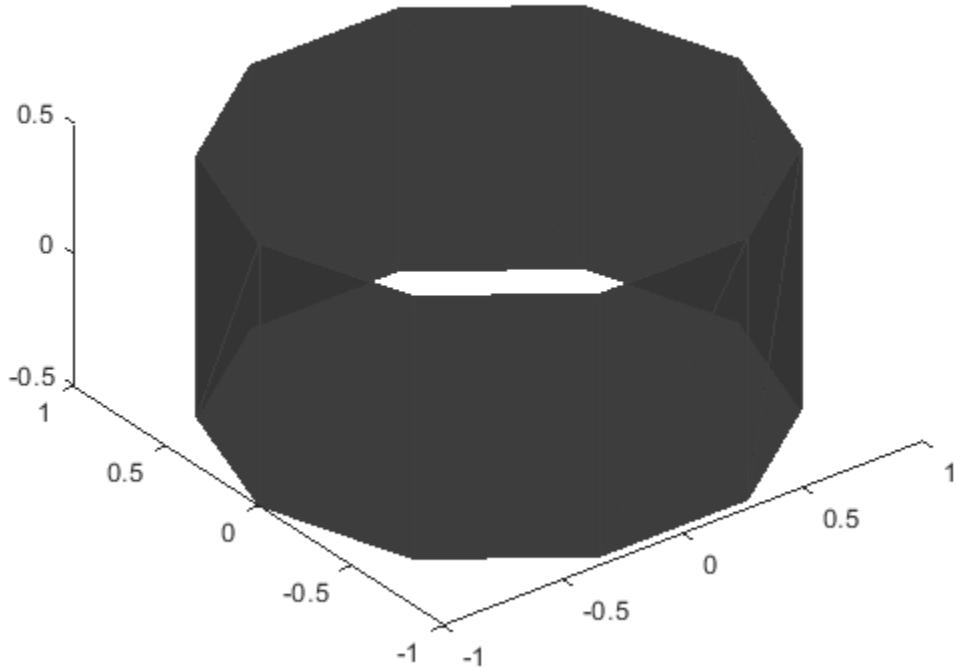
This example shows how to create an `extendedObjectMesh` object and visualize the object.

Construct a cylinder mesh.

```
mesh = extendedObjectMesh('cylinder');
```

Visualize the mesh.

```
ax = show(mesh);
```



### Create and Auto-scale Sphere Mesh

This example shows how to create an `extendedObjectMesh` object and auto-scale the object to the required dimensions.

Construct a sphere mesh of unit dimensions.

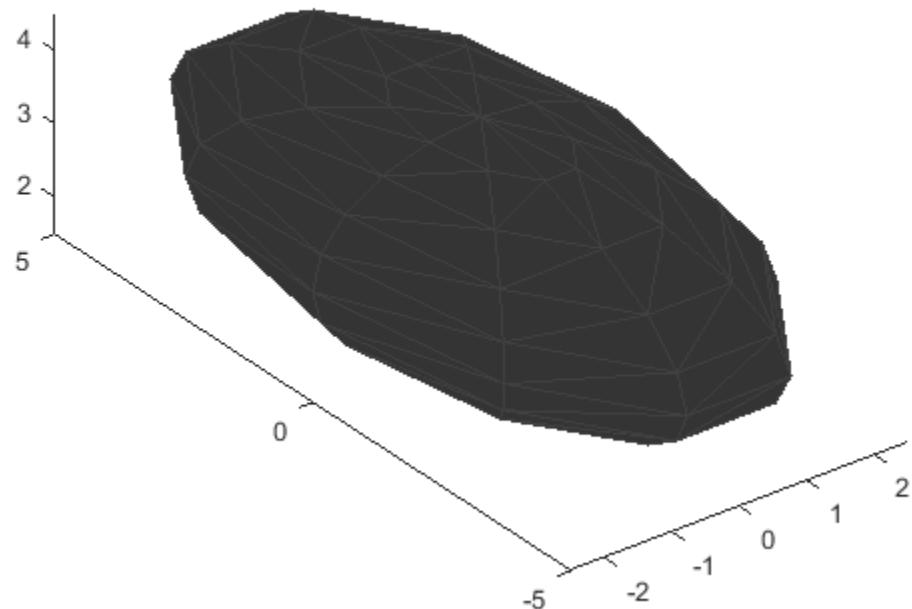
```
sph = extendedObjectMesh('sphere');
```

Auto-scale the mesh to the dimensions in `dims`.

```
dims = struct('Length',5,'Width',10,'Height',3,'OriginOffset',[0 0 -3]);  
sph = scaleToFit(sph,dims);
```

Visualize the mesh.

```
show(sph);
```



## See Also

**Introduced in R2020b**

## applyTransform

Apply forward transformation to mesh vertices

### Syntax

```
transformedMesh = applyTransform(mesh,T)
```

### Description

`transformedMesh = applyTransform(mesh,T)` applies the forward transformation matrix  $T$  to the vertices of the object mesh.

### Examples

#### Create and Transform Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and transform the object in a way defined by a given transformation matrix.

Create a cuboid mesh of unit dimensions.

```
cuboid = extendedObjectMesh('cuboid');
```

Create a transformation matrix that is a combination of a translation, a scaling, and a rotation.

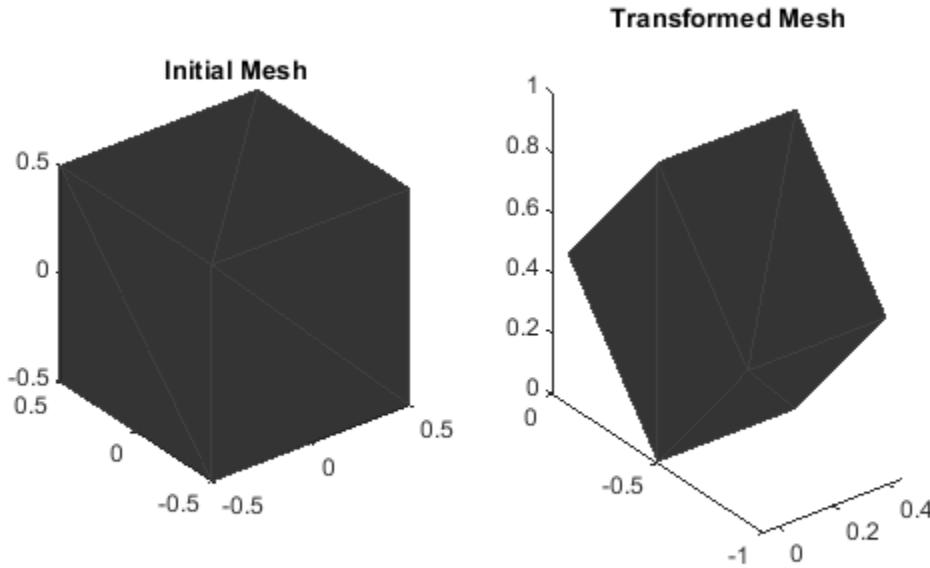
```
T = makehgtform('translate',[0.2 -0.5 0.5], 'scale',[0.5 0.6 0.7], 'xrotate',pi/4);
```

Transform the mesh.

```
transformedCuboid = applyTransform(cuboid,T);
```

Visualize the mesh.

```
subplot(1,2,1);
show(cuboid);
title('Initial Mesh');
subplot(1,2,2);
show(transformedCuboid);
title('Transformed Mesh');
```



## Input Arguments

**mesh — Extended object mesh**  
extendedObjectMesh object

Extended object mesh, specified as an extendedObjectMesh object.

**T — Transformation matrix**  
4-by-4 matrix

Transformation matrix applied on the object mesh, specified as a 4-by-4 matrix. The 3-D coordinates of each point in the object mesh is transformed according to this formula:

$$[x_T; y_T; z_T; 1] = T * [x; y; z; 1]$$

$x_T$ ,  $y_T$ , and  $z_T$  are the transformed 3-D coordinates of the point.

Data Types: single | double

## Output Arguments

**transformedMesh — Transformed object mesh**  
extendedObjectMesh object

Transformed object mesh, returned as an extendedObjectMesh object.

## See Also

### Objects

extendedObjectMesh

### Functions

join | rotate | scale | scaleToFit | show | translate

**Introduced in R2020b**

# join

Join two object meshes

## Syntax

```
joinedMesh = join(mesh1, mesh2)
```

## Description

`joinedMesh = join(mesh1, mesh2)` joins the object meshes `mesh1` and `mesh2` and returns `joinedMesh` with the combined objects.

## Examples

### Create and Join Two Object Meshes

This example shows how to create `extendedObjectMesh` objects and join them together.

Construct two meshes of unit dimensions.

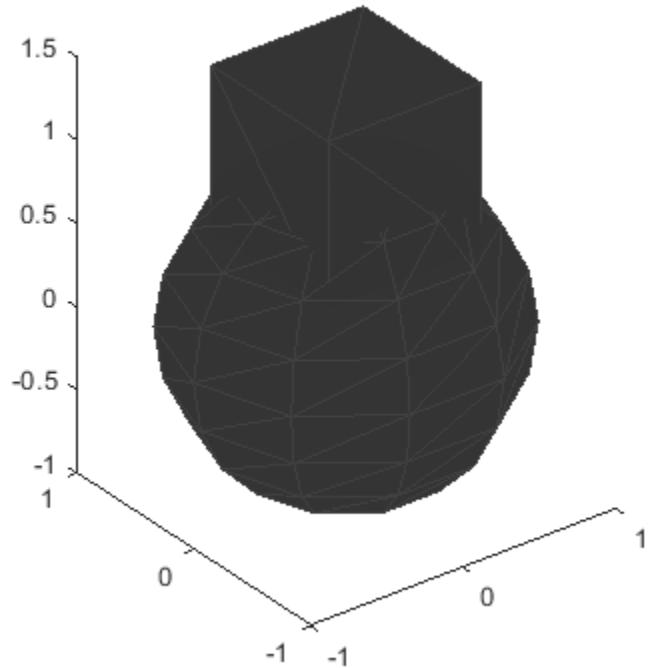
```
sph = extendedObjectMesh('sphere');  
cub = extendedObjectMesh('cuboid');
```

Join the two meshes.

```
cub = translate(cub,[0 0 1]);  
sphCub = join(sph,cub);
```

Visualize the final mesh.

```
show(sphCub);
```



## Input Arguments

### **mesh1 — Extended object mesh**

extendedObjectMesh object

Extended object mesh, specified as an extendedObjectMesh object.

### **mesh2 — Extended object mesh**

extendedObjectMesh object

Extended object mesh, specified as an extendedObjectMesh object.

## Output Arguments

### **joinedMesh — Joined object mesh**

extendedObjectMesh object

Joined object mesh, specified as an extendedObjectMesh object.

## See Also

### **Objects**

extendedObjectMesh

**Functions**

`applyTransform` | `rotate` | `scale` | `scaleToFit` | `show` | `translate`

**Introduced in R2020b**

## rotate

Rotate mesh about coordinate axes

### Syntax

```
rotatedMesh = rotate(mesh,orient)
```

### Description

`rotatedMesh = rotate(mesh,orient)` rotate the mesh object by an orientation, `orient`.

### Examples

#### Create and Rotate Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and rotate the object.

Construct a cuboid mesh.

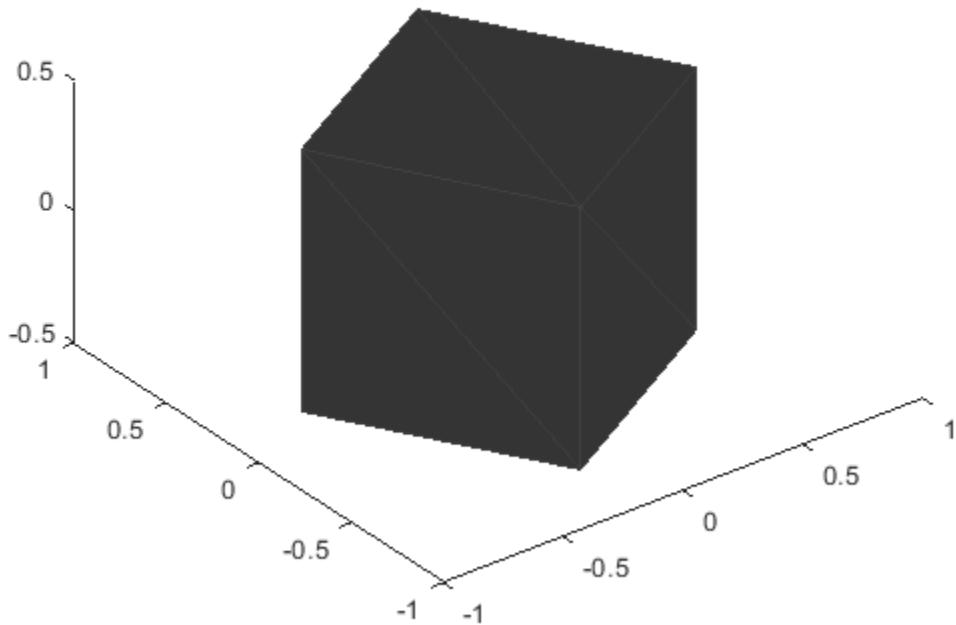
```
mesh = extendedObjectMesh('cuboid');
```

Rotate the mesh by 30 degrees around the *z* axis.

```
mesh = rotate(mesh,[30 0 0]);
```

Visualize the mesh.

```
ax = show(mesh);
```



## Input Arguments

### **mesh — Extended object mesh**

extendedObjectMesh object

Extended object mesh, specified as an extendedObjectMesh object.

### **orient — Description of rotation**

3-by-3 orthonormal matrix | quaternion | 1-by-3 vector

Description of rotation for an object mesh, specified as:

- 3-by-3 orthonormal rotation matrix
- quaternion
- 1-by-3 vector, where the elements are positive rotations in degrees about the  $z$ ,  $y$ , and  $x$  axes, in that order.

## Output Arguments

### **rotatedMesh — Rotated object mesh**

extendedObjectMesh object

Rotated object mesh, returned as an extendedObjectMesh object.

## See Also

### Objects

`extendedObjectMesh`

### Functions

`applyTransform | join | scale | scaleToFit | show | translate`

**Introduced in R2020a**

# scale

Scale mesh in each dimension

## Syntax

```
scaledMesh = scale(mesh,scaleFactor)
scaledMesh = scale(mesh,[sx sy sz])
```

## Description

`scaledMesh = scale(mesh,scaleFactor)` scales the object mesh by `scaleFactor`. `scaleFactor` can be the same for all dimensions or defined separately as elements of a 1-by-3 vector in the order `x`, `y`, and `z`.

`scaledMesh = scale(mesh,[sx sy sz])` scales the object mesh along the dimensions `x`, `y`, and `z` by the scaling factors `sx`, `sy`, and `sz`.

## Examples

### Create and Scale Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and scale the object.

Construct a cuboid mesh of unit dimensions.

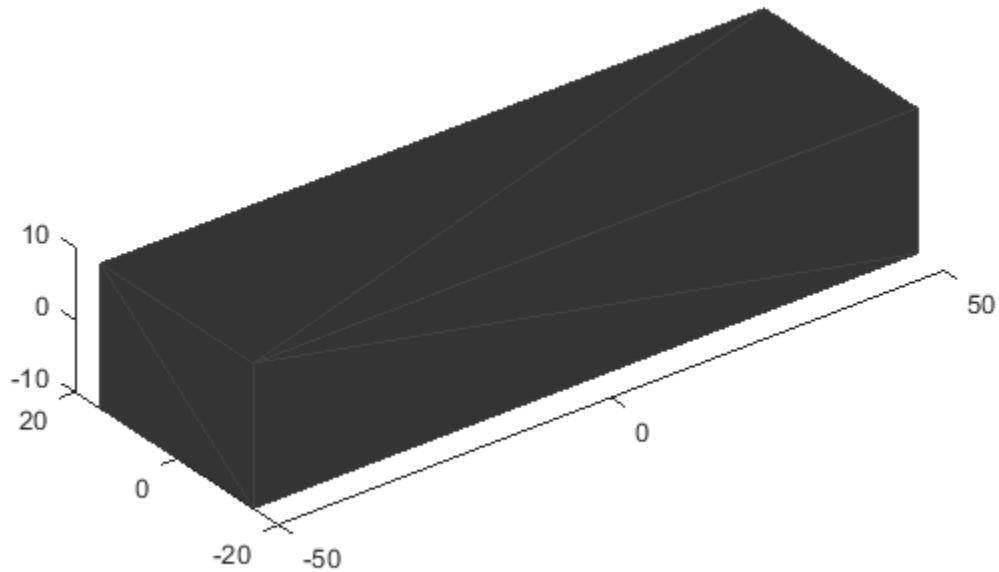
```
cuboid = extendedObjectMesh('cuboid');
```

Scale the mesh by different factors along each of the three axes.

```
scaledCuboid = scale(cuboid,[100 30 20]);
```

Visualize the mesh.

```
show(scaledCuboid);
```



## Input Arguments

**mesh — Extended object mesh**  
extendedObjectMesh object

Extended object mesh, specified as an extendedObjectMesh object.

**scaleFactor — Scaling factor**  
positive real scalar | 1-by-3 vector

Scaling factor for the object mesh, specified as a single positive real value or as a 1-by-3 vector in the order x, y, and z.

Data Types: single | double

**sx — Scaling factor for x-axis**  
positive real scalar

Scaling factor for x-axis, specified as a positive real scalar.

Data Types: single | double

**sy — Scaling factor for y-axis**  
positive real scalar

Scaling factor for y-axis, specified as a positive real scalar.

Data Types: `single` | `double`

**sz — Scaling factor for z-axis**

positive real scalar

Scaling factor for *z*-axis, specified as a positive real scalar.

Data Types: `single` | `double`

## Output Arguments

**scaledMesh — Scaled object mesh**

`extendedObjectMesh` object

Scaled object mesh, returned as an `extendedObjectMesh` object.

## See Also

**Objects**

`extendedObjectMesh`

**Functions**

`applyTransform` | `join` | `rotate` | `scaleToFit` | `show` | `translate`

**Introduced in R2020b**

## scaleToFit

Auto-scale object mesh to match specified cuboid dimensions

### Syntax

```
scaledMesh = scaleToFit(mesh,dims)
```

### Description

`scaledMesh = scaleToFit(mesh,dims)` auto-scales the object mesh to match the dimensions of a cuboid specified in the structure `dims`.

### Examples

#### Create and Auto-scale Sphere Mesh

This example shows how to create an `extendedObjectMesh` object and auto-scale the object to the required dimensions.

Construct a sphere mesh of unit dimensions.

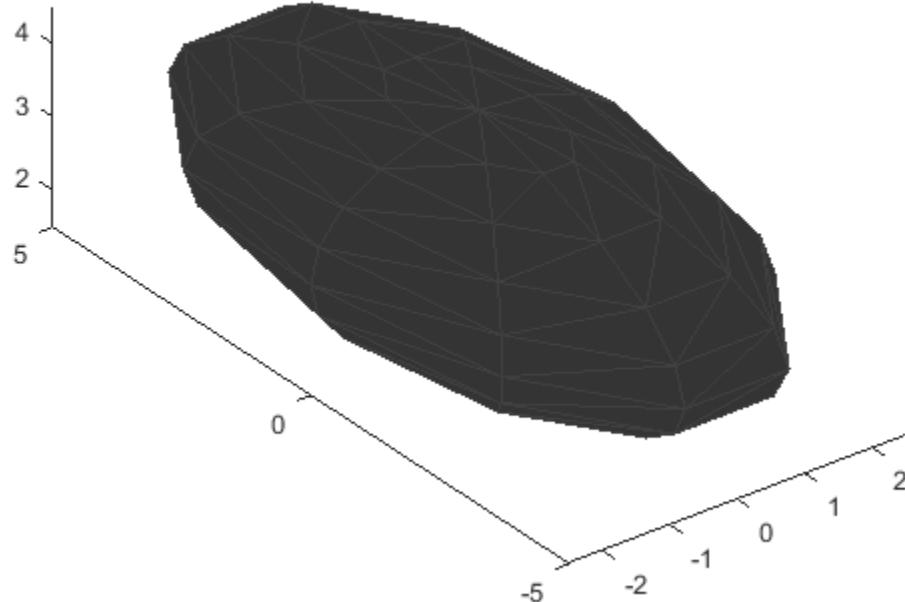
```
sph = extendedObjectMesh('sphere');
```

Auto-scale the mesh to the dimensions in `dims`.

```
dims = struct('Length',5,'Width',10,'Height',3,'OriginOffset',[0 0 -3]);  
sph = scaleToFit(sph,dims);
```

Visualize the mesh.

```
show(sph);
```



## Input Arguments

### **mesh — Extended object mesh**

extendedObjectMesh object

Extended object mesh, specified as an extendedObjectMesh

### **dims — Cuboid dimensions**

struct

Dimensions of the cuboid to scale an object mesh, specified as a struct with these fields:

- **Length** – Length of the cuboid
- **Width** – Width of the cuboid
- **Height** – Height of the cuboid
- **OriginOffset** – Origin offset in 3-D coordinates

All the dimensions are in meters.

Data Types: struct

## Output Arguments

**scaledMesh — Scaled object mesh**  
extendedObjectMesh object

Scaled object mesh, returned as an extendedObjectMesh object.

## See Also

**Objects**  
extendedObjectMesh

**Functions**  
applyTransform | join | rotate | scale | show | translate

**Introduced in R2020b**

# show

Display the mesh as a patch on the current axes

## Syntax

```
show(mesh)
show(mesh,ax)
ax = show(mesh)
```

## Description

`show(mesh)` displays the as a patch on the current axes. If there are no active axes, the function creates new axes.

`show(mesh,ax)` displays the object mesh as a patch on the axes `ax`.

`ax = show(mesh)` optionally outputs the handle to the axes where the mesh was plotted.

## Examples

### Create and Translate Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

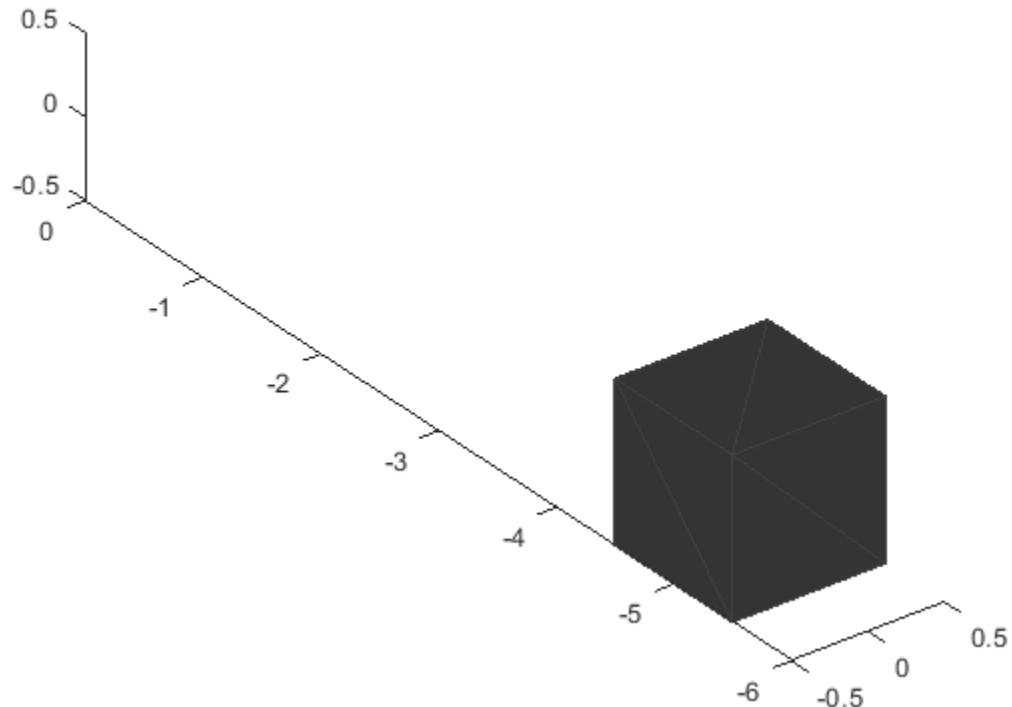
```
mesh = extendedObjectMesh('cuboid');
```

Translate the mesh by 5 units along the negative y axis.

```
mesh = translate(mesh,[0 -5 0]);
```

Visualize the mesh.

```
ax = show(mesh);
ax.YLim = [-6 0];
```



## Input Arguments

**mesh — Extended object mesh**

extendedObjectMesh object

Extended object mesh, specified as an extendedObjectMesh object.

**ax — Current axes**

axes

Current axes, specified as an axes object.

## See Also

**Objects**

extendedObjectMesh

**Functions**

applyTransform | join | rotate | scale | scaleToFit | translate

**Introduced in R2020b**

# translate

Translate mesh along coordinate axes

## Syntax

```
translatedMesh = translate(mesh,deltaPos)
```

## Description

`translatedMesh = translate(mesh,deltaPos)` translates the object mesh by the distances specified by `deltaPos` along the coordinate axes.

## Examples

### Create and Translate Cuboid Mesh

This example shows how to create an `extendedObjectMesh` object and translate the object.

Construct a cuboid mesh.

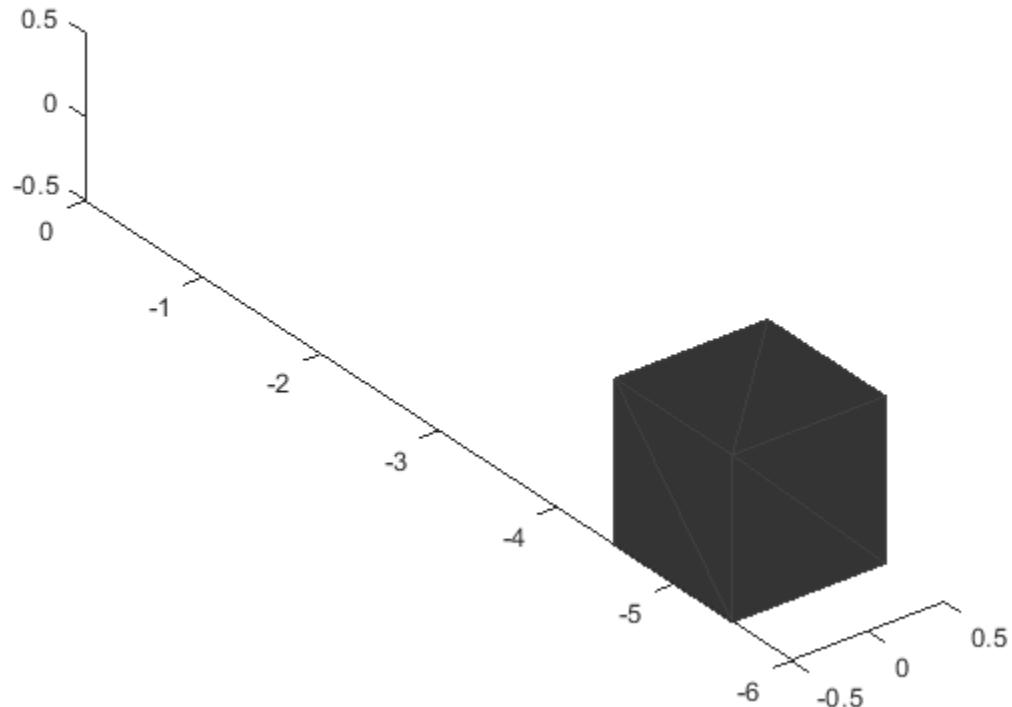
```
mesh = extendedObjectMesh('cuboid');
```

Translate the mesh by 5 units along the negative y axis.

```
mesh = translate(mesh,[0 -5 0]);
```

Visualize the mesh.

```
ax = show(mesh);  
ax.YLim = [-6 0];
```



## Input Arguments

**mesh — Extended object mesh**

extendedObjectMesh object

Extended object mesh, specified as an extendedObjectMesh object.

**deltaPos — Translation vector**

three-element, real-valued vector

Translation vector for an object mesh, specified as a three-element, real-valued vector. The three elements in the vector define the translation along the  $x$ ,  $y$ , and  $z$  axes.

Data Types: `single` | `double`

## Output Arguments

**translatedMesh — Translated object mesh**

extendedObjectMesh object

Translated object mesh, returned as an extendedObjectMesh object.

## See Also

### Objects

extendedObjectMesh

### Functions

applyTransform | join | rotate | scale | scaleToFit | show

**Introduced in R2020b**

# tracking.scenario.airplaneMesh

Mesh representation of airplane

## Syntax

```
mesh = tracking.scenario.airplaneMesh
```

## Description

`mesh = tracking.scenario.airplaneMesh` returns an `extendedObjectMesh` object defining an airplane mesh that can be used with the `trackingScenario` object.

## Examples

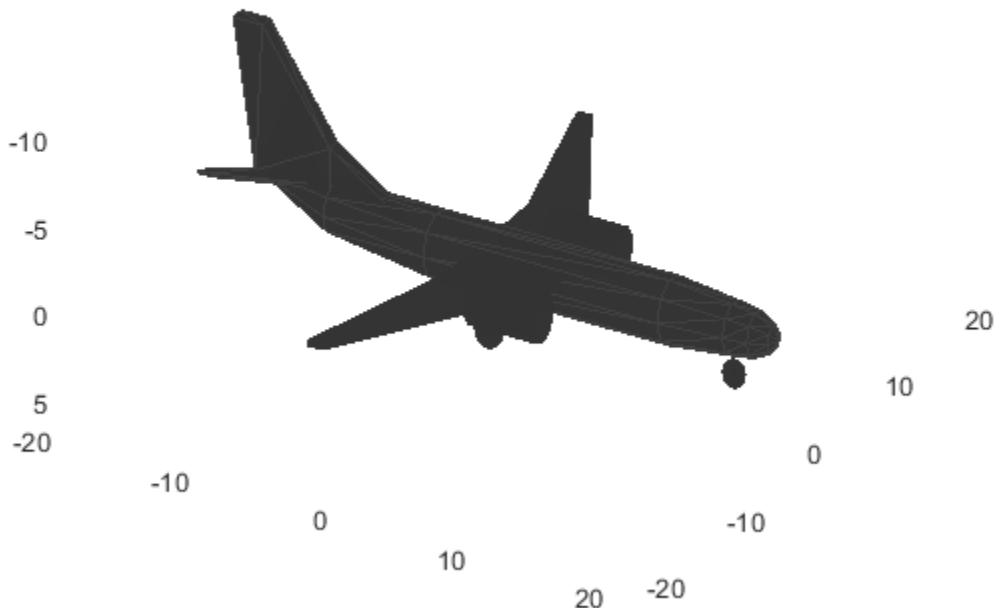
### Create and Visualize Airplane Mesh

Create the airplane mesh.

```
mesh = tracking.scenario.airplaneMesh;
```

Visualize the mesh.

```
ax = axes('ZDir','reverse');  
show(mesh,ax);  
view(30,50);
```



### Simulate Lidar Detection with Airplane Mesh

Create a tracking scenario object and an airplane mesh object.

```
scene = trackingScenario;
mesh = tracking.scenario.airplaneMesh;
```

Create two tower platforms.

```
% Create the first tower.
tower = platform(scene);
h = 50;
tower.Trajectory.Position = [0 0 -h];
tower.Dimensions = struct('Length',10,'Width',10,'Height',h,'OriginOffset',[0 0 -h/2]);
tower.Sensors = monostaticLidarSensor('SensorIndex',1,...
    'MaxRange',200,...
    'HasINS',true,...
    'DetectionCoordinates','scenario',...
    'AzimuthLimits',[-75 75],...
    'ElevationLimits',[-10 30]);

% Create the second tower.
tower2 = platform(scene);
h = 50;
tower2.Trajectory.Position = [0 500 -h];
```

```
tower2.Dimensions = struct('Length',10,'Width',10,'Height',h,'OriginOffset',[0 0 -h/2]);
tower2.Sensors = monostaticLidarSensor('SensorIndex',2,...  
    'MaxRange',200,...  
    'HasINS',true,...  
    'DetectionCoordinates','scenario',...  
    'AzimuthLimits',[-75 75],...  
    'ElevationLimits',[-10 30]);
```

Create the airplane target with associated mesh.

```
airplane = platform(scene);
airplane.Mesh = mesh;
% Set the dimensions of the plane which automatically adjust the size of the mesh.
airplane.Dimensions = struct('Length',40,...  
    'Width',40,...  
    'Height',12.5,...  
    'OriginOffset',[0 0 12.5/2]);
```

Create a landing trajectory for the plane.

```
x = 50*ones(10,1);
y = linspace(-500,1000,10)';
yToLand = max(0,-y);
z = -1e4*(2.*((yToLand./50e3).^3 + 3*(yToLand./50e3).^2));
wps = [x y z];
toa = linspace(0,30,10)';
traj = waypointTrajectory(wps,toa);
airplane.Trajectory = traj;
```

Create a plotter to visualize the scenario.

```
lp = scatter3(nan,nan,nan,6,nan,'o','DisplayName','Lidar data');
tp = theaterPlot('Parent',lp.Parent,...  
    'XLimits',[0 100],...  
    'YLimits',[-500 1000],...  
    'ZLimits',[-75 0]);
lp.Parent.ZDir = 'reverse';
view(lp.Parent,169,5);
pp = platformPlotter(tp,'DisplayName','Platforms','Marker','^');
cp = coveragePlotter(tp,'DisplayName','Lidar coverage');
hold on;
```

Advance the simulation, generate data, and visualize the results.

```
scene.UpdateRate = 0; % Automatic update rate
while advance(scene)
    % Generate point cloud.
    ptCloud = lidarDetect(scene);

    % Obtain coverage configurations.
    cfgs = coverageConfig(scene);

    % Plot coverage.
    cp.plotCoverage(cfgs);

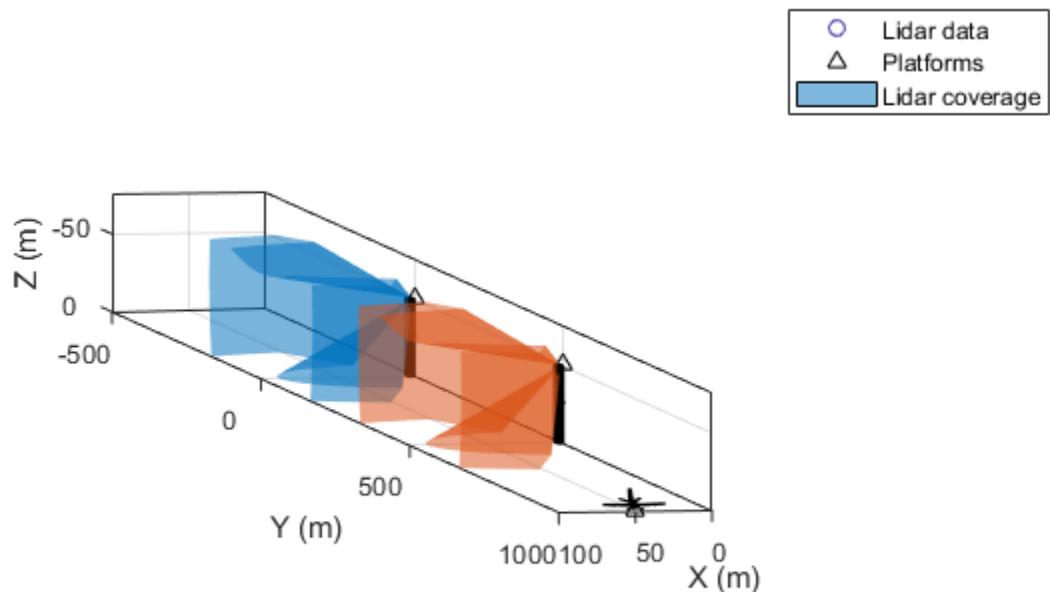
    % Plot platforms.
    platPoses = platformPoses(scene);
    pos = vertcat(platPoses.Position);
    mesh = cellfun(@(x)x.Mesh,scene.Platforms);
```

```

orient = vertcat(platPoses.Orientation);
pp.plotPlatform(pos,mesh,orient);

% Concatenate all point clouds.
s = vertcat(ptCloud{:});
% Plot lidar data.
set(lp,'XData',s(:,1),...
    'YData',s(:,2),...
    'ZData',s(:,3),...
    'CData',s(:,3));
drawnow;
end

```



## Output Arguments

### **mesh — Airplane mesh**

extendedObjectMesh object

Airplane mesh, returned as an extendedObjectMesh object defining the mesh of an airplane.

## See Also

[extendedObjectMesh](#) | [lidarDetect](#) | [monostaticLidarSensor](#)

**Introduced in R2020b**

# trackingKF

Linear Kalman filter for object tracking

## Description

A `trackingKF` object is a discrete-time linear Kalman filter used to track the 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 is linear when the evolution of the state follows a linear motion model and the measurements are linear functions of the state. The filter assumes that both the process and measurements have additive noise. When the process noise and measurement noise are Gaussian, the Kalman filter is the optimal minimum mean squared error (MMSE) state estimator for linear processes.

You can use this object in these ways:

- Explicitly set the motion model. Set the motion model property, `MotionModel`, to `Custom`, and then use the `StateTransitionModel` property to set the state transition matrix.
- Set the `MotionModel` property to a predefined state transition model:

Motion Model
'1D Constant Velocity'
'1D Constant Acceleration'
'2D Constant Velocity'
'2D Constant Acceleration'
'3D Constant Velocity'
'3D Constant Acceleration'

## Creation

### Syntax

```
filter = trackingKF
filter = trackingKF(F,H)
filter = trackingKF(F,H,G)
filter = trackingKF('MotionModel',model)
filter = trackingKF(___,Name,Value)
```

### Description

`filter = trackingKF` creates a linear Kalman filter object for a discrete-time, 2-D, constant-velocity moving object. The Kalman filter uses default values for the `StateTransitionModel`, `MeasurementModel`, and `ControlModel` properties. The function also sets the `MotionModel` property to '`2D Constant Velocity`'.

`filter = trackingKF(F,H)` specifies the state transition model, F, and the measurement model, H. With this syntax, the function also sets the `MotionModel` property to 'Custom'.

`filter = trackingKF(F,H,G)` also specifies the control model, G. With this syntax, the function also sets the `MotionModel` property to 'Custom'.

`filter = trackingKF('MotionModel',model)` sets the motion model property, `MotionModel`, to `model`.

`filter = trackingKF(____,Name,Value)` configures the properties of the Kalman filter by using one or more `Name,Value` pair arguments and any of the previous syntaxes. Any unspecified properties take default values.

## 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.  $M$  is the size of the state vector. Typical state vector sizes are described in the `MotionModel` property. When the initial state is specified as a scalar, the state is expanded into an  $M$ -element vector.

You can set the state to a scalar in these cases:

- When the `MotionModel` property is set to 'Custom',  $M$  is determined by the size of the state transition model.
- When the `MotionModel` property is set to '2D Constant Velocity', '3D Constant Velocity', '2D Constant Acceleration', or '3D Constant Acceleration', you must first specify the state as an  $M$ -element vector. You can use a scalar for all subsequent specifications of the state vector.

Example: [200; 0.2; -40; -0.01]

Data Types: double

### StateCovariance — State estimation error covariance

1 (default) | positive scalar | positive-definite real-valued  $M$ -by- $M$  matrix

State 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. Specifying the value as a scalar creates a multiple of the  $M$ -by- $M$  identity matrix. This matrix represents the uncertainty in the state.

Example: [20 0.1; 0.1 1]

Data Types: double

### MotionModel — Kalman filter motion model

'Custom' (default) | '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. In this case, the state vector and state transition matrix take the form specified in the table.

Motion Model	Form of State Vector	Form of State Transition Model
'1D Constant Velocity'	[x;vx]	[1 dt; 0 1]
'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
'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.
'1D Constant Acceleration'	[x;vx;ax]	[1 dt 0.5*dt^2; 0 1 dt; 0 0 1]
'2D Constant Acceleration'	[x;vx;ax;y;vy;ay]	Block diagonal matrix with [1 dt 0.5*dt^2; 0 1 dt; 0 0 1] blocks repeated for the x and y spatial dimensions
'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

When the `ControlModel` property is defined, every nonzero element of the state transition model is replaced by `dt`.

When `MotionModel` is 'Custom', you must specify a state transition model matrix, a measurement model matrix, and optionally, a control model matrix as input arguments to the Kalman filter.

Data Types: char

#### **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 relates the state at any time step to the state at the previous step. The state transition model is a function of the filter time step size.

Example: [1 0; 1 2]

#### **Dependencies**

To enable this property, set `MotionModel` to 'Custom'.

Data Types: double

#### **ControlModel — Control model**

[] (default) |  $M$ -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.

Example: [.01 0.2]

Data Types: double

#### **ProcessNoise — Covariance of process noise**

[1 (default) | positive scalar | real-valued positive-definite  $M$ -by- $M$  matrix]

Covariance of process noise, specified as a positive scalar or an  $M$ -by- $M$  matrix where  $M$  is the dimension of the state. If you specify this property as a scalar, the filter uses the value as a multiplier of the  $M$ -by- $M$  identity matrix. Process noise expresses the uncertainty in the dynamic model and is assumed to be zero-mean Gaussian white noise.

---

**Tip** If you specify the `MotionModel` property as any of the predefined motion model, then the corresponding process noise is automatically generated during construction and updated during propagation. In this case, you do not need to specify the `ProcessNoise` property. In fact, the filter neglects your process noise input during object construction. If you want to specify the process noise other than the default values, use the `trackingEKF` object.

---

Data Types: 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 predicted measurements from the predicted state.

Example: [1 0.5 0.01; 1.0 1 0]

Data Types: 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 filter uses the value as a multiplier of the  $N$ -by- $N$  identity matrix. Measurement noise represents the uncertainty of the measurement and is assumed to be zero-mean Gaussian white noise.

Example: 0.2

Data Types: double

## **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**

### 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 object's x-y location.

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 the measured positions from a constant-velocity trajectory.

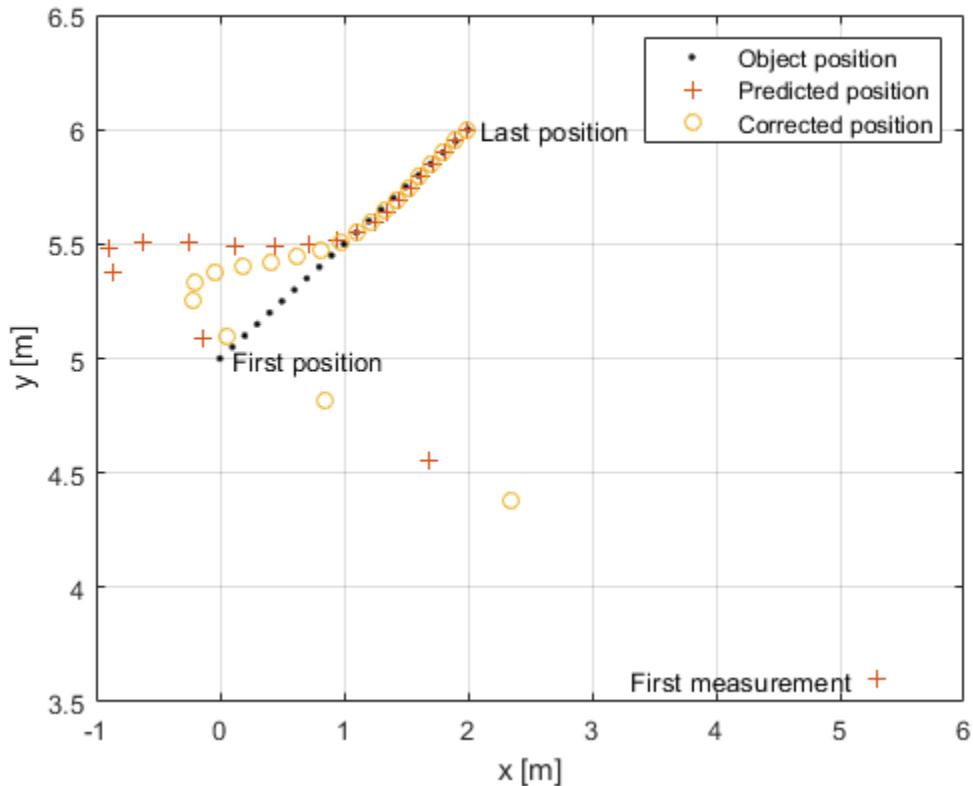
```
vx = 0.2;
vy = 0.1;
T = 0.5;
pos = [0:vx*T:2;5:vy*T:6]';
```

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')
```



## 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.  $L$  is the size of the control model.

Model Parameter	Description	Filter Property	Size
$F_k$	State transition model that specifies a linear model of the force-free equations of motion of the object. This model, together with the control model, determines the state at time $k+1$ as a function of the state at time $k$ . The state transition model depends on the time step of the filter.	StateTransitionModel	$M$ -by- $M$

Model Parameter	Description	Filter Property	Size
$H_k$	Measurement model that specifies how the measurements are linear functions of the state.	MeasurementModel	$N$ -by- $M$
$G_k$	Control model describing the controls or forces acting on the object.	ControlModel	$M$ -by- $L$
$x_k$	Estimate of the state of the object.	State	$M$ -
$P_k$	Estimated covariance matrix of the state. The covariance represents 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 a measure of the uncertainty in your dynamic model and is assumed to be zero-mean white Gaussian noise.	ProcessNoise	$M$ -by- $M$
$R_k$	Estimate of the measurement noise covariance at step $k$ . Measurement noise represents the uncertainty of the measurement and is assumed to be zero-mean white Gaussian noise.	MeasurementNoise	$N$ -by- $N$

## Algorithms

The Kalman filter describes the motion of an object by estimating its state. The state generally consists of object position and velocity and possibly its acceleration. The state can span one, two, or three spatial dimensions. Most frequently, you use the Kalman filter to model constant-velocity or constant-acceleration motion. A linear Kalman filter assumes that the process obeys the following linear stochastic difference equation:

$$x_{k+1} = F_k x_k + G_k u_k + v_k$$

$x_k$  is the state at step  $k$ .  $F_k$  is the state transition model matrix.  $G_k$  is the control model matrix.  $u_k$  represents known generalized controls acting on the object. In addition to the specified equations of motion, the motion may be affected by random noise perturbations,  $v_k$ . The state, the state transition

matrix, and the controls together provide enough information to determine the future motion of the object in the absence of noise.

In the Kalman filter, the measurements are also linear functions of the state,

$$z_k = H_k x_k + w_k$$

where  $H_k$  is the measurement model matrix. This model expresses the measurements as functions of the state. A measurement can consist of an object position, position and velocity, or its position, velocity, and acceleration, or some function of these quantities. The measurements can also include noise perturbations,  $w_k$ .

These equations, in the absence of noise, model the actual motion of the object and the actual measurements. The noise contributions at each step are unknown and cannot be modeled. Only the noise covariance matrices are known. The state covariance matrix is updated with knowledge of the noise covariance only.

For a brief description of the linear Kalman filter algorithm, see "Linear Kalman Filters".

## 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:

- When 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. However, motion models do not specify the data type, for example, double precision or single precision. Both size and data type are required for code generation.

## See Also

### Functions

`initcakf` | `initcvkf`

### Objects

`trackerGNN` | `trackerTOMHT` | `trackingABF` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

### Topics

"Linear Kalman Filters"

**Introduced in R2018b**

# trackingEKF

Extended Kalman filter for object tracking

## Description

A `trackingEKF` object is a discrete-time extended Kalman filter used to track the 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 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.

## 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.

Example: [200; 0.2]

Data Types: 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.

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 ( <code>HasAdditiveProcessNoise = true</code> )	Valid Syntaxes ( <code>HasAdditiveProcessNoise = false</code> )
<pre>x(k) = statetransitionfcn(x(k-1)) x(k) = statetransitionfcn(x(k-1),parameters)</pre> <ul style="list-style-type: none"> <li>• <math>x(k)</math> is the state at time <math>k</math>.</li> <li>• <code>parameters</code> stands for all additional arguments required by the state transition function.</li> </ul>	<pre>x(k) = statetransitionfcn(x(k-1),w(k-1)) x(k) = statetransitionfcn(x(k-1),w(k-1),dt) x(k) = statetransitionfcn(__,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 value for the process noise at time <math>k</math>.</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 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 ( <code>HasAdditiveProcessNoise = true</code> )	Valid Syntaxes ( <code>HasAdditiveProcessNoise = false</code> )
<pre>Jx(k) = statejacobianfcn(x(k)) Jx(k) = statejacobianfcn(x(k),parameters)</pre> <ul style="list-style-type: none"> <li>• <code>x(k)</code> is the state at time <code>k</code>.</li> <li>• <code>Jx(k)</code> 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 <code>k</code>. 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>• <code>x(k)</code> is the state at time <code>k</code></li> <li>• <code>w(k)</code> is a sample <math>Q</math>-element vector of the process noise at time <code>k</code>. <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>• <code>Jx(k)</code> 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 <code>k</code>. The Jacobian function can take additional input parameters, such as control inputs or time-step size.</li> <li>• <code>Jw(k)</code> 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, 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. This function can be a nonlinear function that models measurements from the predicted state. Input to the function is the  $M$ -element state vector. The output is the  $N$ -element measurement vector. The function can take additional input arguments, such as sensor position and orientation.

- 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.

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 sensor position and orientation.

- If `HasAdditiveMeasurementNoise` is `true`, specify the Jacobian function using one of these syntaxes:

```
Jmx(k) = measjacobianfcn(x(k))
```

```
Jmx(k) = measjacobianfcn(x(k),parameters)
```

$x(k)$  is the state at time  $k$ .  $Jx(k)$  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:

```
[Jmx(k),Jmv(k)] = measjacobianfcn(x(k),v(k))
```

```
[Jmx(k),Jmv(k)] = measjacobianfcn(x(k),v(k),parameters)
```

$x(k)$  is the state at time  $k$  and  $v(k)$  is an  $R$ -dimensional sample noise vector.  $Jmx(k)$  denotes the  $N$ -by- $M$  Jacobian of the measurement function with respect to the state.  $Jmv(k)$  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`

#### **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.

## 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
$P_k$	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.

Filter Parameter	Description	Filter Property	Size
$F$	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™.

## See Also

### Functions

`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac` | `initcaekf` | `initctekf` | `initcvekf`

### Objects

`trackerGNN` | `trackerTOMHT` | `trackingABF` | `trackingCKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

### Topics

“Extended Kalman Filters”

### Introduced in R2018b

# 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 objects target platforms.

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.

Example: [200; 0.2]

Data Types: 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.

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 ( <code>HasAdditiveProcessNoise = true</code> )	Valid Syntaxes ( <code>HasAdditiveProcessNoise = false</code> )
<pre>x(k) = statetransitionfcn(x(k-1)) x(k) = statetransitionfcn(x(k-1),parameters)</pre> <ul style="list-style-type: none"> <li>• <math>x(k)</math> is the state at time <math>k</math>.</li> <li>• <code>parameters</code> stands for all additional arguments required by the state transition function.</li> </ul>	<pre>x(k) = statetransitionfcn(x(k-1),w(k-1)) x(k) = statetransitionfcn(x(k-1),w(k-1),dt) x(k) = statetransitionfcn(__,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 value for the process noise at time <math>k</math>.</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. This function can be a nonlinear function that models measurements from the predicted state. Input to the function is the  $M$ -element state vector. The output is the  $N$ -element measurement vector. The function can take additional input arguments, such as sensor position and orientation.

- 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.

Example: @cameas

Data Types: function\_handle

**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.

## 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 = 4x1`

```
1.2500
0.2500
1.2500
0.2500
```

`Ppred = 4x4`

```
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
<code>f</code>	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.	<code>StateTransitionFcn</code>	Function returns $M$ -element vector

Model Parameter	Description	Filter Property	Size
$h$	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	$M$
$P_k$	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 HasAdditiveProcess Noise is true. $Q$ -by- $Q$ when HasAdditiveProcess Noise 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™.

## See Also

### Functions

`cameas` | `cameasjac` | `constacc` | `constaccjac` | `constturn` | `constturnjac` | `constvel` | `constveljac` | `ctmeas` | `ctmeasjac` | `cvmeas` | `cvmeasjac` | `initcaukf` | `initctukf` | `initcvukf`

### Objects

`trackerGNN` | `trackerTOMHT` | `trackingABF` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF`

**Introduced in R2018b**

# 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 `sonarEmission` 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**

[1;5] | 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. Each component must lie in the interval (0, 180]. Units are in degrees.

Example: [14;7]

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 ESM Sensor

Create an radar emission and then detect the emission using a `radarSensor` object.

First, create an 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 `radarSensor`.

```
sensor = radarSensor(1);
```

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]
    MeasurementParameters: [1x1 struct]

```

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[emissionsInBody](#) | [radarChannel](#) | [radarEmitter](#) | [sonarEmission](#)

### Introduced in R2018b

# sonarEmission

Emitted sonar signal structure

## Description

The `sonarEmission` class creates a sonar emission object. This object contains all the properties that describe a signal radiated by a sonar source.

## Creation

### Syntax

```
signal = sonarEmission  
signal = sonarEmission(Name,Value)
```

### Description

`signal = sonarEmission` creates a `sonarEmission` object with default properties. The object represents sonar signals from emitters, channels, and sensors.

`signal = sonarEmission(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 sonar emission**

[180, 180] | two-element vector of positive real values

Field of view of emission, specified as a two-element vector of positive real values, [azfov, elfov]. The field of view defines the total angular extent of the signal emitted. Each component must lie in the interval (0, 180]. Units are in degrees.

Example: [14, 7]

Data Types: double

**SourceLevel — Cumulative source level**

0 (default) | scalar

Cumulative source level of an emitted signal, specified as a scalar. The cumulative source level of the emitted signal in decibels is relative to the intensity of a sound wave having an rms pressure of 1 micro-pascal. Units are in dB // 1 micro-pascal.

Example: 10

Data Types: double

**TargetStrength — Cumulative target strength**

0 (default) | scalar

Cumulative target strength of the source platform emitting the signal, specified as a scalar. Units are in dB.

Example: 10

Data Types: double

**CenterFrequency — Center frequency of sonar signal**

20e3 (default) | positive scalar

Center frequency of the signal, specified as a positive scalar. Units are in Hz.

Example: 10.5e3

Data Types: double

**Bandwidth — Half-power bandwidth of sonar signal**

2e3 (default) | positive scalar

Half-power bandwidth of the sonar signal, specified as a positive scalar. Units are in Hz.

Example: 1e3

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 Sonar Emission Object

Create a `sonarEmission` object with specified properties.

```
signal = sonarEmission('PlatformID',6,'EmitterIndex',2, ...
    'OriginPosition',[100,3000,50],'TargetStrength',20, ...
    'CenterFrequency',20e3,'Bandwidth',500.0)

signal =
    sonarEmission with properties:

        PlatformID: 6
        EmitterIndex: 2
        OriginPosition: [100 3000 50]
        OriginVelocity: [0 0 0]
        Orientation: [1x1 quaternion]
        FieldOfView: [180 180]
        CenterFrequency: 20000
        Bandwidth: 500
        WaveformType: 0
        ProcessingGain: 0
        PropagationRange: 0
        PropagationRangeRate: 0
        SourceLevel: 0
        TargetStrength: 20
```

## Detect Sonar Emission with Passive Sensor

Create a sonar emission and then detect the emission using a `sonarSensor` object.

First, create a sonar emission.

```
orient = quaternion([180 0 0],'eulerd','zyx','frame');
sonarSig = sonarEmission('PlatformID',1,'EmitterIndex',1, ...
    'OriginPosition',[30 0 0],'Orientation',orient, ...
    'SourceLevel',140,'TargetStrength',100);
```

Then create a passive sonar sensor.

```
sensor = sonarSensor(1,'No scanning');
```

Detect the sonar emission.

```
time = 0;
[dets, numDets, config] = sensor(sonarSig,time)

dets = 1x1 cell array
    {1x1 objectDetection}

numDets = 1

config = struct with fields:
    SensorIndex: 1
    IsValidTime: 1
    IsScanDone: 1
    FieldOfView: [1 5]
    MeasurementParameters: [1x1 struct]
```

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`emissionsInBody` | `emissionsInBody` | `radarEmission` | `sonarEmitter` | `underwaterChannel`

**Introduced in R2018b**

## theaterPlot

Plot objects, detections, and tracks in trackingScenario

### Description

The `theaterPlot` object is used to display a plot of a `trackingScenario`. This type of plot can be used with sensors capable of detecting objects.

To display aspects of a driving scenario on a bird's-eye plot:

- 1 Create a `theaterPlot` object.
- 2 Create plotters for the aspects of the tracking 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 tracking scenario aspect that you want to plot.

Tracking 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 2-423 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. See “Orientation, Position, and Coordinate” for coordinate system definitions.

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. See “Orientation, Position, and Coordinate” for coordinate system definitions.

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. See “Orientation, Position, and Coordinate” for coordinate system definitions.

Data Types: double

## 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

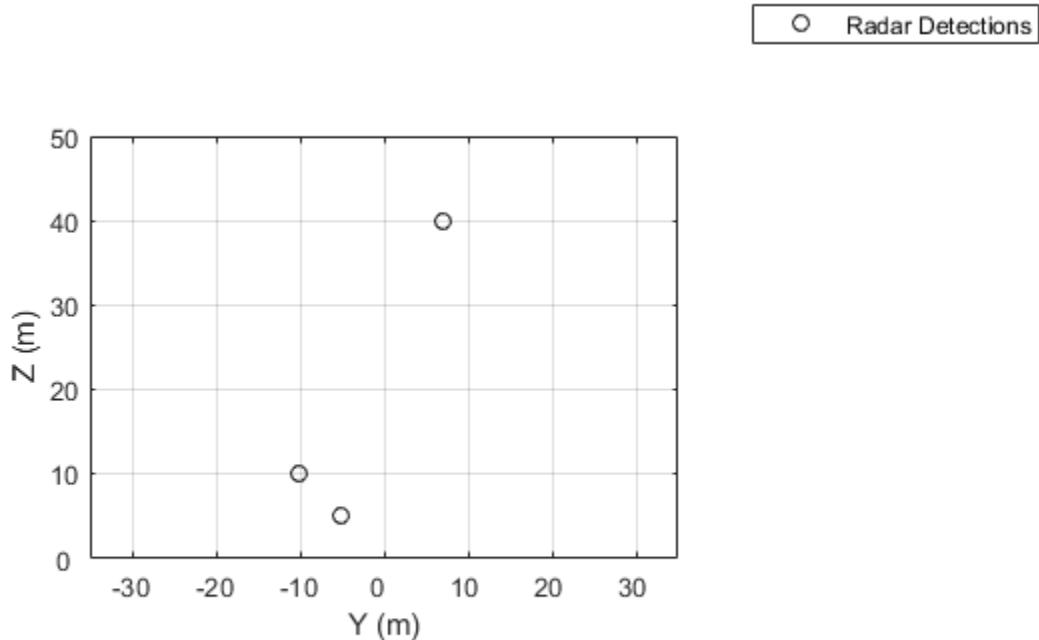
### Create and Display Theater Plot

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

[trackingScenario](#)

**Introduced in R2018b**

## clearPlotterData

Clear plotter data from theater plot

### Syntax

```
clearPlotterData(tp)
```

### Description

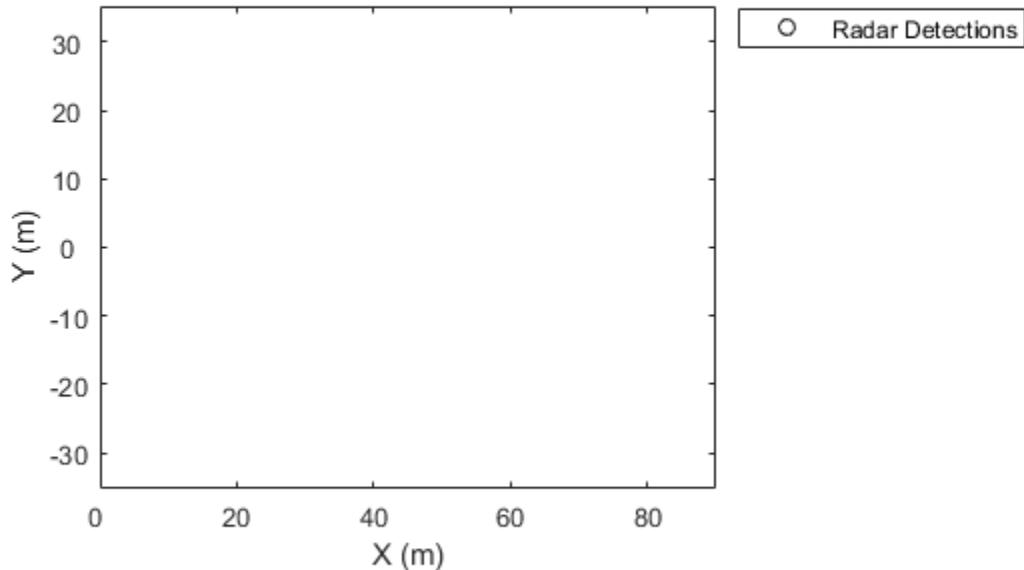
`clearPlotterData(tp)` clears data shown in the plot from all the plotters used in the theater plot, `tp`. Legend entries and coverage areas are not cleared from the plot.

### 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');
```

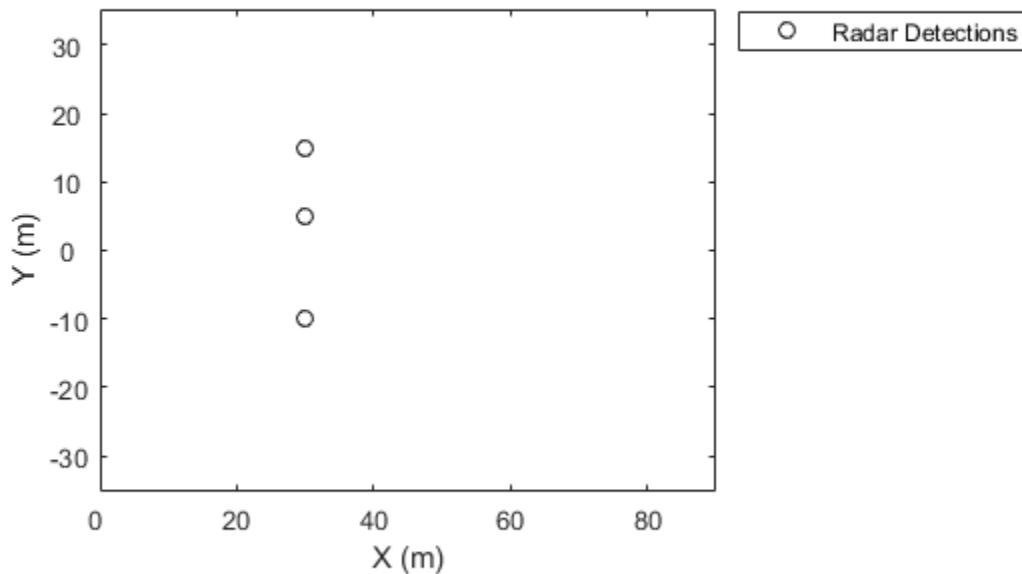


Use `findPlotter` to locate the plotter by its display name.

```
radarPlotter = findPlotter(tp,'DisplayName','Radar Detections');
```

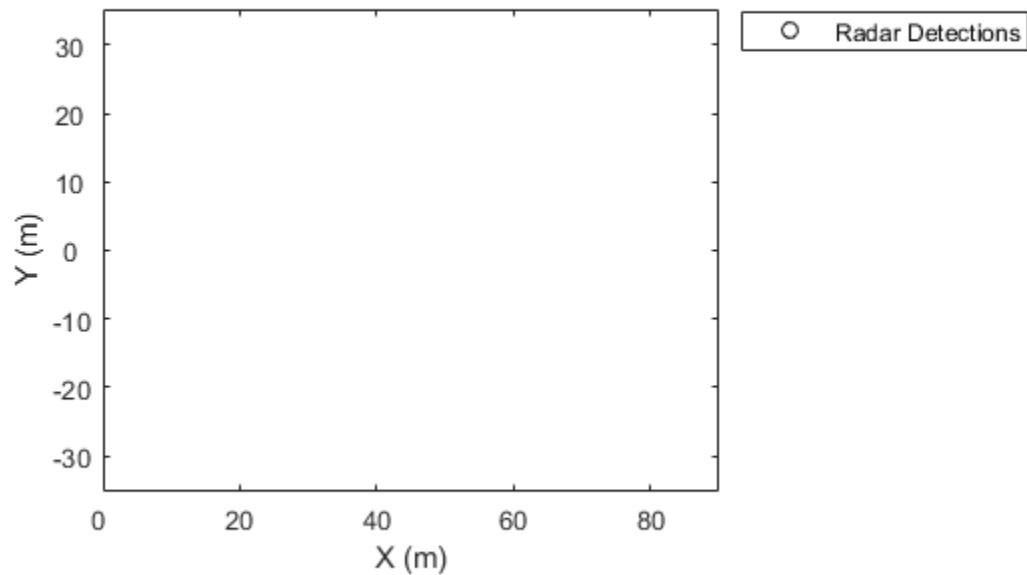
Plot three detections.

```
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

`clearData` | `findPlotter` | `theaterPlot`

**Introduced in R2018b**

# 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`'.

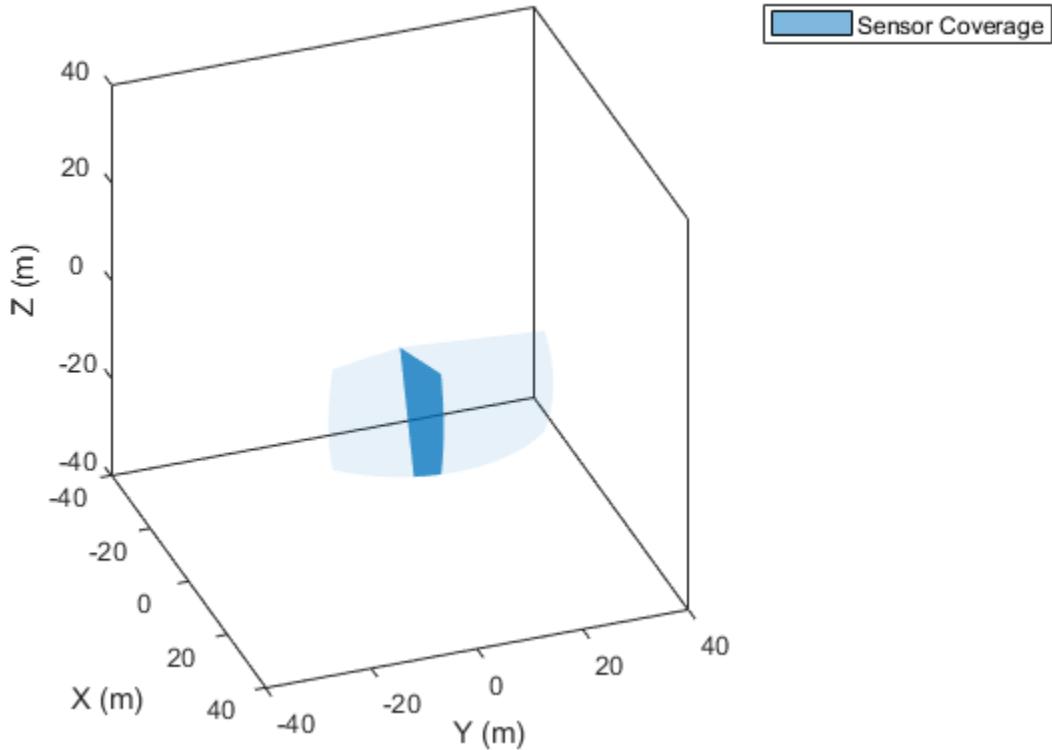
```
tp = theaterPlot('XLim',[-40 40], 'YLim', [-40 40], 'ZLim', [-40 40]);
covp = coveragePlotter(tp,'DisplayName','Sensor Coverage');
```

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 = monostaticRadarSensor(radarIndex, 'No Scanning');
RasterIndex = 2;
raster = monostaticRadarSensor(RasterIndex, 'Raster');
```

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;
```

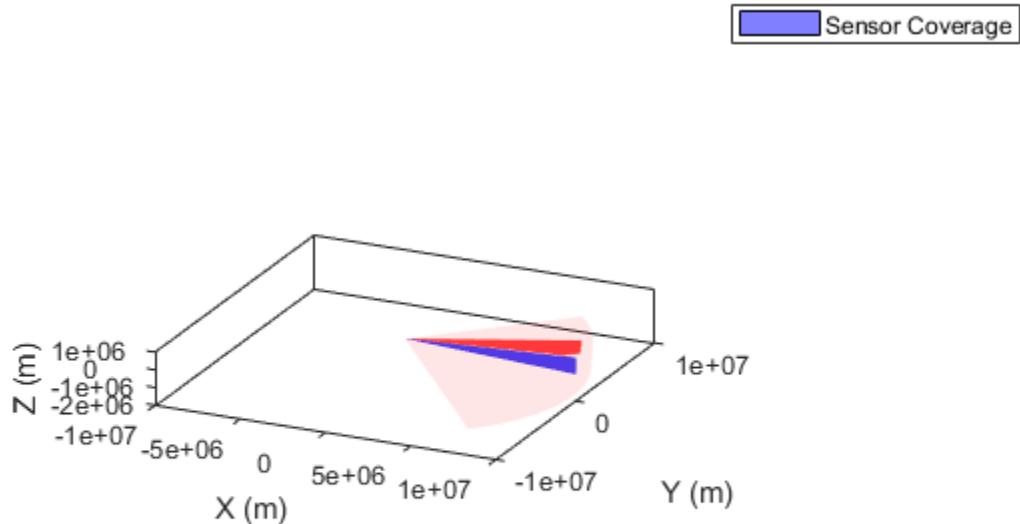
```

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

**tp — Theater plot**  
theaterPlot object

Theater plot, specified as a `theaterPlot` object.

## 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: `'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

`clearData` | `clearPlotterData` | `plotCoverage` | `theaterPlot`

## Introduced in R2020a

# 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'.

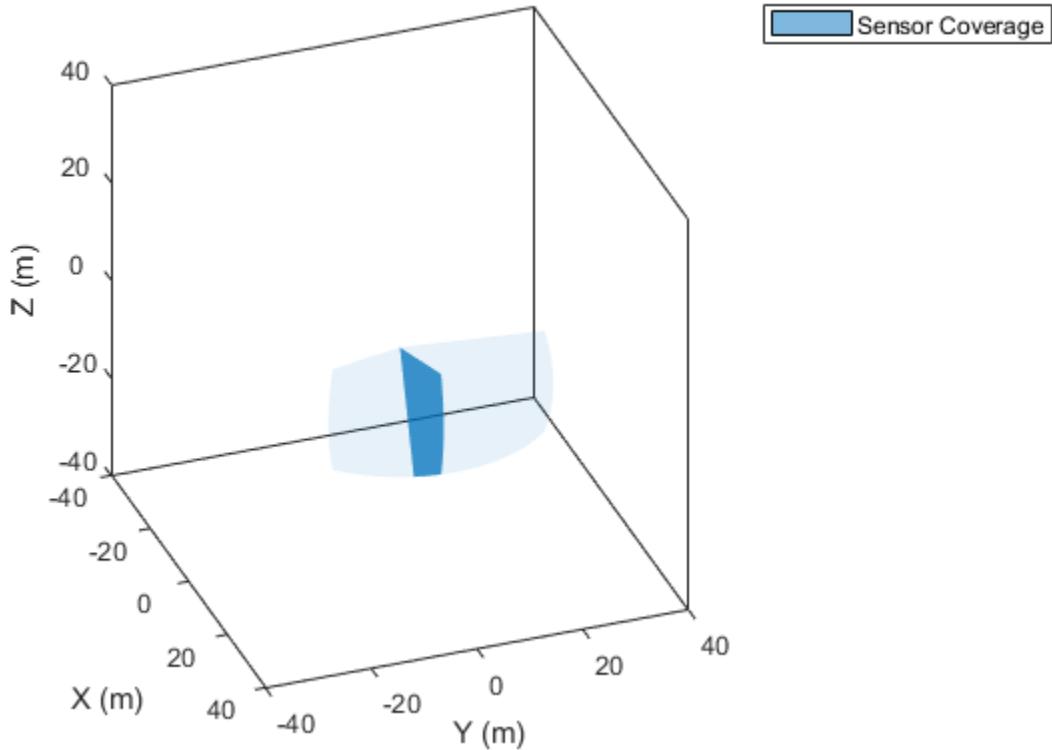
```
tp = theaterPlot('XLim',[-40 40], 'YLim', [-40 40], 'ZLim', [-40 40]);
covp = coveragePlotter(tp,'DisplayName','Sensor Coverage');
```

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 = monostaticRadarSensor(radarIndex, 'No Scanning');
RasterIndex = 2;
raster = monostaticRadarSensor(RasterIndex, 'Raster');
```

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;
```

```

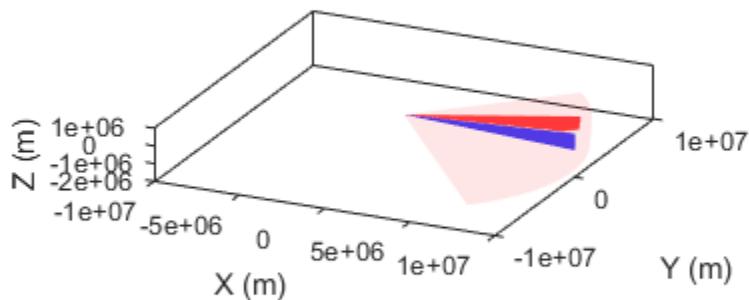
stopTime = 90;
while time < stopTime
    time = time+timestep;
    radar(tgt,time);
    raster(tgt,time);

    % 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

**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. In practice, you can use <code>SensorIndex</code> or <code>EmitterIndex</code> of the sensor or emitter objects, respectively.
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 [<code>azimuth</code>; <code>elevation</code>] 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 [ <code>azimuth</code> ; <code>elevation</code> ] in degrees.
ScanLimits	The minimum and maximum angles the sensor or emitter can scan from its <code>Orientation</code> . <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 [<code>minAz</code>, <code>maxAz</code>] 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 [<code>minAz</code>, <code>maxAz</code>; <code>minEl</code>, <code>maxEl</code>] 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 [ <code>X</code> , <code>Y</code> , <code>Z</code> ] 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**

[clearData](#) | [clearPlotterData](#) | [coveragePlotter](#) | [theaterPlot](#)

**Introduced in R2020a**

# 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.

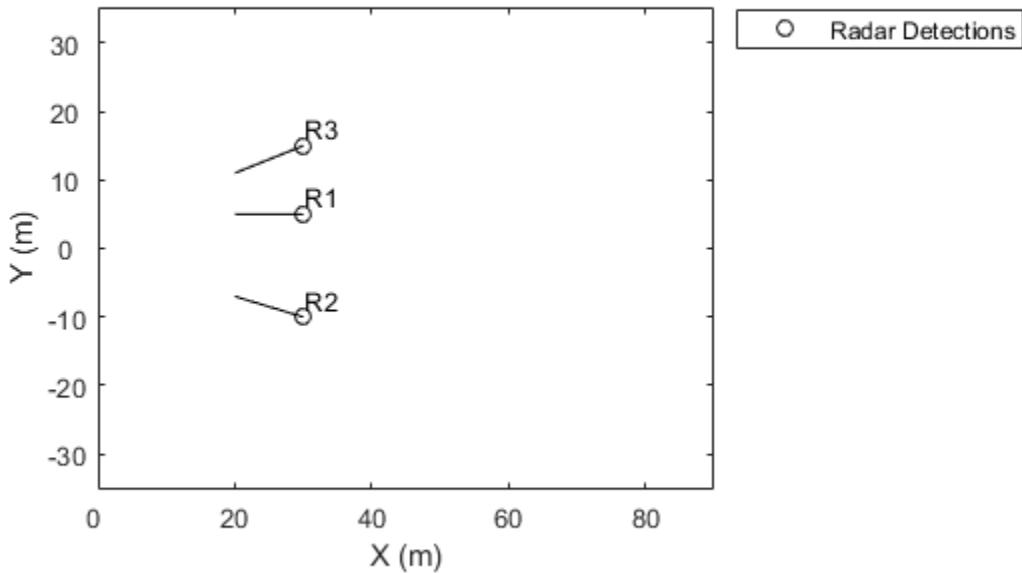
```
tp = theaterPlot('XLim',[0,90], 'YLim',[-35,35], 'ZLim',[1,10]);
```

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 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: `'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.

Value	Description
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'. '	Point
'x'	Cross
'—'	Horizontal line
' '	Vertical line
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
Right-pointing triangle	
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No markers

#### **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**`clearData` | `clearPlotterData` | `plotDetection` | `theaterPlot`**Introduced in R2018b**

## 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.

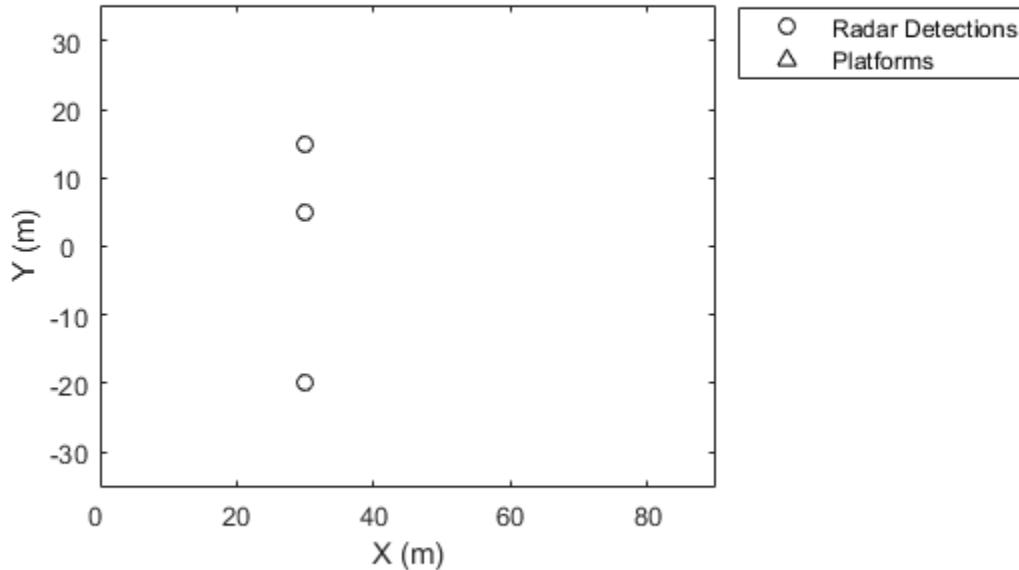
```
radarPlotter = findPlotter(tp, 'Tag', 'radPlot')

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 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: `'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**[clearData](#) | [clearPlotterData](#) | [theaterPlot](#)**Introduced in R2018b**

# 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 Orientation of Oscillating Device

This example shows how to animate the orientation of an oscillating device.

Load `rpy_9axis.mat`. The data in `rpy_9axis.mat` is recorded accelerometer, gyroscope, and magnetometer sensor data from a device oscillating in pitch (around `y`-axis), then yaw (around `z`-axis), then roll (around `x`-axis). The device's `x`-axis was pointing southward when recorded.

```
ld = load('rpy_9axis.mat')
ld = struct with fields:
    Fs: 200
    sensorData: [1x1 struct]
```

Set the sampling frequency. Extract the accelerometer and gyroscope data. Set the decimation factor to 2. Use `fuse` to create an indirect Kalman sensor fusion filter from the data.

```
accel = ld.sensorData.Acceleration;
gyro = ld.sensorData.AngularVelocity;
Fs = ld.Fs;
decim = 2;
fuse = imufilter('SampleRate',Fs,'DecimationFactor',decim);
```

Obtain the pose information of the fused data.

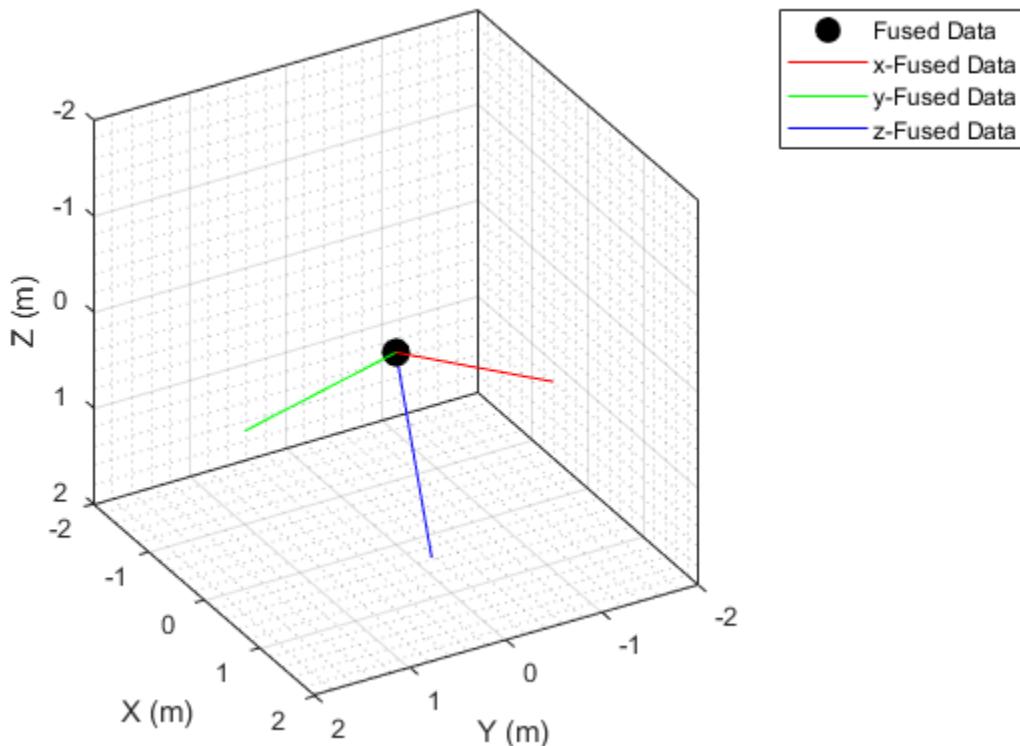
```
pose = fuse(accel,gyro);
```

Create a theater plot. Add to the theater plot an orientation plotter with `'DisplayName'` set to `'Fused Data'` and `'LocalAxesLength'` set to 2.

```
tp = theaterPlot('XLimit',[-2 2], 'YLimit', [-2 2], 'ZLimit', [-2 2]);
op = orientationPlotter(tp, 'DisplayName', 'Fused Data',...
    'LocalAxesLength', 2);
```

Loop through the pose information to animate the changing orientation.

```
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 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: '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.

Value	Description
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'_'	Horizontal line
' '	Vertical line
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
Right-pointing triangle	
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No markers

**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 Nth plotter associated with the theaterPlot.

Tags provide a way to identify plotter objects, for example when searching using `findPlotter`.

**See Also**

`clearData` | `clearPlotterData` | `plotOrientation` | `theaterPlot`

**Introduced in R2018b**

# 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.

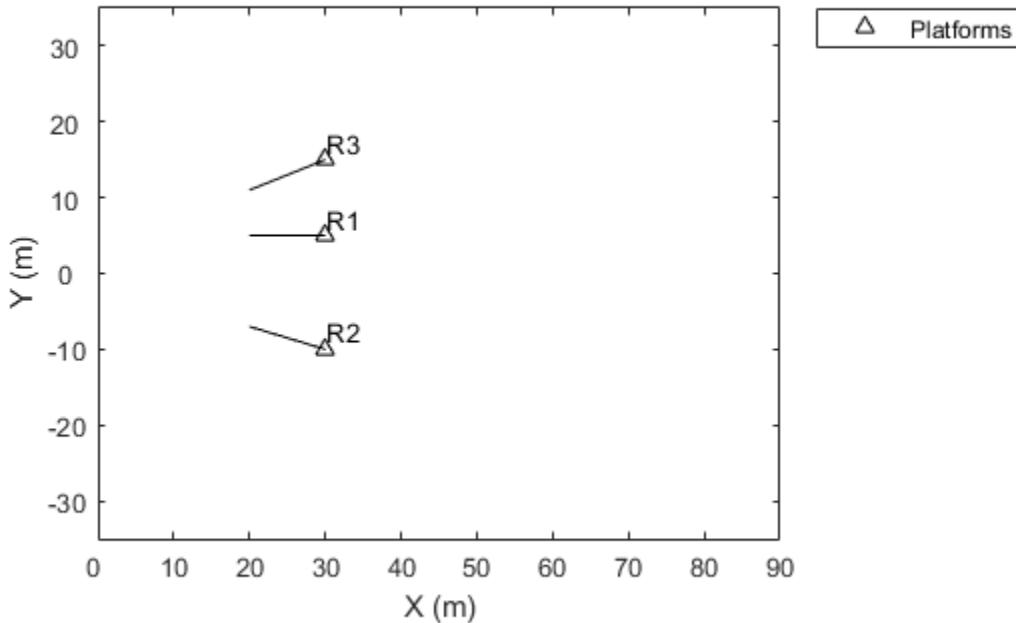
```
tp = theaterPlot('XLim',[0,90], 'YLim',[-35,35], 'ZLim',[1,10]);
```

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 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: `'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.

Value	Description
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'. '	Point
'x'	Cross
'_'	Horizontal line
' '	Vertical line
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
Right-pointing triangle	
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No markers

#### **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**

`clearData` | `clearPlotterData` | `plotPlatform` | `theaterPlot`

**Introduced in R2018b**

# 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.

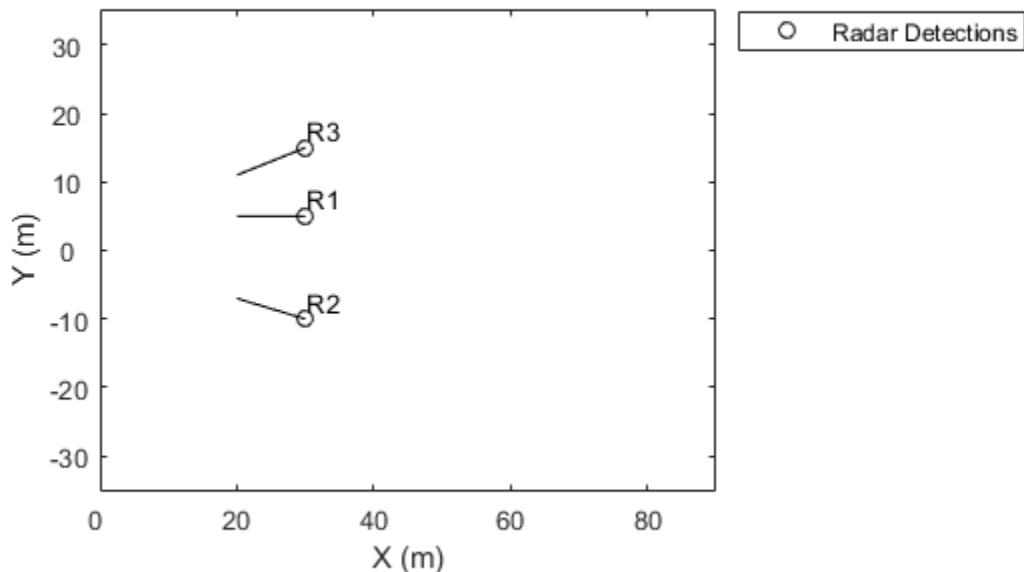
```
tp = theaterPlot('XLim',[0,90], 'YLim',[-35,35], 'ZLim',[1,10]);
```

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**

`clearData` | `clearPlotterData` | `detectionPlotter` | `theaterPlot`

**Introduced in R2018b**

# 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 Orientation of Oscillating Device

This example shows how to animate the orientation of an oscillating device.

Load `rpy_9axis.mat`. The data in `rpy_9axis.mat` is recorded accelerometer, gyroscope, and magnetometer sensor data from a device oscillating in pitch (around  $y$ -axis), then yaw (around  $z$ -axis), then roll (around  $x$ -axis). The device's  $x$ -axis was pointing southward when recorded.

```
ld = load('rpy_9axis.mat')
ld = struct with fields:
    Fs: 200
    sensorData: [1x1 struct]
```

Set the sampling frequency. Extract the accelerometer and gyroscope data. Set the decimation factor to 2. Use `fuse` to create an indirect Kalman sensor fusion filter from the data.

```
accel = ld.sensorData.Acceleration;
gyro = ld.sensorData.AngularVelocity;
Fs = ld.Fs;
```

```
decim = 2;
fuse = imufilter('SampleRate',Fs,'DecimationFactor',decim);
```

Obtain the pose information of the fused data.

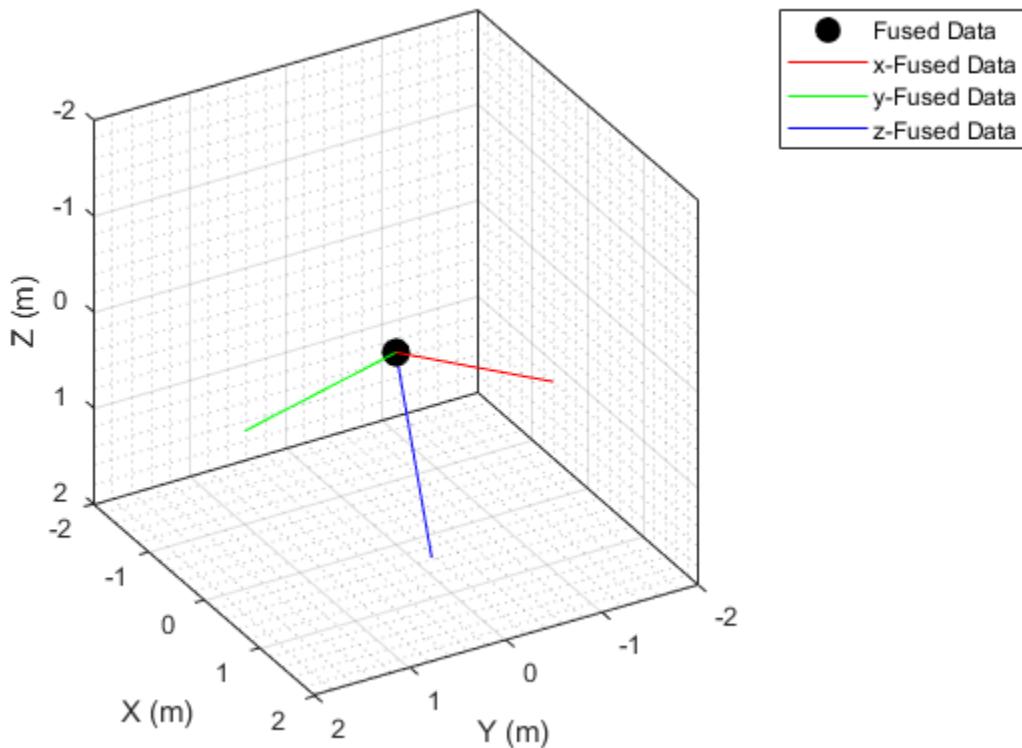
```
pose = fuse(accel,gyro);
```

Create a theater plot. Add to the theater plot an orientation plotter with 'DisplayName' set to 'Fused Data' and 'LocalAxesLength' set to 2.

```
tp = theaterPlot('XLimit',[-2 2], 'YLimit', [-2 2], 'ZLimit', [-2 2]);
op = orientationPlotter(tp, 'DisplayName', 'Fused Data',...
    'LocalAxesLength', 2);
```

Loop through the pose information to animate the changing orientation.

```
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**[clearData](#) | [clearPlotterData](#) | [orientationPlotter](#) | [theaterPlot](#)**Introduced in R2018b**

# 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.

## 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'.

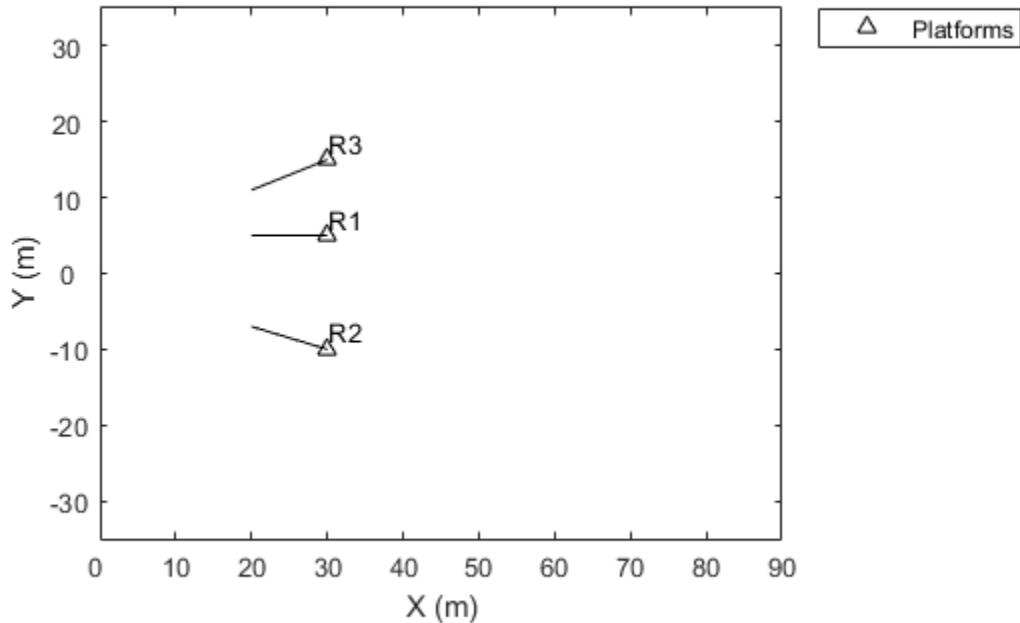
```
plotter = platformPlotter(tp, 'DisplayName', 'Platforms');
```

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

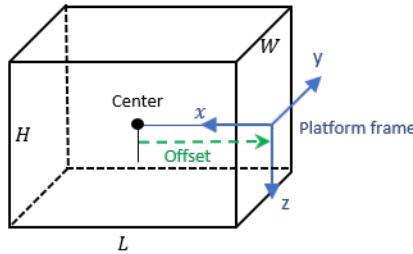
Platform labels, specified as an  $M$ -by-1 cell array of character vectors, where  $M$  is the number of platforms. `labels` contains the text labels corresponding to the  $M$  platforms specified in `positions`. If `labels` is omitted, no labels are plotted.

**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 R2018b**

## 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 2-0 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`'.

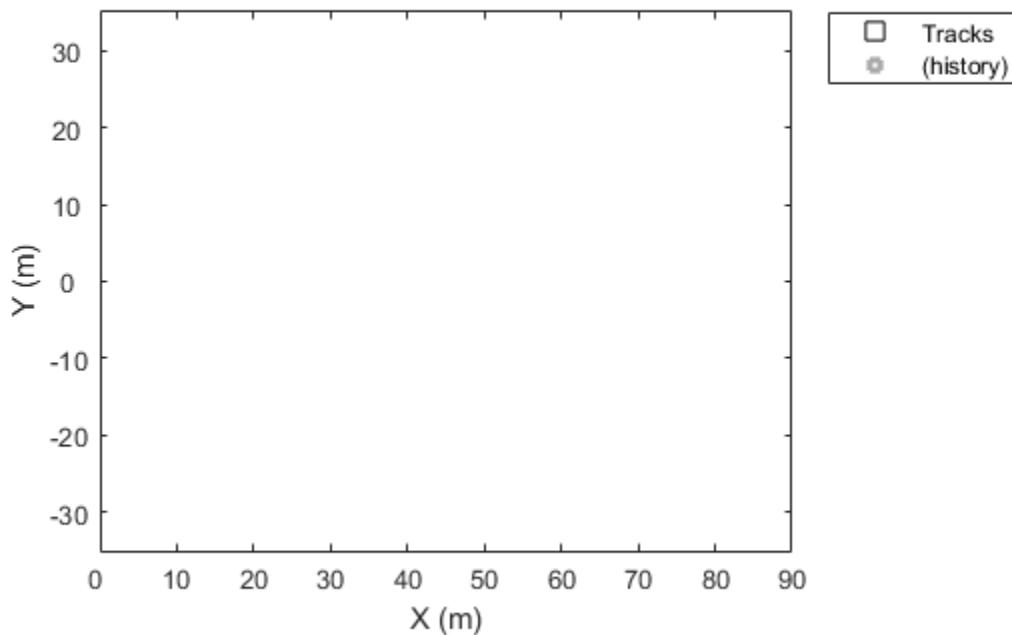
`plotTrack(tPlotter,positions,____,dimensions,orientations)` specifies the dimension and orientation of each tracked object in the plot.

### Examples

## Plot Tracks in Theater Plot

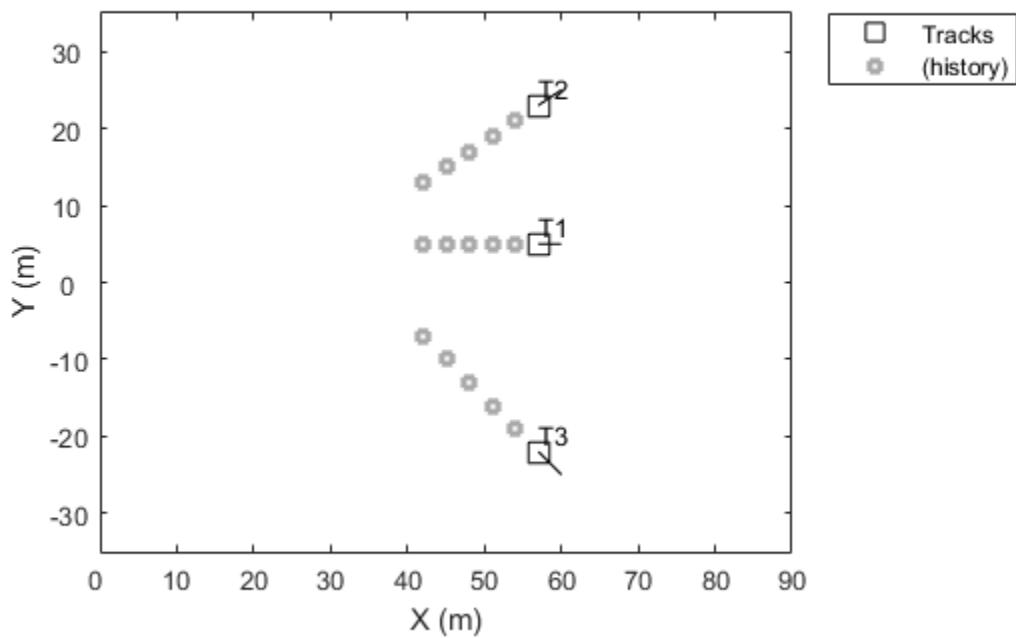
Create a theater plot. Create a track plotter with `DisplayName` set to 'Tracks' and with `HistoryDepth` set to 5.

```
tp = theaterPlot('XLim',[0,90], 'YLim',[-35,35]);  
tPlotter = trackPlotter(tp, 'DisplayName', 'Tracks', 'HistoryDepth', 5);
```

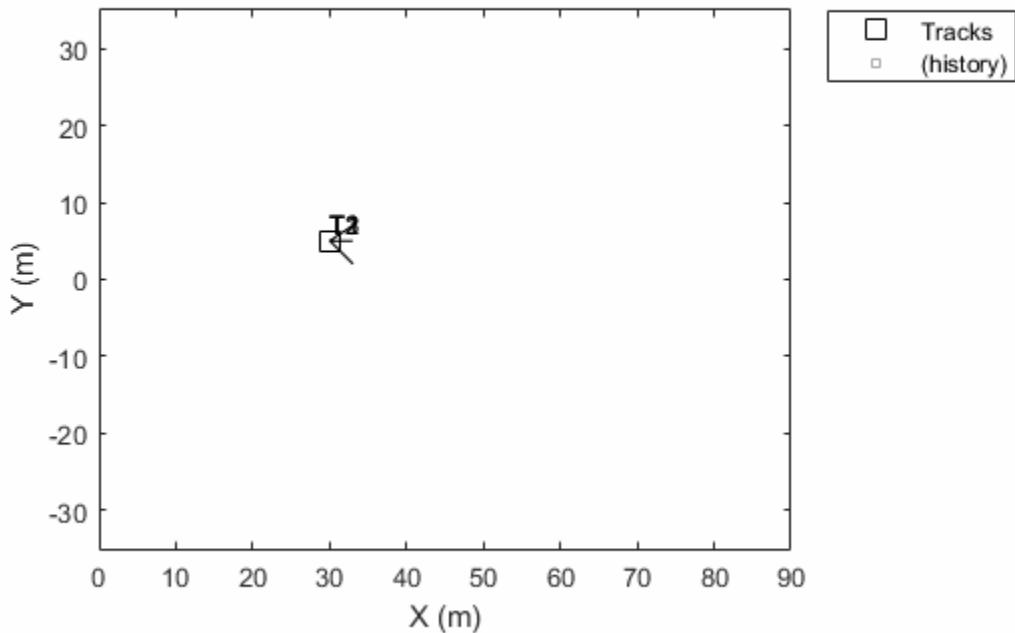


Update the track plotter with three tracks labeled 'T1', 'T2', and 'T3' with start positions in units of meters all starting at (30, 5, 1) with corresponding velocities (in m/s) of (3, 0, 1), (3, 2, 2) and (3, -3, 5), respectively. Update the tracks with the velocities for ten iterations.

```
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'.

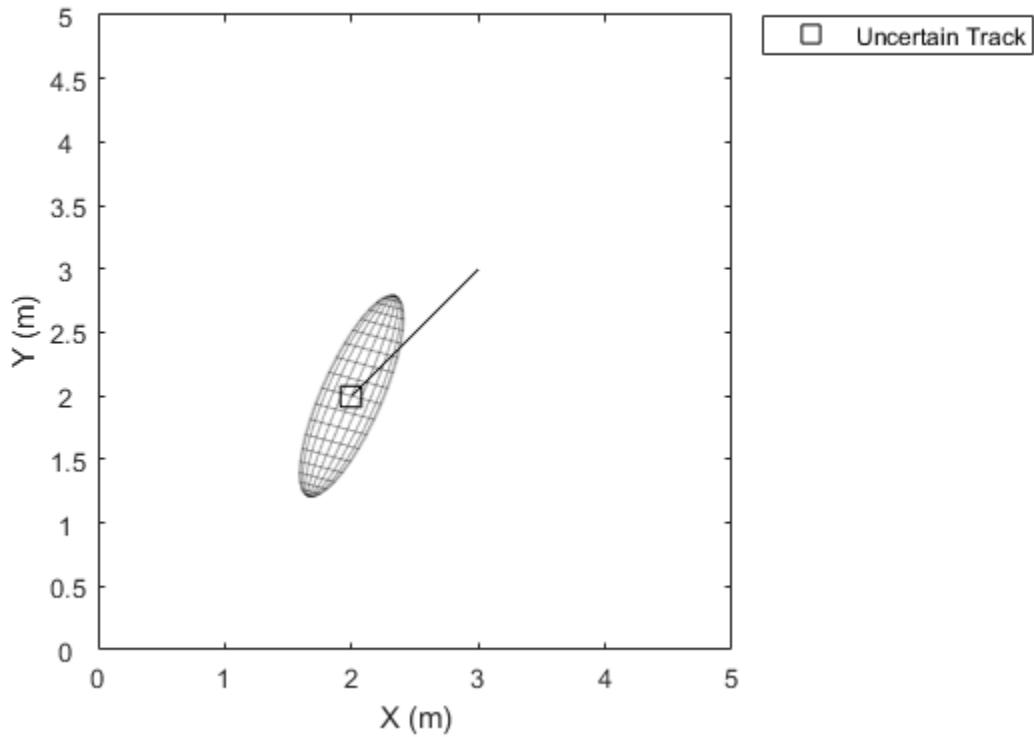
```
tp = theaterPlot('Xlim',[0 5],'Ylim',[0 5]);
tPlotter = trackPlotter(tp,'DisplayName','Uncertain Track');
```

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.

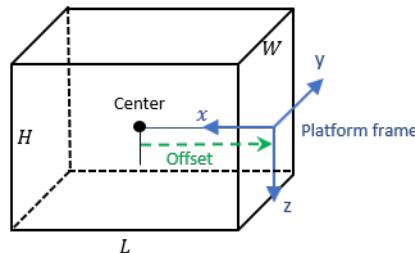
#### **dimensions — Platform dimensions**

$M$ -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**

[clearData](#) | [clearPlotterData](#) | [theaterPlot](#) | [trackPlotter](#)

**Introduced in R2018b**

# 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 a Trajectory

This example shows how to create an animation of a platform moving on a trajectory.

First, create a `trackingScenario` and add waypoints for a trajectory.

```
ts = trackingScenario;
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.

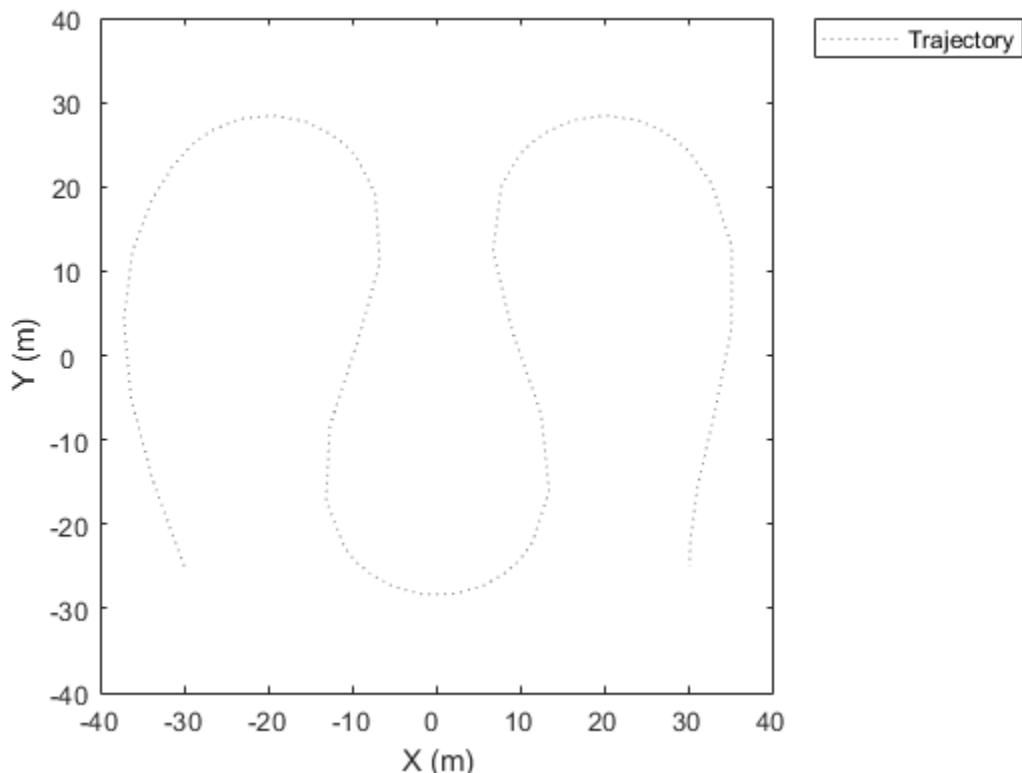
```
target = platform(ts);
traj = waypointTrajectory('Waypoints',wayPoints,'TimeOfArrival',elapsedTime);
target.Trajectory = traj;
```

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})
```

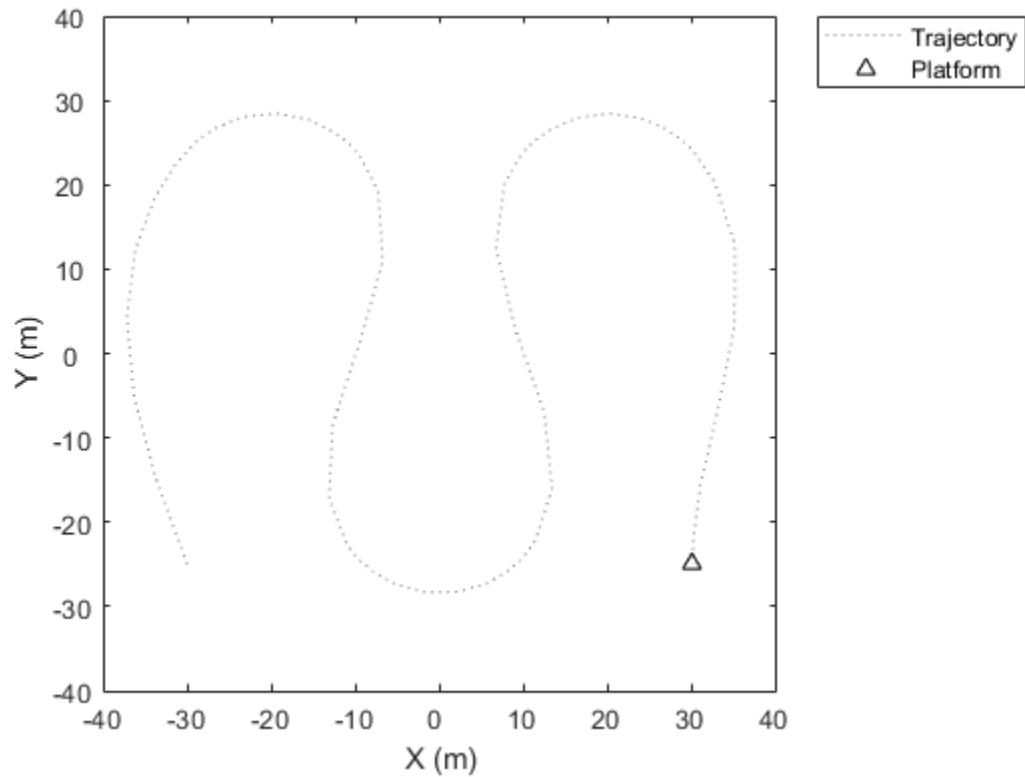


Animate using the `platformPlotter`.

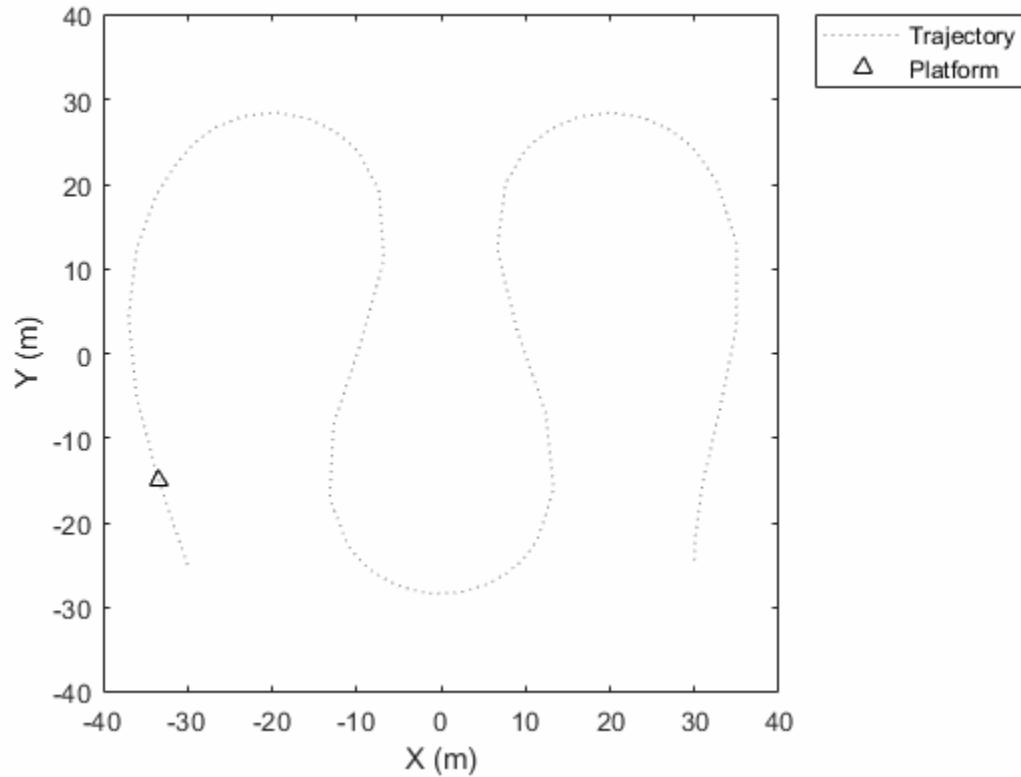
```
restart(ts);
trajPlotter = platformPlotter(tp, 'DisplayName', 'Platform');

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

[clearData](#) | [clearPlotterData](#) | [theaterPlot](#) | [trajectoryPlotter](#)

**Introduced in R2018b**

# 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`.

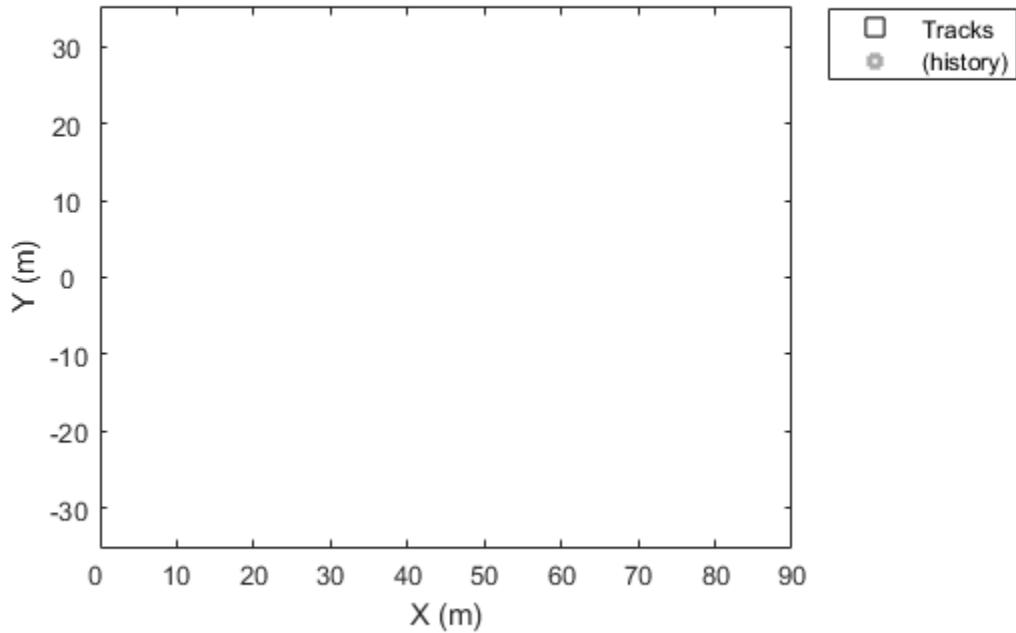
`tPlotter = trackPlotter(tp,Name,Value)` creates a track plotter with additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Plot Tracks in Theater Plot

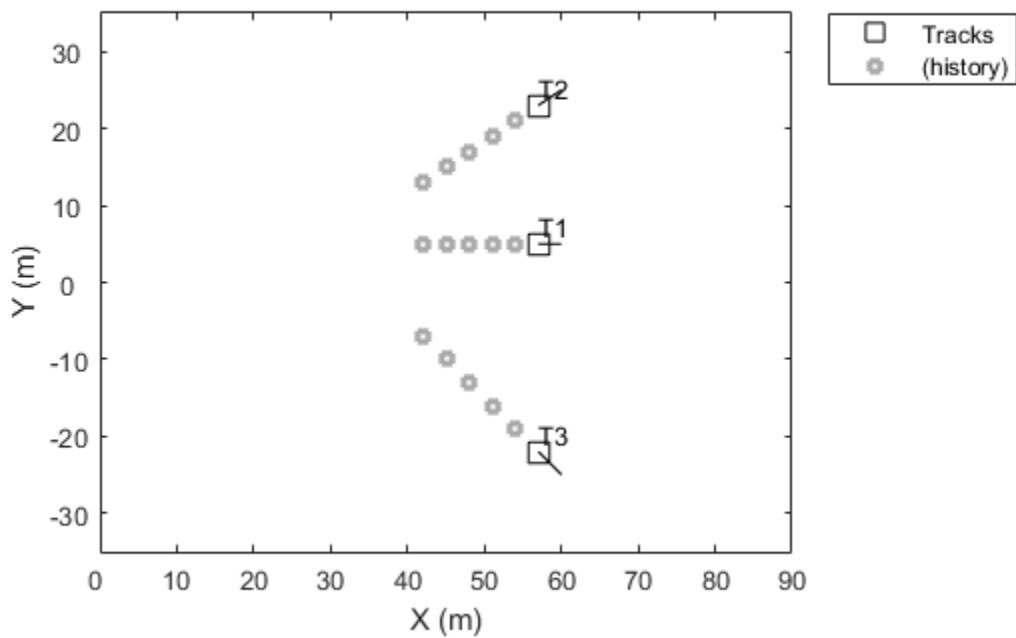
Create a theater plot. Create a track plotter with `DisplayName` set to 'Tracks' and with `HistoryDepth` set to 5.

```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35]);
tPlotter = trackPlotter(tp,'DisplayName','Tracks','HistoryDepth',5);
```

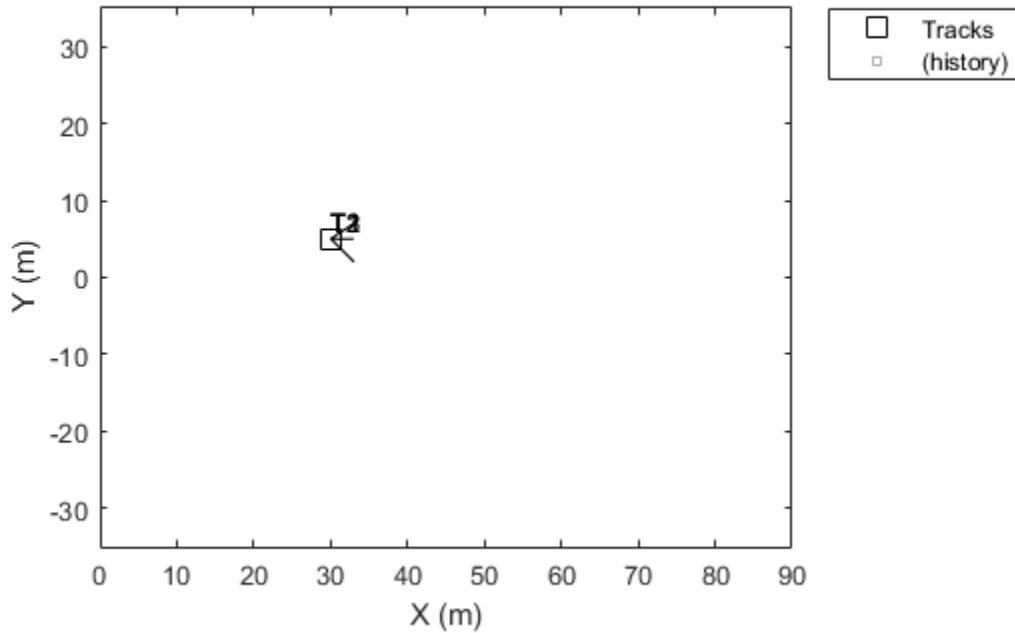


Update the track plotter with three tracks labeled 'T1', 'T2', and 'T3' with start positions in units of meters all starting at (30, 5, 1) with corresponding velocities (in m/s) of (3, 0, 1), (3, 2, 2) and (3, -3, 5), respectively. Update the tracks with the velocities for ten iterations.

```
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 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: '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 2-0 input argument of `plotTrack`, 'ConnectHistory' must be 'on'. If `trackIDs` on page 2-0 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.

Value	Description
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'—'	Horizontal line
' '	Vertical line
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
Right-pointing triangle	
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No markers

**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 '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**

`clearData` | `clearPlotterData` | `plotTrack` | `theaterPlot`

**Introduced in R2018b**

# 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 a Trajectory

This example shows how to create an animation of a platform moving on a trajectory.

First, create a `trackingScenario` and add waypoints for a trajectory.

```
ts = trackingScenario;
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.

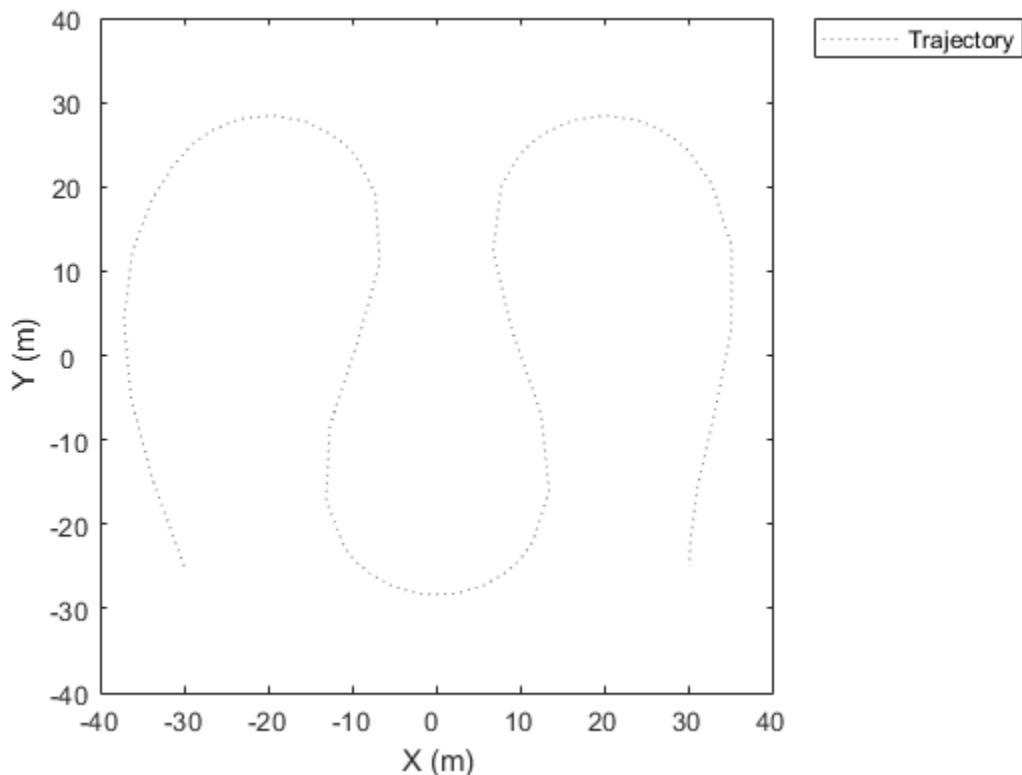
```
target = platform(ts);
traj = waypointTrajectory('Waypoints',wayPoints,'TimeOfArrival',elapsedTime);
target.Trajectory = traj;
```

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})
```

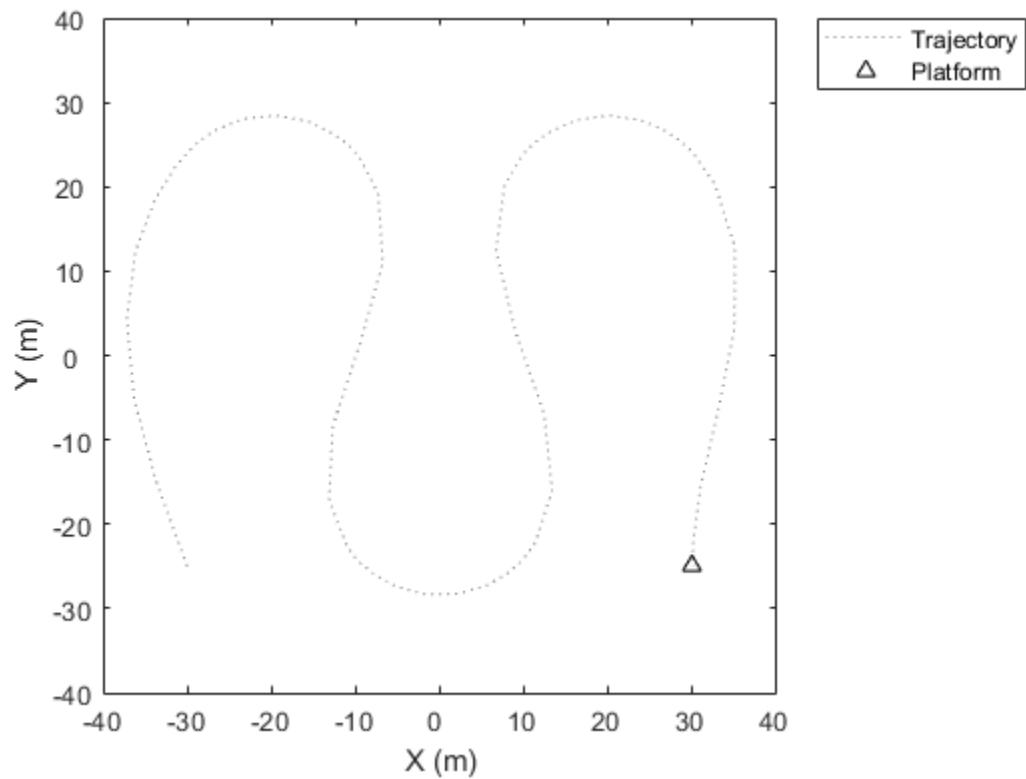


Animate using the `platformPlotter`.

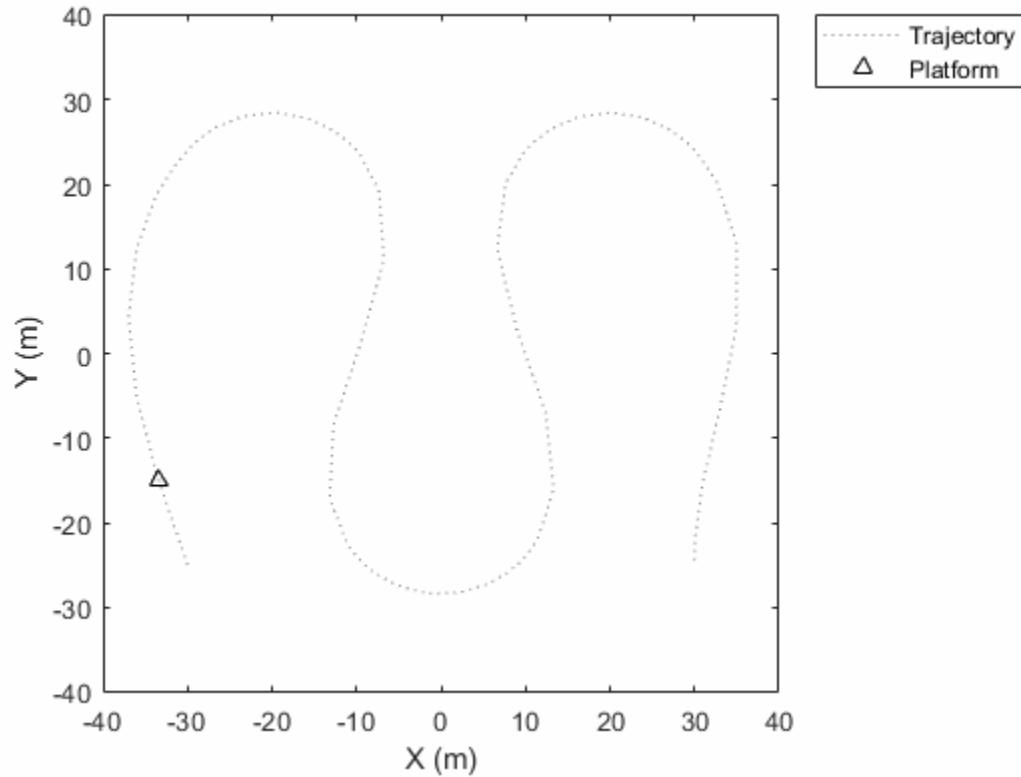
```
restart(ts);
trajPlotter = platformPlotter(tp, 'DisplayName', 'Platform');

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 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: `'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 Nth plotter associated with the theaterPlot.

Tags provide a way to identify plotter objects, for example when searching using `findPlotter`.

## **See Also**

`clearData` | `clearPlotterData` | `plotTrajectory` | `theaterPlot`

## **Introduced in R2018b**

# 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 '`x`' and '`x'` indicate first and second order derivatives, respectively.

Example: [200;0.2;150;0.1;0;0.25]

Data Types: 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]

## **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

[trackerGNN](#) | [trackerTOMHT](#) | [trackingABF](#) | [trackingCKF](#) | [trackingEKF](#) | [trackingGSF](#) | [trackingIMM](#) | [trackingKF](#) | [trackingMSCEKF](#) | [trackingPF](#) | [trackingUKF](#)

### Introduced in R2018b

# trackingCKF

Cubature Kalman filter for object tracking

## Description

The `trackingCKF` object represents a cubature Kalman filter designed for tracking objects that follow a nonlinear motion model or are measured by a nonlinear measurement model. Use the filter to predict the future location of an object, to reduce noise in a measured location, or to help associate multiple object detections with their tracks.

The cubature Kalman filter estimates the uncertainty of the state and the propagation of that uncertainty through the nonlinear state and measurement equations. There are a fixed number of cubature points chosen based on the spherical-radial transformation to guarantee an exact approximation of a Gaussian distribution up to the third moment. As a result, the corresponding filter is the same as an unscented Kalman filter, `trackingUKF`, with `Alpha` = 1, `Beta` = 0, and `Kappa` = 0.

## Creation

### Syntax

```
ckf = trackingCKF
ckf = trackingCKF(transitionFcn,measurementFcn,state)
ckf = trackingCKF(___,Name,Value)
```

### Description

`ckf = trackingCKF` returns a cubature Kalman filter object with default state transition function, measurement function, state, and additive noise model.

`ckf = trackingCKF(transitionFcn,measurementFcn,state)` specifies the `StateTransitionFcn`, `MeasurementFcn`, and `State` properties directly.

`ckf = trackingCKF(___,Name,Value)` specifies the properties of the Kalman filter using one or more `Name,Value` pair arguments. Any unspecified properties take default values.

## Properties

### State — Kalman filter state

real-valued  $M$ -element vector

Kalman filter state, specified as a real-valued  $M$ -element vector.

Example: [200;0.2;150;0.1;0;0.25]

Data Types: 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: `eye(6)`

#### **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.

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 ( <code>HasAdditiveProcessNoise = true</code> )	Valid Syntaxes ( <code>HasAdditiveProcessNoise = false</code> )
<pre>x(k) = statetransitionfcn(x(k-1)) x(k) = statetransitionfcn(x(k-1),parameters)</pre> <ul style="list-style-type: none"> <li>• <math>x(k)</math> is the state at time <math>k</math>.</li> <li>• <code>parameters</code> stands for all additional arguments required by the state transition function.</li> </ul>	<pre>x(k) = statetransitionfcn(x(k-1),w(k-1)) x(k) = statetransitionfcn(x(k-1),w(k-1),dt) x(k) = statetransitionfcn(__,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 value for the process noise at time <math>k</math>.</li> <li>• <code>dt</code> is the time step of the <code>trackingCKF</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 state transition function.</li> </ul>

Example: `@constacc`

Data Types: `function_handle`

#### **ProcessNoise — Process noise covariance**

1 (default) | positive real-valued scalar | positive-definite real-valued matrix

Process noise covariance:

- When `HasAdditiveProcessNoise` is `true`, specify the process noise covariance as a 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.

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; 0.05 1.0 2.0; 0 2.0 1.0]

### **Dependencies**

This parameter depends on the `HasAdditiveNoise` property.

**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. This function can be a nonlinear function that models measurements from the predicted state. Input to the function is the  $M$ -element state vector. The output is the  $N$ -element measurement vector. The function can take additional input arguments, such as sensor position and orientation.

- If `HasAdditiveMeasurementNoise` is `true`, specify the function using one of these syntaxes:

```
z(k) = measurementfcn(x(k))
z(k) = measurementfcn(x(k),parameters)
```

where  $x(k)$  is the state at time  $k$ , and  $z(k)$  is the predicted measurement at time  $k$ . The `parameters` term 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)
```

where  $x(k)$  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.

Example: `@cameas`

Data Types: `function_handle`

**MeasurementNoise — Measurement noise covariance**

`1` (default) | positive scalar | positive-definite real-valued matrix

Measurement noise covariance::

- 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.

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.

## 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

## Examples

### Run trackingCKF Filter

This example shows how to create and run a `trackingCKF` filter. Call the `predict` and `correct` functions to track an object and correct the state estimation based on measurements.

Create the filter. Specify the constant velocity motion model, the measurement model, and the initial state.

```
state = [0;0;0;0;0;0];
ckf = trackingCKF(@constvel,@cvmeas,state);
```

Call `predict` to get the predicted state and covariance of the filter. Use a 0.5 second time step.

```
[xPred,pPred] = predict(ckf,0.5);
```

Call `correct` with a given measurement.

```
meas = [1;1;0];
[xCorr,pCorr] = correct(ckf,meas);
```

Continue to predict the filter state. Specify the desired time step in seconds if necessary.

```
[xPred,pPred] = predict(ckf);           % Predict over 1 second
[xPred,pPred] = predict(ckf,2);          % Predict over 2 seconds
```

## References

- [1] Arasaratnam, Ienkaran, and Simon Haykin. "Cubature kalman filters." IEEE Transactions on automatic control 54, no. 6 (2009): 1254-1269.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

#### Functions

`clone` | `constvel` | `correct` | `cvmeas` | `distance` | `likelihood` | `predict` | `residual`

#### Objects

`trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

**Introduced in R2018b**

# trackingGSF

Gaussian-sum filter for object tracking

## Description

The `trackingGSF` object represents a Gaussian-sum filter designed for object tracking. You can define the state probability density function by a set of finite Gaussian-sum components. Use this filter for tracking objects that require a multi-model description due to incomplete observability of state through measurements. For example, this filter can be used as a range-parameterized extended Kalman filter when the detection contains only angle measurements.

## Creation

### Syntax

```
gsf = trackingGSF  
gsf = trackingGSF(trackingFilters)  
gsf = trackingGSF(trackingFilters,modelProbabilities)  
gsf = trackingGSF(____,'MeasurementNoise',measNoise)
```

### Description

`gsf = trackingGSF` returns a Gaussian-sum filter with two constant velocity extended Kalman filters (`trackingEKF`) with equal initial weight.

`gsf = trackingGSF(trackingFilters)` specifies the Gaussian components of the filter in `trackingFilters`. The initial weights of the filters are assumed to be equal.

`gsf = trackingGSF(trackingFilters,modelProbabilities)` specifies the initial weight of the Gaussian components in `modelProbabilities` and sets the `ModelProbabilities` property.

`gsf = trackingGSF(____,'MeasurementNoise',measNoise)` specifies the measurement noise of the filter. The `MeasurementNoise` property is set for each Gaussian component.

## Properties

### State — Weighted estimate of filter state

real-valued  $M$ -element vector

This property is read-only.

Weighted estimate of filter state, specified as a real-valued  $M$ -element vector. This state is estimated based on the weighted combination of filters in `TrackingFilters`. Use `ModelProbabilities` to change the weights.

Example: [200;0.2]

Data Types: `single` | `double`

**StateCovariance — State estimation error covariance**positive-definite real-valued  $M$ -by- $M$  matrix

This property is read-only.

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. This state covariance is estimated based on the weighted combination of filters in **TrackingFilters**. Use **ModelProbabilities** to change the weights.

Example: [20 0.1; 0.1 1]

Data Types: single | double

**TrackingFilters — List of filters**

{trackingEKF,trackingEKF} (default) | cell array of tracking filters

List of filters, specified as a cell array of tracking filters. Specify these filters when creating the object. By default, the filters have equal probability. Specify **modelProbabilities** if the filters have different probabilities.

---

**Note** The state of each filter must be the same size and have the same physical meaning.

---

Data Types: cell

**ModelProbabilities — Weight of each filter**

[0.5 0.5] (default) | vector of probabilities between 0 and 1

Weight of each filter, specified as a vector of probabilities from 0 to 1. By default, the weight of each component of the filter is equal.

Data Types: single | double

**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. The matrix is a square with side lengths equal to the number of measurements. A scalar input is extended to a square diagonal matrix.

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, where  $R$  is the number of measurements.

Example: 0.2

Data Types: single | double

**Object Functions**

<b>predict</b>	Predict state and state estimation error covariance of tracking filter
<b>correct</b>	Correct state and state estimation error covariance using tracking filter
<b>correctjpda</b>	Correct state and state estimation error covariance using tracking filter and JPDA
<b>distance</b>	Distances between current and predicted measurements of tracking filter
<b>likelihood</b>	Likelihood of measurement from tracking filter

clone Create duplicate tracking filter

## Examples

### Run trackingGSF Filter

This example shows how to create and run a `trackingGSF` filter. Specify three extended Kalman filters (EKFs) as the components of the Gaussian-sum filter. Call the `predict` and `correct` functions to track an object and correct the state estimate based on measurements.

Create three EKFs each with a state distributed around  $[0;0;0;0;0;0]$  and running on position measurements. Specify them as the input to the `trackingGSF` filter.

```
filters = cell(3,1);
filter{1} = trackingEKF(@constvel,@cvmeas,rand(6,1), 'MeasurementNoise',eye(3));
filter{2} = trackingEKF(@constvel,@cvmeas,rand(6,1), 'MeasurementNoise',eye(3));
filter{3} = trackingEKF(@constvel,@cvmeas,rand(6,1), 'MeasurementNoise',eye(3));
gsf = trackingGSF(filter);
```

Call `predict` to get the predicted state and covariance of the filter. Use a 0.1 sec time step.

```
[x_pred, P_pred] = predict(gsf,0.1);
```

Call `correct` with a given measurement.

```
meas = [0.5;0.2;0.3];
[xCorr,pCorr] = correct(gsf,meas);
```

Compute the distance between the filter and a different measurement.

```
d = distance(gsf,[0;0;0]);
```

## References

- [1] Alspach, Daniel, and Harold Sorenson. "Nonlinear Bayesian estimation using Gaussian sum approximations." *IEEE Transactions on Automatic Control*. Vol. 17, No. 4, 1972, pp. 439-448.
- [2] Ristic, B., Arulampalam, S. and McCarthy, J., 2002. *Target motion analysis using range-only measurements: algorithms, performance and application to ISAR data*. Signal Processing, 82(2), pp.273-296.
- [3] Peach, N. "Bearings-only tracking using a set of range-parameterised extended Kalman filters." IEE Proceedings-Control Theory and Applications 142, no. 1 (1995): 73-80.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

`trackingCKF` | `trackingEKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

**Introduced in R2018b**

# trackingIMM

Interacting multiple model (IMM) filter for object tracking

## Description

The `trackingIMM` object represents an interacting multiple model (IMM) filter designed for tracking objects that are highly maneuverable. Use the filter to predict the future location of an object, to reduce noise in the detected location, or help associate multiple object detections with their tracks.

The IMM filter deals with the multiple motion models in the Bayesian framework. This method resolves the target motion uncertainty by using multiple models at a time for a maneuvering target. The IMM algorithm processes all the models simultaneously and switches between models according to their updated weights.

## Creation

### Syntax

```
imm = trackingIMM
imm = trackingIMM(trackingFilters)
imm = trackingIMM(trackingFilters,modelConversionFcn)
imm = trackingIMM(trackingFilters,modelConversionFcn,transitionProbabilities)
imm = trackingIMM(___,Name,Value)
```

### Description

`imm = trackingIMM` returns an IMM filter object with default tracking filters `{trackingEKF,trackingEKF,trackingEKF}` with the motion models set as constant velocity, constant acceleration, and constant turn, respectively. The filter uses the default conversion function, `@switchimm`.

`imm = trackingIMM(trackingFilters)` specifies the `TrackingFilters` property and sets all other properties to default values.

`imm = trackingIMM(trackingFilters,modelConversionFcn)` also specifies the `ModelConversionFcn` property.

`imm = trackingIMM(trackingFilters,modelConversionFcn,transitionProbabilities)` also specifies the `TransitionProbabilities` property.

`imm = trackingIMM(___,Name,Value)` specifies the properties of the filter using one or more `Name,Value` pair arguments. Any unspecified properties take default values. Specify any other input arguments from previous syntaxes first.

## Properties

### State — Filter state

`[0;0;0;0;0;0]` (default) | real-valued  $M$ -element vector

Filter state, specified as a real-valued  $M$ -element vector. Specify the initial state when creating the object using name-value pairs.

Data Types: single | double

**StateCovariance — State estimation error covariance**  
 $\text{diag}([1 \ 100 \ 1 \ 100 \ 1 \ 100])$  (default) |  $M$ -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. Specify the initial state covariance when creating the object using name-value pairs.

Example: `eye(6)`

Data Types: single | double

**TrackingFilters — List of filters**  
 $\{\text{trackingEKF}, \text{trackingEKF}, \text{trackingEKF}\}$  (default) | cell array of tracking filters

List of filters, specified as a cell array of tracking filters. By default, the filters have equal probability. Specify **ModelProbabilities** if the filters have different probabilities.

Data Types: cell

**ModelConversionFcn — Function to convert state or state covariance**  
 $\text{@switchimm}$  (default) | function handle

Function to convert the state or state covariance, specified as a function handle. The function converts the state or state covariance from one model type to another. The function signature is:

```
function x2 = modelConversionFcn(modelType1,x1,modelType2)
```

The `modelType1` and `modelType2` inputs are the names of the two model names. `x1` specifies the **State** or **StateCovariance** of the first model. `x2` outputs the **State** or **StateCovariance**

Data Types: function\_handle

**TransitionProbabilities — Probability of filter model transitions**  
 $0.9$  (default) | positive real scalar |  $L$ -element vector |  $L$ -by- $L$  matrix

Probability of filter model transitions, specified as a positive real scalar,  $L$ -element vector, or  $L$ -by- $L$  matrix, where  $L$  is the number of filters:

- When specified as a scalar, the probability is uniform for staying on each filter. The remaining probability ( $1-p$ ) is distributed evenly across the other motion models.
- When specified as a vector, each element defines the probability of staying on each filter. The remaining probability ( $1-p$ ) is distributed evenly across the other motion models evenly.
- When specified as a matrix, the  $(j,k)$  element defines the probability of transitioning from the  $j$ th filter to the  $k$ th filter. All elements must lie on the interval  $[0,1]$ , and each row must sum to 1.

The transition probability defined for each model corresponds to the probability that the filter switches from this model to another model in one second.

Example: `0.75`

Data Types: single | double

**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 specified as a scalar, the matrix is a multiple of the  $N$ -by- $N$  identity matrix.  $N$  is the size of the measurement vector.

Specify `MeasurementNoise` before any call to the `correct` function.

Example: `0.2`

**ModelProbabilities — Weight of each filter**

1/L\*ones(L) (default) | vector of probabilities between 0 and 1

Weight of each filter, specified as a vector of probabilities from 0 to 1. By default, the weight of each component of the filter is equal.  $L$  is the number of filters. The IMM filter updates the weight of each filter in the prediction step.

Data Types: `single` | `double`

## 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>initialize</code>	Initialize state and covariance of tracking filter

## Examples

### Run trackingIMM Filter

This example shows how to create and run an interacting multiple model (IMM) filter using a `trackingIMM` object. Call the `predict` and `correct` functions to track an object and correct the state estimate based on measurements.

Create the filter. Use name-value pairs to specify additional properties of the object.

```
detection = objectDetection(0, [1;1;0], 'MeasurementNoise', [1 0.2 0; 0.2 2 0; 0 0 1]);
filter = {initctekf(detection);initcvekf(detection)};
modelConv = @switchimm;
transProb = [0.9,0.9];
imm = trackingIMM('State',[1;1;3;1;5;1;1], 'StateCovariance',eye(7),...
    'TransitionProbabilities',transProb, 'TrackingFilters',filter,... 
    'ModelConversionFcn',modelConv);
```

Call `predict` to get the predicted state and covariance of the filter. Use a 0.5 sec time step.

```
[xPred,pPred] = predict(imm,0.5);
```

Call `correct` with a given measurement.

```
meas = [1;1;0];
[xCorr,pCorr] = correct(imm,meas);
```

Continue to predict the filter state. Specify the desired time step in seconds if necessary.

```
[xPred,pPred] = predict(imm);           % Predict over 1 second  
[xPred,pPred] = predict(imm,2);         % Predict over 2 seconds
```

## References

- [1] Bar-Shalom, Yaakov, Peter K. Willett, and Xin Tian. *Tracking and data fusion*. Storrs, CT, USA:: YBS publishing, 2011.
- [2] Blackman, Samuel, and Robert Popoli. "Design and analysis of modern tracking systems." Norwood, MA: Artech House, 1999.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[constacc](#) | [constturn](#) | [constvel](#) | [trackingCKF](#) | [trackingEKF](#) | [trackingGSF](#) | [trackingKF](#) | [trackingUKF](#)

**Introduced in R2018b**

# trackingMSCEKF

Extended Kalman filter for object tracking in modified spherical coordinates (MSC)

## Description

The `trackingMSCEKF` object represents an extended Kalman filter (EKF) for object tracking in modified spherical coordinates (MSC) using angle-only measurements from a single observer. Use the filter to predict the future location of an object in the MSC frame or associate multiple object detections with their tracks. You can specify the observer maneuver or acceleration required by the state-transition functions (`@constantvelmsc` and `@constantvelmscjac`) by using the `ObserverInput` property.

The following properties are fixed for the `trackingMSCEKF` object:

- `StateTransitionFcn` - `@constvelmsc`
- `StateTransitionJacobianFcn` - `@constvelmscjac`
- `MeasurementFcn` - `@cvmeasmsc`
- `Measurement JacobianFcn` - `@cvmeasmscjac`
- `HasAdditiveProcessNoise` - `false`
- `HasAdditiveMeasurementNoise` - `true`

## Creation

### Syntax

```
mscef = trackingMSCEKF  
mscef = trackingMSCEKF(Name,Value)
```

#### Description

`mscef = trackingMSCEKF` returns an extended Kalman filter to use the MSC state-transition and measurement functions with object trackers. The default `State` implies a static target at 1 meter from the observer at zero azimuth and elevation.

`mscef = trackingMSCEKF(Name,Value)` specifies the properties of the filter using one or more `Name,Value` pair arguments. Any unspecified properties take default values.

## Properties

### State — Filter state

`[0;0;0;0;1;0]` (default) | real-valued  $M$ -element vector

Filter state, specified as a real-valued  $M$ -element vector.

- For 2-D tracking,  $M$  is equal to four and the four-dimensional state is: `[az; azRate; 1/r; rDot/r]`.

For 3-D tracking,  $M$  is equal to six and the six-dimensional state is:  $[az; azRate; el; elRate; 1/r; rDot/r]$ .

$az$  and  $el$  are the azimuth and elevation angle in radians.  $azRate$  and  $elRate$  are the azimuth and elevation angular rate in radians per second.  $r$  is the range in meters, and  $rDot$  is the range rate in meters per second.

Data Types: double

#### **StateCovariance — State estimation error covariance**

1 (default) |  $M$ -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.  $M$  is either 4 for 2-D tracking or 6 for 3-D tracking.

Example: `eye(6)`

#### **StateTransitionFcn — State transition function**

`@constvelmsc` (default)

This property is read-only.

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  $k-1$ . For the `trackingMSCEKF` object, the transition function is fixed to `@constvelmsc`.

Data Types: function\_handle

#### **StateTransitionJacobianFcn — State transition function Jacobian**

`@constvelmscjac` (default)

This property is read-only.

The Jacobian of the state transition function, specified as a function handle. This function has the same input arguments as the state transition function. For the `trackingMSCEKF` object, the transition function Jacobian is fixed to `@constvelmsc`.

Data Types: function\_handle

#### **ProcessNoise — Process noise covariance**

1 (default) | positive real-valued scalar | positive-definite real-valued matrix

Process noise covariance, specified as a  $Q$ -by- $Q$  matrix.  $Q$  is either 2 or 3. The process noise represents uncertainty in the acceleration of the target.

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]`

#### **ObserverInput — Acceleration or maneuver of observer**

`[0;0;0]` (default) |  $M$ /2-element vector |  $M$ -element vector

Acceleration or maneuver of the observer, specified as a three-element vector. To specify an acceleration, use an  $M/2$  vector, where  $M$  is either 4 for 2-D tracking or 6 for 3-D tracking. To specify a maneuver, give an  $M$ -element vector.

Example: [1;2;3]

**HasAdditiveProcessNoise — Model additive process noise**  
false (default)

This property is read-only.

Model additive process noise, specified as false. For the trackingMSCEKF object, this property is fixed to false.

**MeasurementFcn — Measurement model function**  
@cvmeasm sc (default)

This property is read-only.

Measurement model function, specified as a function handle, @cvmeasm sc. Input to the function is the  $M$ -element state vector. The output is the  $N$ -element measurement vector. For the trackingMSCEKF object, the measurement model function is fixed to @cvmeasm sc.

Data Types: function\_handle

**MeasurementJacobianFcn — Jacobian of measurement function**  
@cvmeasm sc jac

This property is read-only.

Jacobian of the measurement function, specified as a function handle. The function has the same input arguments as the measurement function. For the trackingMSCEKF object, the Jacobian of the measurement function is fixed to @cvmeasm sc jac.

Data Types: function\_handle

**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 specified as a scalar, the matrix is a multiple of the  $N$ -by- $N$  identity matrix.  $N$  is the size of the measurement vector.

Specify MeasurementNoise before any call to the correct function.

Example: 0.2

**HasAdditiveMeasurementNoise — Model additive measurement noise**  
true (default)

This property is read-only.

Model additive process noise, specified as true. For the trackingMSCEKF object, this property is fixed to true.

## Object Functions

predict	Predict state and state estimation error covariance of tracking filter
correct	Correct state and state estimation error covariance using tracking filter
correctjpda	Correct state and state estimation error covariance using tracking filter and JPDA

---

distance	Distances between current and predicted measurements of tracking filter
likelihood	Likelihood of measurement from tracking filter
clone	Create duplicate tracking filter
residual	Measurement residual and residual noise from tracking filter
initialize	Initialize state and covariance of tracking filter

## Examples

### Create MSC-EKF Tracking Object for 3-D Motion Model

This example shows how to make an extended Kalman filter (EKF) for object tracking in modified spherical coordinates (MSC). Create the filter, predict the state, and correct the state estimate using measurement observations.

Create the filter for a 3-D motion model. Specify the state estimates for the MSC frame.

```
az = 0.1; % in radians
azRate = 0;
r = 1000;
rDot = 10;
el = 0.3; % in radians
elRate = 0;
omega = azRate*cos(el);

mscekf = trackingMSCEKF('State',[az;omega;el;elRate;1/r;rDot/r]);
```

Predict the filter state using a constant observer acceleration.

```
mscekf.ObserverInput = [1;2;3];
predict(mscekf); % Default time 1 second.
predict(mscekf,0.1); % Predict using dt = 0.1 second.
```

Correct the filter state using an angle-only measurement.

```
meas = [5;18]; % measured azimuth and elevation in degrees
correct(mscekf,meas);
```

## References

- [1] Aidala, V. and Hammel, S., 1983. *Utilization of modified polar coordinates for bearings-only tracking*. IEEE Transactions on Automatic Control, 28(3), pp.283-294.

## See Also

[trackingCKF](#) | [trackingEKF](#) | [trackingGSF](#) | [trackingIMM](#) | [trackingPF](#)

## Introduced in R2018b

# trackingPF

Particle filter for object tracking

## Description

The `trackingPF` object represents an object tracker that follows a nonlinear motion model or that is measured by a nonlinear measurement model. The filter uses a set of discrete particles to approximate the posterior distribution of the state. The particle filter can be applied to arbitrary nonlinear system models. The process and measurement noise can follow an arbitrary non-Gaussian distribution.

The particles are generated using various resampling methods defined by `ResamplingMethod`.

## Creation

### Syntax

```
pf = trackingPF  
pf = trackingPF(transitionFcn,measurementFcn,state)  
pf = trackingPF(___,Name,Value)
```

### Description

`pf = trackingPF` returns a `trackingPF` object with state transition function, `@constvel`, measurement function, `@cvmeas`, and a distribution of particles around the state, `[0;0;0;0]`, with unit covariance in each dimension. The filter assumes an additive Gaussian process noise model and Gaussian likelihood calculations.

`pf = trackingPF(transitionFcn,measurementFcn,state)` specifies the `StateTransitionFcn`, `MeasurementFcn`, and `State` properties directly. The filter assumes a unit covariance around the state.

`pf = trackingPF(___,Name,Value)` specifies the properties of the particle filter using one or more `Name,Value` pair arguments. Any unspecified properties take default values.

## Properties

### State — Current filter state

real-valued  $M$ -element vector

This property is read-only.

Current filter state, specified as a real-valued  $M$ -element vector. The current state is calculated from `Particles` and `Weight` using the specified `StateEstimationMethod`.  $M$  is the `NumStateVariables`. `StateOrientation` determines if the state is given as a row or column vector.

Example: `[0.1;0.05;0.04;-0.01]`

Data Types: double

**StateCovariance — State estimation error covariance**  
*M*-by-*M* matrix

This property is read-only.

State error covariance, specified as an *M*-by-*M* matrix, where *M* is the size of the filter state. The current state covariance is calculated from Particles and Weight using the specified StateEstimationMethod. *M* is the NumStateVariables. The covariance matrix represents the uncertainty in the filter state.

**IsStateVariableCircular — Indicates if state variables have circular distribution**  
[0 0 0 0] (default) | *M*-element vector of zeros and ones

This property is read-only.

Indicates if state variables have circular distribution, specified as an *M*-element vector of zeros and ones. Values of 1 indicate it does have a circular distribution. The probability density function of a circular variable takes on angular values in the range [-pi, pi].

**StateOrientation — Orientation of state vector**  
'column' (default) | 'row'

Orientation of state vector, specified as 'column' or 'row'.

---

**Note** If you set the orientation to 'row', the default StateTransitionFcn and MeasurementFcn are not supported. All state transition functions and measurement functions provided (constvel and cvmeas, for example) assume a 'column' orientation.

---

**StateTransitionFcn — State transition function**  
@constvel (default) | function handle

State transition function, specified as a function handle. The state transition function evolves the system state from each particle. The callback function accepts at least one input argument, prevParticles, that represents the system at the previous time step. If StateOrientation is 'row', the particles are input as a NumParticles-by-NumStateVariables array. If StateOrientation is 'column', the particles are input as a NumStateVariables-by-NumParticles array.

Additional input arguments can be provided with varargin, which are passed to the predict function. The function signature is:

```
function predictParticles = stateTransitionFcn(prevParticles,varargin)
```

When the HasAdditiveProcessNoise property of the filter is false, the state transition function can accept an additional input argument, dt. For example:

```
function predictParticles = stateTransitionFcn(prevParticles,dt,varargin)
```

dt is the time step of the trackingPF filter, filter, that was specified in the most recent call to the predict function. The dt argument applies when you use the filter within a tracker and call the predict 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: `predict(filter,dt)`

### Dependencies

This parameter depends on the `StateOrientation` property.

Data Types: `function_handle`

**ProcessNoiseSamplingFcn — Function to generate noise sample for each particle**  
@gaussianSampler (default) | function handle

Function to generate noise sample for each particle, specified as a function handle. The function signature is:

```
function noiseSample = processNoiseSamplingFcn(pf)
```

- When `HasAdditiveProcessNoise` is `false`, this function outputs a noise sample as a  $W$ -by- $N$  matrix, where  $W$  is the number of process noise terms, and  $N$  is the number of particles.
- When `HasAdditiveProcessNoise` is `true`, this function outputs a noise sample as an  $M$ -by- $N$  matrix, where  $M$  is the number of state variables, and  $N$  is the number of particles.

To generate a sample from a non-Gaussian distribution, use this property with a custom function handle.

### Dependencies

This parameter depends on the `HasAdditiveProcessNoise` property.

Data Types: `function_handle`

**ProcessNoise — Process noise covariance**

1 (default) | positive real-valued scalar | positive-definite real-valued matrix

Process noise covariance:

- When `HasAdditiveProcessNoise` is `true`, specify the process noise covariance as a 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.

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.

If `ProcessNoiseSamplingFcn` is specified as `@gaussianSample`, this property defines the Gaussian noise covariance of the process.

Example: `[1.0 0.05; 0.05 2]`

### Dependencies

This parameter depends on the `HasAdditiveProcessNoise` property.

**HasAdditiveProcessNoise — Model additive process noise**  
`true` (default) | `false`

Option to model processes 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**

`@cvmeas` (default) | function handle

Measurement model function, specified as a function handle. This function calculates the measurements given the current particles' state. Additional input arguments can be provided with `varargin`. The function signature is:

```
function predictedParticles = measurementFcn(particles,varargin)
```

Data Types: `function_handle`

#### **MeasurementLikelihoodFcn — Callback function calculating the likelihood of sensor measurements**

`@gaussianLikelihood` (default) | function handle

Callback function calculating the likelihood of sensor measurements, specified as a function handle. Once a sensor measurement is available, this callback function calculates the likelihood that the measurement is consistent with the state hypothesis of each particle.

The callback function accepts at least three input arguments, `pf`, `predictedParticles`, and `measurement`. There are two function signatures:

```
function likelihood = measurementLikelihoodFcn(pf,predictedParticles,measurement,varargin)
function [likelihood,distance] = measurementLikelihoodFcn(pf,predictedParticles,measurement,varargin)
```

`pf` is the particle filter object.

`predictedParticles` represents the set of particles returned from `MeasurementFcn`. If `StateOrientation` is '`row`', the particles are input as a `NumParticles`-by-`NumStateVariables` array. If `StateOrientation` is '`column`', the particles are input as a `NumStateVariables`-by-`NumParticles` array.

`measurement` is the state measurement at the current time step.

`varargin` allows you to specify additional inputs to the `correct` function.

The callback output, `likelihood`, is a vector of length `NumParticles`, which is the likelihood of the given measurement for each particle state hypothesis.

The optional output, `distance`, allows you to specify the distance calculations returned by the `distance` function.

Data Types: `function_handle`

#### **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 specified as a scalar, the matrix is a multiple of the  $N$ -by- $N$  identity matrix.  $N$  is the size of the measurement vector.

If `MeasurementLikelihoodFcn` is specified as `@gaussianLikelihood`, this property is used to specify the Gaussian noise covariance of the measurement.

Example: `0.2`

**Particles — State hypothesis of each particle**

matrix

State hypothesis of each particle, specified as a matrix. If `StateOrientation` is 'row' the particles are a `NumParticles`-by-`NumStateVariables` array. If `StateOrientation` is 'column', the particles are a `NumStateVariables`-by-`NumParticles` array.

Each row or column corresponds to the state hypothesis of a single particle.

Data Types: double

**Weights — Particle weights**

`ones(1,NumParticles)` (default) | vector

Particle weights, specified as a vector. The vector is either a row or column vector based on `StateOrientation`. Each row or column is the weight associated with the same row or column in `Particles`.

Data Types: double

**NumStateVariables — Number of state variables**

4 (default) | integer

Number of state variables, specified as an integer. The `State` is comprised of this number of state variables.

**NumParticles — Number of particles used**

1000 (default) | integer

Number of particles used by the filter, specified as an integer. Each particle represents a state hypothesis.

**ResamplingPolicy — Policy settings for triggering resampling**

`trackingResamplingPolicy` object

Policy settings for triggering resampling, specified as a `trackingResamplingPolicy` object. The resampling can be triggered either at fixed intervals or dynamically based on the number of effective particles.

**ResamplingMethod — Method used for particle resampling**

`'multinomial'` (default) | `'systemic'` | `'stratified'` | `'residual'`

Method used for particle resampling, specified as '`multinomial`', '`systemic`', '`stratified`', or '`residual`'.

**StateEstimationMethod — Method used for state estimation**

`'mean'` (default) | `'maxweight'`

Method used for state estimation, specified as '`mean`' or '`maxweight`'.

## 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>initialize</code>	Initialize state and covariance of tracking filter

## Examples

### Run trackingPF Filter

This example shows how to create and run a `trackingPF` filter. Call the `predict` and `correct` functions to track an object and correct the state estimate based on measurements.

Create the filter. Specify the initial state and state covariance. Specify the number of particles and that there is additive process noise.

```
state = [0;0;0;0];
stateCov = 10*eye(4);
pf = trackingPF(@constvel,@cvmeas,state,'StateCovariance',stateCov, ...
    'NumParticles',2500,'HasAdditiveProcessNoise',true);
```

Call `predict` to get the predicted state and covariance of the filter. Use a 0.5 sec time step.

```
[xPred,pPred] = predict(pf,0.5);
```

You can also modify the particles in the filter to carry a multi-model state hypothesis. Modify the `Particle` property with particles around multiple states after initialization.

```
state1 = [0;0;0;0];
stateCov1 = 10*eye(4);
state2 = [100;0;100;0];
stateCov2 = 10*eye(4);

pf.Particles(:,1:1000) = (state1 + chol(stateCov1)*randn(4,1000));
pf.Particles(:,1001:2000) = (state2 + chol(stateCov2)*randn(4,1000));
```

Call `correct` with a given measurement.

```
meas = [1;1;0];
[xCorr,pCorr] = correct(pf,meas);
```

Continue to predict the filter state. Specify the desired time step in seconds if necessary.

```
[xPred,pPred] = predict(pf);           % Predict over 1 second
[xPred,pPred] = predict(pf,2);         % Predict over 2 seconds
```

## References

- [1] Arulampalam, M.S., S. Maskell, N. Gordon, and T. Clapp. "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking." *IEEE Transactions on Signal Processing*. Vol. 50, No. 2, Feb 2002, pp. 174-188.

[2] Chen, Z. "Bayesian Filtering: From Kalman Filters to Particle Filters, and Beyond." *Statistics*. Vol. 182, No. 1, 2003, pp. 1-69.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[constvel](#) | [cvmeas](#) | [trackingCKF](#) | [trackingEKF](#) | [trackingKF](#) | [trackingUKF](#)

### Introduced in R2018b

# trackScoreLogic

Confirm and delete tracks based on track score

## Description

The `trackScoreLogic` object determines if a track should be confirmed or deleted based on the track score (also known as the log likelihood of a track). A track should be confirmed if the current track score is greater than or equal to the confirmation threshold. A track should be deleted if the current track score has decreased relative to the maximum track score by the deletion threshold.

The confirmation and deletion decisions contribute to the track management by a `trackerGNN` or `trackerTOMHT`.

## Creation

### Syntax

```
logic = trackScoreLogic
logic = trackScoreLogic(Name,Value,...)
```

### Description

`logic = trackScoreLogic` creates a `trackScoreLogic` object with default confirmation and deletion thresholds.

`logic = trackScoreLogic(Name,Value,...)` specifies the `ConfirmationThreshold` and `DeletionThreshold` properties of the track score logic object using one or more `Name,Value` pair arguments. Any unspecified properties take default values.

## Properties

### ConfirmationThreshold — Confirmation threshold

20 (default) | positive scalar

Confirmation threshold, specified as a positive scalar. If the logic score is above this threshold, then the track is confirmed.

Data Types: `single` | `double`

### DeletionThreshold — Deletion threshold

-5 (default) | negative scalar

Deletion threshold, specified as a negative scalar. If the value of `Score - MaxScore` is more negative than the deletion threshold, then the track is deleted.

Data Types: `single` | `double`

### Score — Current track logic score

numeric scalar

This property is read-only.

Current track logic score, specified as a numeric scalar.

#### MaxScore — Maximum track logic score

numeric scalar

This property is read-only.

Maximum track logic score, specified as a numeric scalar.

## Object Functions

init	Initialize track logic with first hit
hit	Update track logic with subsequent hit
miss	Update track logic with miss
sync	Synchronize scores of trackScoreLogic objects
mergeScores	Update track score by track merging
checkConfirmation	Check if track should be confirmed
checkDeletion	Check if track should be deleted
output	Get current state of track logic
reset	Reset state of track logic
clone	Create copy of track logic

## Examples

### Create and Update Score-Based Logic

Create a score-based logic. Specify the confirmation threshold as 20 and the deletion threshold as -5.

```
scoreLogic = trackScoreLogic('ConfirmationThreshold',20,'DeletionThreshold',-5)
```

```
scoreLogic =  
trackScoreLogic with properties:
```

```
ConfirmationThreshold: 20  
DeletionThreshold: -5  
Score: 0  
MaxScore: 0
```

Specify the probability of detection (**pd**), the probability of false alarm (**pfa**), the volume of a sensor detection bin (**volume**), and the new target rate in a unit volume (**beta**). Initialize the logic using these parameters. The first update to the logic is a hit.

```
pd = 0.9; % Probability of detection  
pfa = 1e-6; % Probability of false alarm  
volume = 1; % Volume of a sensor detection bin  
beta = 0.1; % New target rate in a unit volume  
  
init(scoreLogic,volume,beta,pd,pfa);  
  
disp(['Score and MaxScore: ', num2str(output(scoreLogic))])  
Score and MaxScore: 11.4076 11.4076
```

Update the logic four more times, where only the odd updates register a hit. The score increases with each hit and decreases with each miss. The confirmation flag is `true` whenever the current score is larger than 20.

```
for i = 2:5

    isOdd = logical(mod(i,2));
    if isOdd
        likelihood = 0.05 + 0.05*rand(1);
        hit(scoreLogic,volume,likelihood)
    else
        miss(scoreLogic)
    end

    confFlag = checkConfirmation(scoreLogic);
    delFlag = checkDeletion(scoreLogic);
    disp(['Score and MaxScore: ', num2str(output(scoreLogic)), ...
        '. Confirmation Flag: ',num2str(confFlag), ...
        '. Deletion Flag: ',num2str(delFlag)])
end

Score and MaxScore: 9.10498      11.4076. Confirmation Flag: 0. Deletion Flag: 0
Score and MaxScore: 20.4153      20.4153. Confirmation Flag: 1. Deletion Flag: 0
Score and MaxScore: 18.1127      20.4153. Confirmation Flag: 0. Deletion Flag: 0
Score and MaxScore: 29.4721      29.4721. Confirmation Flag: 1. Deletion Flag: 0
```

Update the logic with a miss three times. The deletion flag is `true` by the end of the third miss, because the difference between the current score and maximum score is greater than five.

```
for i = 1:3
    miss(scoreLogic)

    confFlag = checkConfirmation(scoreLogic);
    delFlag = checkDeletion(scoreLogic);
    disp(['Score and MaxScore: ', num2str(output(scoreLogic)), ...
        '. Confirmation Flag: ',num2str(confFlag), ...
        '. Deletion Flag: ',num2str(delFlag)])
end

Score and MaxScore: 27.1695      29.4721. Confirmation Flag: 1. Deletion Flag: 0
Score and MaxScore: 24.8669      29.4721. Confirmation Flag: 1. Deletion Flag: 0
Score and MaxScore: 22.5643      29.4721. Confirmation Flag: 1. Deletion Flag: 1
```

## Tips

- If you specify either `ConfirmationThreshold` or `DeletionThreshold` in single precision, then the `trackScoreLogic` object converts the other property to single precision and performs computations in single precision.

## 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

[trackHistoryLogic](#) | [trackerGNN](#)

### Topics

"Introduction to Track Logic"

**Introduced in R2018b**

# mergeScores

Update track score by track merging

## Syntax

```
mergeScores(scoreLogic1,scoreLogic2)
```

## Description

`mergeScores(scoreLogic1,scoreLogic2)` updates the score of `scoreLogic1` by merging the score with the score of `scoreLogic2`. Score merging increases the score of `scoreLogic1` by  $\log(1+\exp(score2-score1))$ .

## Examples

### Merge Score Logics

Create a score logic using the default confirmation and deletion thresholds. Initialize the score logic.

```
scoreLogic1 = trackScoreLogic;
volume = 1.3; % Volume of a sensor detection bin
beta1 = 1e-5; % New target rate in a unit volume
init(scoreLogic1,volume,beta1);
disp(['Score and MaxScore of ScoreLogic1: ', num2str(output(scoreLogic1))])

Score and MaxScore of ScoreLogic1: 2.4596      2.4596
```

Create a copy of the score logic.

```
scoreLogic2 = clone(scoreLogic1);
```

Specify the likelihood that the detection is assigned to the track, the probability of detection (`pd`) and the probability of false alarm (`pfa`). Update the second score logic with a hit.

```
likelihood = 0.05 + 0.05*rand(1);
pd = 0.8;
pfa = 1e-3;
hit(scoreLogic2,volume,likelihood,pd,pfa)
disp(['Score and MaxScore of ScoreLogic2: ', num2str(output(scoreLogic2))])

Score and MaxScore of ScoreLogic2: 7.0068      7.0068
```

Merge the score of `scoreLogic1` with the score of `scoreLogic2`. The score of `scoreLogic2` is larger, therefore the merged score of `scoreLogic1` increases.

```
mergeScores(scoreLogic1,scoreLogic2)
disp(['Score and MaxScore of merged ScoreLogic1: ', num2str(output(scoreLogic1))])

Score and MaxScore of merged ScoreLogic1: 7.0173      7.0173
```

## Input Arguments

### **scoreLogic1 — Track score logic to update**

trackScoreLogic object

Track score logic to update, specified as a `trackScoreLogic` object.

### **scoreLogic2 — Reference track score logic**

trackScoreLogic object

Reference track score logic, specified as a `trackScoreLogic` object.

## Extended Capabilities

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

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

### **See Also**

`clone` | `sync`

**Introduced in R2018b**

# sync

Synchronize scores of `trackScoreLogic` objects

## Syntax

```
sync(scoreLogic1,scoreLogic2)
```

## Description

`sync(scoreLogic1,scoreLogic2)` sets the values of '`Score on page 2-0`' and '`MaxScore on page 2-0`' of `scoreLogic1` to the values of `scoreLogic2`.

## Examples

### Synchronize Track Score Logics

Create a score logic using the default confirmation and deletion thresholds.

```
scoreLogic1 = trackScoreLogic

scoreLogic1 =
    trackScoreLogic with properties:

    ConfirmationThreshold: 20
    DeletionThreshold: -5
        Score: 0
    MaxScore: 0
```

Create a second score logic, specifying the confirmation threshold as 30 and the deletion threshold as -10.

```
scoreLogic2 = trackScoreLogic('ConfirmationThreshold',30,'DeletionThreshold',-10)

scoreLogic2 =
    trackScoreLogic with properties:

    ConfirmationThreshold: 30
    DeletionThreshold: -10
        Score: 0
    MaxScore: 0
```

Initialize the two score logics using different target rates in a unit volume.

```
volume = 1.3; % Volume of a sensor detection bin

beta1 = 0.1; % New target rate in a unit volume
init(scoreLogic1,volume,beta1);
disp(['Score and MaxScore of ScoreLogic1: ', num2str(output(scoreLogic1))])

Score and MaxScore of ScoreLogic1: 11.6699      11.6699
```

```
beta2 = 0.3; % New target rate in a unit volume
init(scoreLogic2,volume,beta2);
disp(['Score and MaxScore of ScoreLogic2: ', num2str(output(scoreLogic2))])
```

```
Score and MaxScore of ScoreLogic2: 12.7685      12.7685
```

Specify the likelihood that a detection is assigned to the track. Then, update the second score logic with a hit.

```
likelihood = 0.05 + 0.05*rand(1);
hit(scoreLogic2,volume,likelihood)
```

```
disp(['Score and MaxScore of ScoreLogic2: ', num2str(output(scoreLogic2))])
```

```
Score and MaxScore of ScoreLogic2: 24.3413      24.3413
```

Synchronize `scoreLogic1` to have the same 'Score' and 'MaxScore' as `scoreLogic2`. The `sync` function does not modify the confirmation or deletion thresholds. To verify this, display the properties of both score logic objects.

```
sync(scoreLogic1,scoreLogic2)
scoreLogic1

scoreLogic1 =
  trackScoreLogic with properties:
```

ConfirmationThreshold: 20  
DeletionThreshold: -5  
Score: 24.3413  
MaxScore: 24.3413

```
scoreLogic2
```

```
scoreLogic2 =
  trackScoreLogic with properties:
```

ConfirmationThreshold: 30  
DeletionThreshold: -10  
Score: 24.3413  
MaxScore: 24.3413

## Input Arguments

**scoreLogic1 — Track score logic to synchronize**  
`trackScoreLogic` object

Track score logic to synchronize, specified as a `trackScoreLogic` object.

**scoreLogic2 — Reference track score logic**  
`trackScoreLogic` object

Reference track score logic, specified as a `trackScoreLogic` object.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`clone` | `mergeScores`

**Introduced in R2018b**

# 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  $Mc$  hits in the recent  $Nc$  updates. A track should be deleted if there are at least  $Md$  misses in the recent  $Nd$  updates.

The confirmation and deletion decisions contribute to the track management by a `trackerGNN` 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  $[Mc \ Nc]$ , where  $Mc$  is the number of hits required for confirmation in the recent  $Nc$  updates. When specified as a scalar, then  $Mc$  and  $Nc$  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  $[Md \ Nd]$ , where  $Md$  is the number of misses required for deletion in the recent  $Nd$  updates. When specified as a scalar, then  $Md$  and  $Nd$  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

init	Initialize track logic with first hit
hit	Update track logic with subsequent hit
miss	Update track logic with miss
checkConfirmation	Check if track should be confirmed
checkDeletion	Check if track should be deleted
output	Get current state of track logic
reset	Reset state of track logic
clone	Create copy of track logic

## Examples

### Create and Update History-Based Logic

Create a history-based logic. Specify confirmation threshold values  $Mc$  and  $Nc$  as the vector [3 5]. Specify deletion threshold values  $Md$  and  $Nd$  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]
```

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 ( $Mc$ ) are counted in the most recent five updates ( $Nc$ ).

```
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]. Confirmation Flag: 0. Deletion Flag: 0
History: [1 0 1 0 0 0]. Confirmation Flag: 0. Deletion Flag: 0
History: [0 1 0 1 0 0]. Confirmation Flag: 0. Deletion Flag: 0
History: [1 0 1 0 1 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 ( $M_d$ ) are counted in the most recent seven updates ( $N_d$ ).

```
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]. Confirmation Flag: 0. Deletion Flag: 0
History: [0 0 1 0 1 0]. Confirmation Flag: 0. Deletion Flag: 0
History: [0 0 0 1 0 1]. Confirmation Flag: 0. Deletion Flag: 0
History: [0 0 0 0 1 0]. Confirmation Flag: 0. Deletion Flag: 0
History: [0 0 0 0 0 1]. Confirmation Flag: 0. Deletion Flag: 1
History: [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

[trackScoreLogic](#) | [trackerGNN](#)

### Topics

“Introduction to Track Logic”

**Introduced in R2018b**

# checkConfirmation

Check if track should be confirmed

## Syntax

```
tf = checkConfirmation(historyLogic)
tf = checkConfirmation(scoreLogic)
```

## 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`.

`tf = checkConfirmation(scoreLogic)` returns a flag that is `true` when the track should be confirmed based on the track score.

## 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
```

## Check Confirmation of Score-Based Logic

Create a score-based logic, specifying the confirmation threshold. The logic uses the default deletion threshold.

```
scoreLogic = trackScoreLogic('ConfirmationThreshold',8);
```

Specify the probability of detection (`pd`), the probability of false alarm (`pfa`), the volume of a sensor detection bin (`volume`), and the new target rate in a unit volume (`beta`).

```
pd = 0.8;
pfa = 1e-3;
volume = 1.3;
beta = 0.1;
```

Initialize the logic using these parameters. The first update to the logic is a hit.

```
init(scoreLogic,volume,beta,pd,pfa);
disp(['Score and MaxScore: ', num2str(output(scoreLogic))]);
```

```
Score and MaxScore: 4.6444 4.6444
```

The confirmation flag is `false` because the score is less than the confirmation threshold.

```
confirmationFlag = checkConfirmation(scoreLogic)
confirmationFlag = logical
0
```

Specify the likelihood that the detection is assigned to the track. Then, update the logic with a hit. The current score and maximum score increase.

```
likelihood = 0.05 + 0.05*rand(1);
hit(scoreLogic,volume,likelihood,pd,pfa)
disp(['Score and MaxScore: ', num2str(output(scoreLogic))])
```

```
Score and MaxScore: 9.1916 9.1916
```

The confirmation flag is now true because the score is greater than the confirmation threshold.

```
confirmationFlag = checkConfirmation(scoreLogic)
confirmationFlag = logical
1
```

## Input Arguments

**historyLogic — Track history logic**  
`trackHistoryLogic`

Track history logic, specified as a `trackHistoryLogic` object.

**scoreLogic — Track score logic**

trackScoreLogic object

Track score logic, specified as a trackScoreLogic 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](#) | [trackScoreLogic](#)

**Introduced in R2018b**

# checkDeletion

Check if track should be deleted

## Syntax

```
tf = checkDeletion(historyLogic)
tf = checkDeletion(historyLogic,tentativeTrack,age)
tf = checkDeletion(scoreLogic)
```

## Description

`tf = checkDeletion(historyLogic)` returns a flag that is `true` when at least  $M_d$  out of  $N_d$  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.

`tf = checkDeletion(scoreLogic)` returns a flag that is `true` when the track should be deleted based on the track score.

## 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
```

## Check Deletion of Score-Based Logic

Create a score-based logic, specifying the deletion threshold. The logic uses the default confirmation threshold.

```
scoreLogic = trackScoreLogic('DeletionThreshold', -1);
```

Specify the probability of detection (`pd`), the probability of false alarm (`pfa`), the volume of a sensor detection bin (`volume`), and the new target rate in a unit volume (`beta`).

```
pd = 0.8;
pfa = 1e-3;
```

```
volume = 1.3;
beta = 0.1;
```

Initialize the logic using these parameters. The first update to the logic is a hit.

```
init(scoreLogic,volume,beta,pd,pfa);
disp(['Score and MaxScore: ', num2str(output(scoreLogic))]);
Score and MaxScore: 4.6444      4.6444
```

Update the logic with a miss. The current score decreases.

```
miss(scoreLogic,pd,pfa)
disp(['Score and MaxScore: ', num2str(output(scoreLogic))])
Score and MaxScore: 3.036      4.6444
```

The deletion flag is `true` because the current score is smaller than the maximum score by more than 1. In other words, `scoreLogic.Score - scoreLogic.MaxScore` is more negative than the deletion threshold, -1.

```
deletionFlag = checkDeletion(scoreLogic)
deletionFlag = 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.

**scoreLogic — Track score logic**  
`trackScoreLogic` object

Track score logic, specified as a `trackScoreLogic` object.

## 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` | `trackScoreLogic`

Introduced in R2018b

## 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  $Mc$  and  $Nc$  as the vector [3 5]. Specify deletion threshold values  $Md$  and  $Nd$  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 logic**

trackHistoryLogic object | trackScoreLogic object

Track logic, specified as a `trackHistoryLogic` object or `trackScoreLogic` object.

## Output Arguments

### **clonedLogic — Cloned track logic**

trackHistoryLogic object | trackScoreLogic object

Cloned track logic, returned as a `trackHistoryLogic` object or `trackScoreLogic` object.

## Extended Capabilities

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

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

## See Also

`trackHistoryLogic` | `trackScoreLogic`

## Introduced in R2018b

## hit

Update track logic with subsequent hit

### Syntax

```
hit(historyLogic)  
hit(scoreLogic, volume, likelihood)  
hit(scoreLogic, volume, likelihood, pd, pfa)
```

### Description

`hit(historyLogic)` updates the track history with a hit.

`hit(scoreLogic, volume, likelihood)` updates the track score in a case of a hit, given the likelihood of a detection being assigned to the track.

`hit(scoreLogic, volume, likelihood, pd, pfa)` updates the track score in a case of a hit, specifying the probability of detection `pd` and probability of false alarm `pfa`.

### 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].
```

#### Update Score Logic with Hit

Create a score-based logic with default confirmation and deletion thresholds.

---

```
scoreLogic = trackScoreLogic;
```

Specify the probability of detection (**pd**), the probability of false alarm (**pfa**), the volume of a sensor detection bin (**volume**), and the new target rate in a unit volume (**beta**).

```
pd = 0.9;
pfa = 1e-6;
volume = 1.3;
beta = 0.1;
```

Initialize the logic using these parameters. The first update to the logic is a hit.

```
init(scoreLogic, volume, beta, pd, pfa);
disp(['Score and MaxScore: ', num2str(output(scoreLogic))]);
Score and MaxScore: 11.6699      11.6699
```

Specify the likelihood that the detection is assigned to the track.

```
likelihood = 0.05 + 0.05*rand(1);
```

Update the logic with a hit. The current score and maximum score increase.

```
hit(scoreLogic, volume, likelihood)
disp(['Score and MaxScore: ', num2str(output(scoreLogic))])
Score and MaxScore: 23.2426      23.2426
```

## Input Arguments

**historyLogic — Track history logic**  
trackHistoryLogic

Track history logic, specified as a **trackHistoryLogic** object.

**scoreLogic — Track score logic**  
trackScoreLogic object

Track score logic, specified as a **trackScoreLogic** object.

**volume — Volume of sensor detection bin**  
nonnegative scalar

Volume of sensor detection bin, specified as a nonnegative scalar. For example, a 2-D radar will have a sensor bin volume of (*azimuth resolution in radians*) \* (*range*) \* (*range resolution*).

Data Types: **single** | **double**

**likelihood — Likelihood of a detection being assigned to the track**  
numeric vector

Likelihood of a detection being assigned to the track, specified as a numeric vector of length *m*.

Data Types: **single** | **double**

**pd — Probability of detection**  
0.9 (default) | nonnegative scalar

Probability of detection, specified as a nonnegative scalar.

Data Types: `single` | `double`

**pfa — Probability of false alarm**

`1e-6` (default) | nonnegative scalar

Probability of false alarm, specified as a nonnegative scalar.

Data Types: `single` | `double`

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`trackHistoryLogic` | `trackScoreLogic`

### Introduced in R2018b

# init

Initialize track logic with first hit

## Syntax

```
init(historyLogic)
init(scoreLogic, volume, beta)
init(scoreLogic, volume, beta, pd, pfa)
```

## Description

`init(historyLogic)` initializes the track history logic with the first hit.

`init(scoreLogic, volume, beta)` initializes the track score logic with the first hit, using default probabilities of detection and false alarm.

`init(scoreLogic, volume, beta, pd, pfa)` initializes the track score logic with the first hit, specifying the probability of detection `pd` and probability of false alarm `pfa`.

## 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].

### Initialize Score-Based Logic

Create a score-based logic with default confirmation and deletion thresholds.

```
scoreLogic = trackScoreLogic
```

```
scoreLogic =  
    trackScoreLogic with properties:  
  
    ConfirmationThreshold: 20  
    DeletionThreshold: -5  
    Score: 0  
    MaxScore: 0
```

Specify the probability of detection (**pd**), the probability of false alarm (**pfa**), the volume of a sensor detection bin (**volume**), and the new target rate in a unit volume (**beta**).

```
pd = 0.9;  
pfa = 1e-6;  
volume = 1.3;  
beta = 0.1;
```

Initialize the logic using these parameters. The first update to the logic is a hit.

```
init(scoreLogic, volume, beta, pd, pfa);
```

Display the current and maximum score of the logic. Since the logic has been updated once, the current score is equal to the maximum score.

```
currentScore = scoreLogic.Score  
currentScore = 11.6699  
maximumScore = scoreLogic.MaxScore  
maximumScore = 11.6699
```

## Input Arguments

**historyLogic — Track history logic**  
trackHistoryLogic object

Track history logic, specified as a `trackHistoryLogic` object.

**scoreLogic — Track score logic**  
trackScoreLogic object

Track score logic, specified as a `trackScoreLogic` object.

**volume — Volume of sensor detection bin**  
nonnegative scalar

Volume of sensor detection bin, specified as a nonnegative scalar. For example, a 2-D radar will have a sensor bin volume of  $(\text{azimuth resolution in radians}) * (\text{range}) * (\text{range resolution})$ .

Data Types: `single` | `double`

**beta — Rate of new targets in unit volume**  
nonnegative scalar

Rate of new targets in unit volume, specified as a nonnegative scalar.

Data Types: single | double

**pd — Probability of detection**

0.9 (default) | nonnegative scalar

Probability of detection, specified as a nonnegative scalar.

Data Types: single | double

**pfa — Probability of false alarm**

1e-6 (default) | nonnegative scalar

Probability of false alarm, specified as a nonnegative scalar.

Data Types: single | double

## Extended Capabilities

**C/C++ Code Generation**

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

## See Also

[trackHistoryLogic](#) | [trackScoreLogic](#)

**Introduced in R2018b**

## miss

Update track logic with miss

### Syntax

```
miss(historyLogic)  
miss(scoreLogic)  
miss(scoreLogic,pd,pfa)
```

### Description

`miss(historyLogic)` updates the track history with a miss.

`miss(scoreLogic)` updates the track score in a case of a miss, using default probabilities of detection and false alarm.

`miss(scoreLogic,pd,pfa)` updates the track score in a case of a miss, specifying the probability of detection `pd` and probability of false alarm `pfa`.

### 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].
```

#### Update Score Logic with Miss

Create a score-based logic with default confirmation and deletion thresholds.

---

```
scoreLogic = trackScoreLogic;
```

Specify the probability of detection (**pd**), the probability of false alarm (**pfa**), the volume of a sensor detection bin (**volume**), and the new target rate in a unit volume (**beta**).

```
pd = 0.9;
pfa = 1e-6;
volume = 1.3;
beta = 0.1;
```

Initialize the logic using these parameters. The first update to the logic is a hit.

```
init(scoreLogic, volume, beta, pd, pfa);
disp(['Score and MaxScore: ', num2str(output(scoreLogic))]);
```

```
Score and MaxScore: 11.6699      11.6699
```

Update the logic with a miss. The current score decreases, but the maximum score does not change.

```
miss(scoreLogic, pd, pfa)
disp(['Score and MaxScore: ', num2str(output(scoreLogic))])
```

```
Score and MaxScore: 9.36735      11.6699
```

## Input Arguments

### **historyLogic — Track history logic**

`trackHistoryLogic`

Track history logic, specified as a `trackHistoryLogic` object.

### **scoreLogic — Track score logic**

`trackScoreLogic` object

Track score logic, specified as a `trackScoreLogic` object.

### **pd — Probability of detection**

`0.9` (default) | nonnegative scalar

Probability of detection, specified as a nonnegative scalar.

Data Types: `single` | `double`

### **pfa — Probability of false alarm**

`1e-6` (default) | nonnegative scalar

Probability of false alarm, specified as a nonnegative scalar.

Data Types: `single` | `double`

## Extended Capabilities

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

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

**See Also**

[trackHistoryLogic](#) | [trackScoreLogic](#)

**Introduced in R2018b**

# output

Get current state of track logic

## Syntax

```
history = output(historyLogic)
scores = output(scoreLogic)
```

## Description

`history = output(historyLogic)` returns the recent history updates of the track history logic object, `historyLogic`.

`scores = output(scoreLogic)` returns in `scores` the current score and maximum score of track score logic object, `scoreLogic`.

## Examples

### Get Recent History of History-Based Logic

Create a history-based logic. Specify confirmation threshold values  $Mc$  and  $Nc$  as the vector [3 5]. Specify deletion threshold values  $Md$  and  $Nd$  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  $Nc$  and  $Nd$ . 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
```

## Get Current Score of Score-Based Logic

Create a score-based logic with default confirmation and deletion thresholds.

```
scoreLogic = trackScoreLogic;
```

Get the current and maximum score of the logic. Both scores are 0 because the logic is not initialized.

```
s = output(scoreLogic)
```

```
s = 1x2
```

```
0 0
```

Specify the volume of a sensor detection bin (`volume`), and the new target rate in a unit volume (`beta`). Initialize the logic using these parameters and the default probabilities of detection and false alarm. The first update to the logic is a hit.

```
volume = 1.3;  
beta = 0.1;  
init(scoreLogic,volume,beta);
```

Get the current and maximum score of the logic.

```
s = output(scoreLogic)
```

```
s = 1x2
```

```
11.6699 11.6699
```

Update the logic with a miss, then get the updated scores.

```
miss(scoreLogic)
```

```
s = output(scoreLogic)
```

```
s = 1x2
```

```
9.3673 11.6699
```

## Input Arguments

### **historyLogic — Track history logic**

```
trackHistoryLogic
```

Track history logic, specified as a `trackHistoryLogic` object.

---

**scoreLogic — Track score logic**  
trackScoreLogic object

Track score logic, specified as a `trackScoreLogic` 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.

**scores — Current and maximum scores**  
1-by-2 numeric vector

Current and maximum scores of `scoreLogic`, returned as a 1-by-2 numeric vector. The first element specifies the current score. The second element specifies the maximum score.

## Extended Capabilities

**C/C++ Code Generation**  
Generate C and C++ code using MATLAB® Coder™.

## See Also

`trackHistoryLogic` | `trackScoreLogic`

**Introduced in R2018b**

## 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

## Reset Track Score Logic

Create a score-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.

```
scoreLogic = trackScoreLogic;
score = output(scoreLogic)

score = 1x2

0      0
```

Initialize the logic, then get the current state of the logic.

```
volume = 1.3;
beta = 0.1;
init(scoreLogic,volume,beta);
score = output(scoreLogic)

score = 1x2

11.6699  11.6699
```

Reset the logic, then get the current state of the logic. The current and maximum score are both 0.

```
reset(scoreLogic)
score = output(scoreLogic)

score = 1x2

0      0
```

## Input Arguments

### logic — Track logic

trackHistoryLogic object | trackScoreLogic object

Track logic, specified as a `trackHistoryLogic` object or `trackScoreLogic` object.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

`trackHistoryLogic` | `trackScoreLogic`

## Introduced in R2018b



# System Objects

---

# altimeterSensor

Altimeter simulation model

## Description

The `altimeterSensor` System object models receiving data from an altimeter sensor.

To model an altimeter:

- 1 Create the `altimeterSensor` 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

```
altimeter = altimeterSensor  
altimeter = altimeterSensor('ReferenceFrame',RF)  
altimeter = altimeterSensor(___,Name,Value)
```

### Description

`altimeter = altimeterSensor` returns an `altimeterSensor` System object that simulates altimeter readings.

`altimeter = altimeterSensor('ReferenceFrame',RF)` returns an `altimeterSensor` System object that simulates altimeter readings relative to the reference frame RF. Specify RF as '`NED`' (North-East-Down) or '`ENU`' (East-North-Up). The default value is '`NED`'.

`altimeter = altimeterSensor(___,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

## 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 — Update rate of sensor (Hz)

1 (default) | positive scalar

Update rate of sensor in Hz, specified as a positive scalar.

Data Types: single | double

**ConstantBias — Constant offset bias (m)**

0 (default) | scalar

Constant offset bias in meters, specified as a scalar.

**Tunable:** Yes

Data Types: single | double

**NoiseDensity — Power spectral density of sensor noise (m/ $\sqrt{\text{Hz}}$ )**

0 (default) | nonnegative scalar

Power spectral density of sensor noise in m/ $\sqrt{\text{Hz}}$ , specified as a nonnegative scalar.

**Tunable:** Yes

Data Types: single | double

**BiasInstability — Instability of bias offset (m)**

0 (default) | nonnegative scalar

Instability of the bias offset in meters, specified as a nonnegative scalar.

**Tunable:** Yes

Data Types: single | double

**DecayFactor — Bias instability noise decay factor**

0 (default) | scalar in the range [0,1]

Bias instability noise decay factor, specified as a scalar in the range [0,1]. A decay factor of 0 models the bias instability noise as a white noise process. A decay factor of 1 models the bias instability noise as a random walk process.

**Tunable:** Yes

Data Types: single | double

**RandomStream — Random number source**

'Global stream' (default) | 'mt19937ar with seed'

Random number source, specified as a character vector or string:

- 'Global stream' -- Random numbers are generated using the current global random number stream.
- 'mt19937ar with seed' -- Random numbers are generated using the mt19937ar algorithm with the seed specified by the Seed property.

Data Types: char | string

**Seed — Initial seed**

67 (default) | nonnegative integer scalar

Initial seed of an mt19937ar random number generator algorithm, specified as a nonnegative integer scalar.

#### Dependencies

To enable this property, set `RandomStream` to '`mt19937ar` with `seed`'.

Data Types: `single` | `double`

#### Usage

`altimeterReadings = altimeter(position)`

#### Description

`altimeterReadings = altimeter(position)` generates an altimeter sensor altitude reading from the `position` input.

#### Input Arguments

##### **`position` — Position of sensor in local navigation coordinate system (m)**

*N*-by-3 matrix

Position of sensor in the local navigation coordinate system, specified as an *N*-by-3 matrix with elements measured in meters. *N* is the number of samples in the current frame.

Data Types: `single` | `double`

#### Output Arguments

##### **`altimeterReadings` — Altitude of sensor relative to local navigation coordinate system (m)**

*N*-element column vector

Altitude of sensor relative to the local navigation coordinate system in meters, returned as an *N*-element column vector. *N* is the number of samples in the current frame.

Data Types: `single` | `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

## Generate Noisy Altimeter Readings from Stationary Input

Create an `altimeterSensor` System object™ to model receiving altimeter sensor data. Assume a typical one Hz sample rate and a 10 minute simulation time. Set `ConstantBias` to `0.01`, `NoiseDensity` to `0.05`, `BiasInstability` to `0.05`, and `DecayFactor` to `0.5`.

```
Fs = 1;
duration = 60*10;
numSamples = duration*Fs;

altimeter = altimeterSensor('SampleRate',Fs, ...
    'ConstantBias',0.01, ...
    'NoiseDensity',0.05, ...
    'BiasInstability',0.05, ...
    'DecayFactor',0.5);

truePosition = zeros(numSamples,3);
```

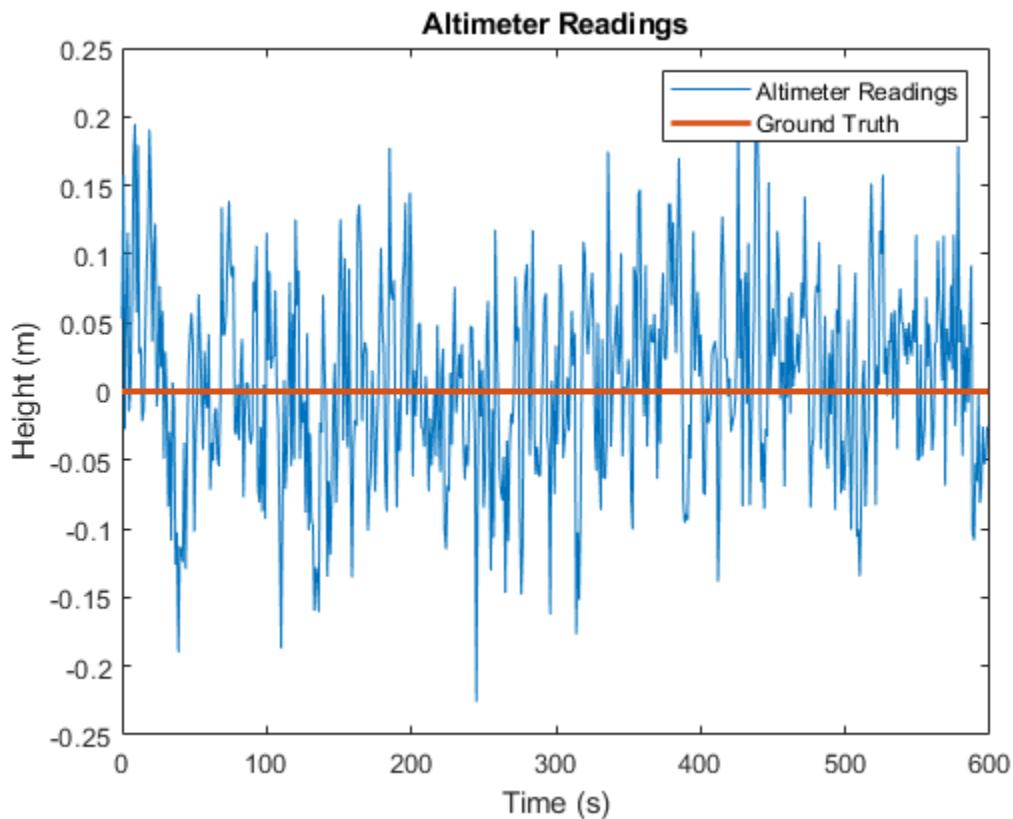
Call `altimeter` with the specified `truePosition` to model noisy altimeter readings from a stationary platform.

```
altimeterReadings = altimeter(truePosition);
```

Plot the true position and the altimeter sensor readings for height.

```
t = (0:(numSamples-1))/Fs;

plot(t,altimeterReadings)
hold on
plot(t,truePosition(:,3), 'LineWidth',2)
hold off
title('Altimeter Readings')
xlabel('Time (s)')
ylabel('Height (m)')
legend('Altimeter Readings','Ground Truth')
```



## 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

[gpsSensor](#) | [imuSensor](#) | [insSensor](#)

### Topics

“Model IMU, GPS, and INS/GPS”

### Introduced in R2019a

# ahrsfilter

Orientation from accelerometer, gyroscope, and magnetometer readings

## Description

The `ahrsfilter` System object fuses accelerometer, magnetometer, and gyroscope sensor data to estimate device orientation.

To estimate device orientation:

- 1** Create the `ahrsfilter` 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

```
FUSE = ahrsfilter
FUSE = ahrsfilter('ReferenceFrame',RF)
FUSE = ahrsfilter(___,Name,Value)
```

### Description

`FUSE = ahrsfilter` returns an indirect Kalman filter System object, `FUSE`, for sensor fusion of accelerometer, gyroscope, and magnetometer data to estimate device orientation and angular velocity. The filter uses a 12-element state vector to track the estimation error for the orientation, the gyroscope bias, the linear acceleration, and the magnetic disturbance.

`FUSE = ahrsfilter('ReferenceFrame',RF)` returns an `ahrsfilter` System object that fuses accelerometer, gyroscope, and magnetometer data to estimate device orientation relative to the reference frame RF. Specify RF as '`NED`' (North-East-Down) or '`ENU`' (East-North-Up). The default value is '`NED`'.

`FUSE = ahrsfilter(___,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

## 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 — Input sample rate of sensor data (Hz)**

100 (default) | positive scalar

Input sample rate of the sensor data in Hz, specified as a positive scalar.

**Tunable:** No

Data Types: single | double

**DecimationFactor — Decimation factor**

1 (default) | positive integer

Decimation factor by which to reduce the input sensor data rate as part of the fusion algorithm, specified as a positive integer.

The number of rows of the inputs -- `accelReadings`, `gyroReadings`, and `magReadings` -- must be a multiple of the decimation factor.

Data Types: single | double

**AccelerometerNoise — Variance of accelerometer signal noise ((m/s<sup>2</sup>)<sup>2</sup>)**

0.00019247 (default) | positive real scalar

Variance of accelerometer signal noise in (m/s<sup>2</sup>)<sup>2</sup>, specified as a positive real scalar.

**Tunable:** Yes

Data Types: single | double

**MagnetometerNoise — Variance of magnetometer signal noise (μT<sup>2</sup>)**

0.1 (default) | positive real scalar

Variance of magnetometer signal noise in μT<sup>2</sup>, specified as a positive real scalar.

**Tunable:** Yes

Data Types: single | double

**GyroscopeNoise — Variance of gyroscope signal noise ((rad/s)<sup>2</sup>)**

9.1385e-5 (default) | positive real scalar

Variance of gyroscope signal noise in (rad/s)<sup>2</sup>, specified as a positive real scalar.

**Tunable:** Yes

Data Types: single | double

**GyroscopeDriftNoise — Variance of gyroscope offset drift ((rad/s)<sup>2</sup>)**

3.0462e-13 (default) | positive real scalar

Variance of gyroscope offset drift in (rad/s)<sup>2</sup>, specified as a positive real scalar.

**Tunable:** Yes

Data Types: single | double

**LinearAccelerationNoise — Variance of linear acceleration noise (m/s<sup>2</sup>)<sup>2</sup>**

0.0096236 (default) | positive real scalar

Variance of linear acceleration noise in  $(m/s^2)^2$ , specified as a positive real scalar. Linear acceleration is modeled as a lowpass-filtered white noise process.

**Tunable:** Yes

Data Types: single | double

**LinearAccelerationDecayFactor — Decay factor for linear acceleration drift**

0.5 (default) | scalar in the range [0,1)

Decay factor for linear acceleration drift, specified as a scalar in the range [0,1]. If linear acceleration is changing quickly, set `LinearAccelerationDecayFactor` to a lower value. If linear acceleration changes slowly, set `LinearAccelerationDecayFactor` to a higher value. Linear acceleration drift is modeled as a lowpass-filtered white noise process.

**Tunable:** Yes

Data Types: single | double

**MagneticDisturbanceNoise — Variance of magnetic disturbance noise ( $\mu T^2$ )**

0.5 (default) | real finite positive scalar

Variance of magnetic disturbance noise in  $\mu T^2$ , specified as a real finite positive scalar.

**Tunable:** Yes

Data Types: single | double

**MagneticDisturbanceDecayFactor — Decay factor for magnetic disturbance**

0.5 (default) | positive scalar in the range [0,1]

Decay factor for magnetic disturbance, specified as a positive scalar in the range [0,1]. Magnetic disturbance is modeled as a first order Markov process.

**Tunable:** Yes

Data Types: single | double

**InitialProcessNoise — Covariance matrix for process noise**

12-by-12 matrix

Covariance matrix for process noise, specified as a 12-by-12 matrix. The default is:

Columns 1 through 6

0.000006092348396	0	0	0	0	0
0	0.000006092348396	0	0	0	0
0	0	0.000006092348396	0	0	0
0	0	0	0.000076154354947	0	0.000076154354947
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Columns 7 through 12

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0.009623610000000	0	0	0
0	0.009623610000000	0	0
0	0	0.009623610000000	0
0	0	0	0.600000000000000
0	0	0	0
0	0	0	0.600000000000000

The initial process covariance matrix accounts for the error in the process model.

Data Types: `single` | `double`

**ExpectedMagneticFieldStrength — Expected estimate of magnetic field strength ( $\mu\text{T}$ )**  
`50` (default) | real positive scalar

Expected estimate of magnetic field strength in  $\mu\text{T}$ , specified as a real positive scalar. The expected magnetic field strength is an estimate of the magnetic field strength of the Earth at the current location.

**Tunable:** Yes

Data Types: `single` | `double`

**OrientationFormat — Output orientation format**  
`'quaternion'` (default) | `'Rotation matrix'`

Output orientation format, specified as `'quaternion'` or `'Rotation matrix'`. The size of the output depends on the input size,  $N$ , and the output orientation format:

- `'quaternion'` -- Output is an  $N$ -by-1 quaternion.
- `'Rotation matrix'` -- Output is a 3-by-3-by- $N$  rotation matrix.

Data Types: `char` | `string`

## Usage

## Syntax

```
[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings,magReadings)
```

## Description

```
[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings,magReadings)
```

fuses accelerometer, gyroscope, and magnetometer data to compute orientation and angular velocity measurements. The algorithm assumes that the device is stationary before the first call.

## Input Arguments

**accelReadings — Accelerometer readings in sensor body coordinate system (m/s<sup>2</sup>)**  
*N*-by-3 matrix

Accelerometer readings in the sensor body coordinate system in m/s<sup>2</sup>, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `accelReadings` represent the [x y z] measurements. Accelerometer readings are assumed to correspond to the sample rate specified by the `SampleRate` property.

Data Types: `single` | `double`

**gyroReadings — Gyroscope readings in sensor body coordinate system (rad/s)**  
*N*-by-3 matrix

Gyroscope readings in the sensor body coordinate system in rad/s, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `gyroReadings` represent the [x y z] measurements. Gyroscope readings are assumed to correspond to the sample rate specified by the `SampleRate` property.

Data Types: `single` | `double`

**magReadings — Magnetometer readings in sensor body coordinate system (μT)**  
*N*-by-3 matrix

Magnetometer readings in the sensor body coordinate system in μT, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `magReadings` represent the [x y z] measurements. Magnetometer readings are assumed to correspond to the sample rate specified by the `SampleRate` property.

Data Types: `single` | `double`

## Output Arguments

**orientation — Orientation that rotates quantities from local navigation coordinate system to sensor body coordinate system**

*M*-by-1 array of quaternions (default) | 3-by-3-by-*M* array

Orientation that can rotate quantities from the local navigation coordinate system to a body coordinate system, returned as quaternions or an array. The size and type of `orientation` depends on whether the `OrientationFormat` property is set to '`'quaternion'`' or '`'Rotation matrix'`':

- '`'quaternion'`' -- the output is an *M*-by-1 vector of quaternions, with the same underlying data type as the inputs
- '`'Rotation matrix'`' -- the output is a 3-by-3-by-*M* array of rotation matrices the same data type as the inputs

The number of input samples, *N*, and the `DecimationFactor` property determine *M*.

You can use `orientation` in a `rotateframe` function to rotate quantities from a local navigation system to a sensor body coordinate system.

Data Types: `quaternion` | `single` | `double`

**angularVelocity — Angular velocity in sensor body coordinate system (rad/s)**

*M*-by-3 array (default)

Angular velocity with gyroscope bias removed in the sensor body coordinate system in rad/s, returned as an  $M$ -by-3 array. The number of input samples,  $N$ , and the `DecimationFactor` property determine  $M$ .

Data Types: `single` | `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 `ahrsfilter`

`tune` Tune `ahrsfilter` parameters to reduce estimation error

### 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
<code>reset</code>	Reset internal states of System object

## Examples

### Estimate Orientation Using `ahrsfilter`

Load the `rpy_9axis` file, which contains recorded accelerometer, gyroscope, and magnetometer sensor data from a device oscillating in pitch (around y-axis), then yaw (around z-axis), and then roll (around x-axis). The file also contains the sample rate of the recording.

```
load 'rpy_9axis' sensorData Fs
accelerometerReadings = sensorData.Acceleration;
gyroscopeReadings = sensorData.AngularVelocity;
magnetometerReadings = sensorData.MagneticField;
```

Create an `ahrsfilter` System object™ with `SampleRate` set to the sample rate of the sensor data. Specify a decimation factor of two to reduce the computational cost of the algorithm.

```
decim = 2;
fuse = ahrsfilter('SampleRate',Fs,'DecimationFactor',decim);
```

Pass the accelerometer readings, gyroscope readings, and magnetometer readings to the `ahrsfilter` object, `fuse`, to output an estimate of the sensor body orientation over time. By default, the orientation is output as a vector of quaternions.

```
q = fuse(accelerometerReadings,gyroscopeReadings,magnetometerReadings);
```

Orientation is defined by angular displacement required to rotate a parent coordinate system to a child coordinate system. Plot the orientation in Euler angles in degrees over time.

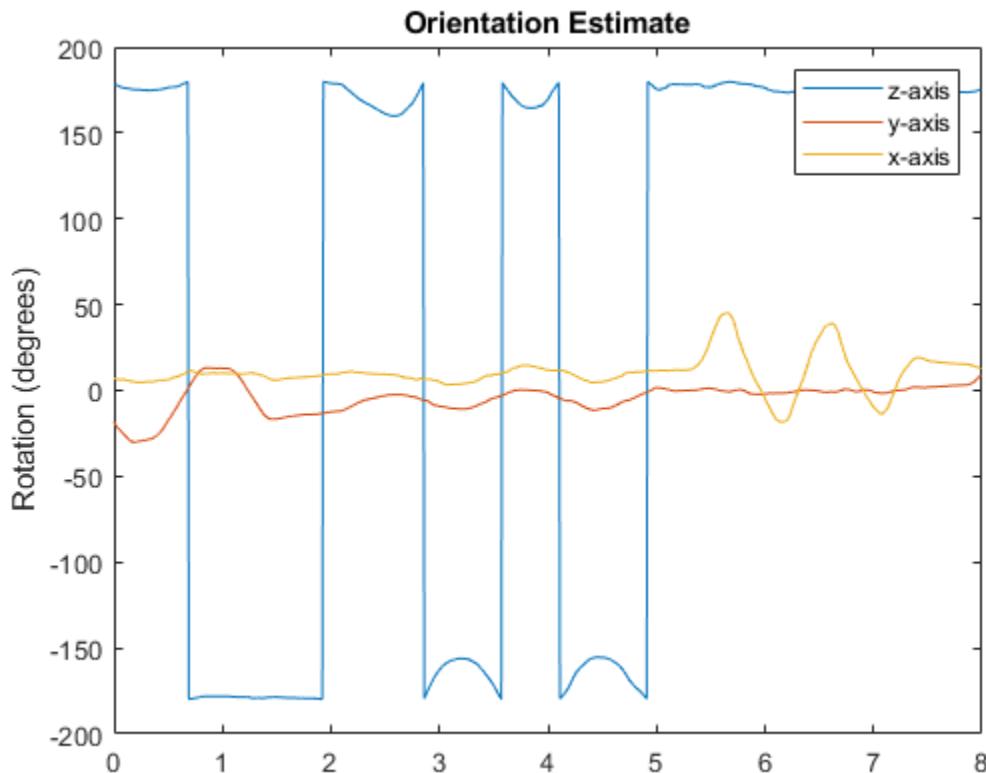
`ahrsfilter` correctly estimates the change in orientation over time, including the south-facing initial orientation.

```

time = (0:decim:size(accelerometerReadings,1)-1)/Fs;

plot(time,eulerd(q,'ZYX','frame'))
title('Orientation Estimate')
legend('z-axis', 'y-axis', 'x-axis')
ylabel('Rotation (degrees)')

```



### Simulate Magnetic Jamming on ahrsFilter

This example shows how performance of the `ahrsfilter` System object™ is affected by magnetic jamming.

Load `StationaryIMUReadings`, which contains accelerometer, magnetometer, and gyroscope readings from a stationary IMU.

```

load 'StationaryIMUReadings.mat' accelReadings magReadings gyroReadings SampleRate
numSamples = size(accelReadings,1);

```

The `ahrsfilter` uses magnetic field strength to stabilize its orientation against the assumed constant magnetic field of the Earth. However, there are many natural and man-made objects which output magnetic fields and can confuse the algorithm. To account for the presence of transient magnetic fields, you can set the `MagneticDisturbanceNoise` property on the `ahrsfilter` object.

Create an `ahrsfilter` object with the decimation factor set to 2 and note the default expected magnetic field strength.

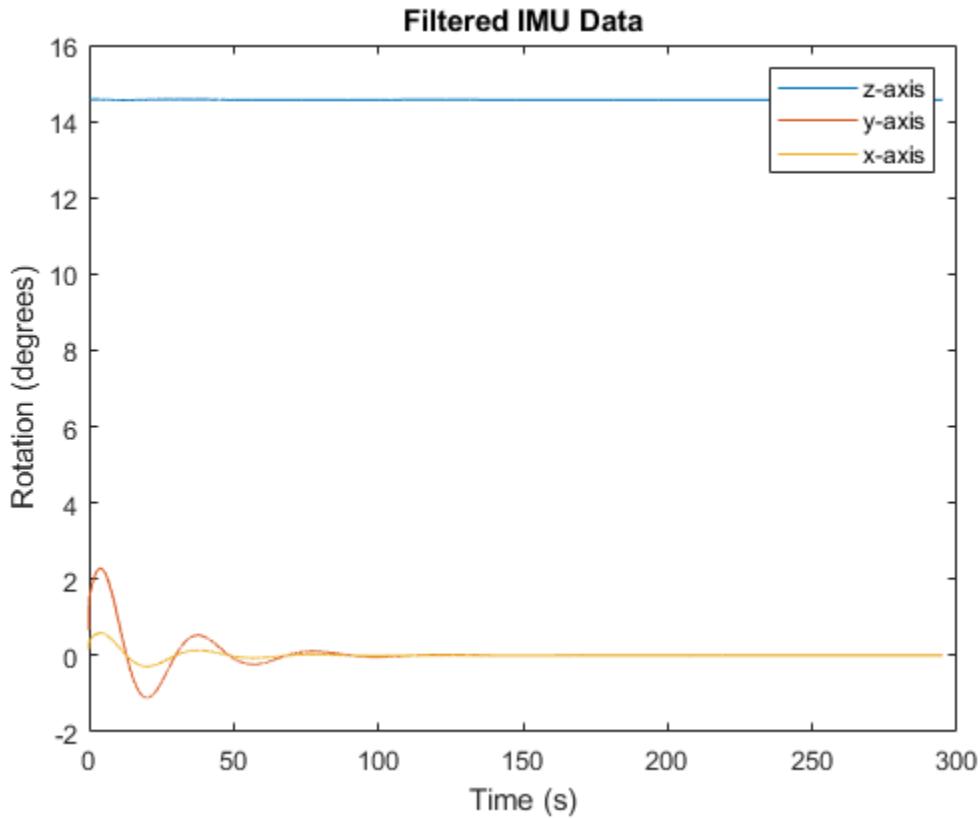
```
decim = 2;
FUSE = ahrsfilter('SampleRate', SampleRate, 'DecimationFactor', decim);
```

Fuse the IMU readings using the attitude and heading reference system (AHRS) filter, and then visualize the orientation of the sensor body over time. The orientation fluctuates at the beginning and stabilizes after approximately 60 seconds.

```
orientation = FUSE(accelReadings, gyroReadings, magReadings);

orientationEulerAngles = eulerd(orientation, 'ZYX', 'frame');
time = (0:decim:(numSamples-1))/SampleRate;

figure(1)
plot(time, orientationEulerAngles(:,1), ...
      time, orientationEulerAngles(:,2), ...
      time, orientationEulerAngles(:,3))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
legend('z-axis', 'y-axis', 'x-axis')
title('Filtered IMU Data')
```



Mimic magnetic jamming by adding a transient, strong magnetic field to the magnetic field recorded in the `magReadings`. Visualize the magnetic field jamming.

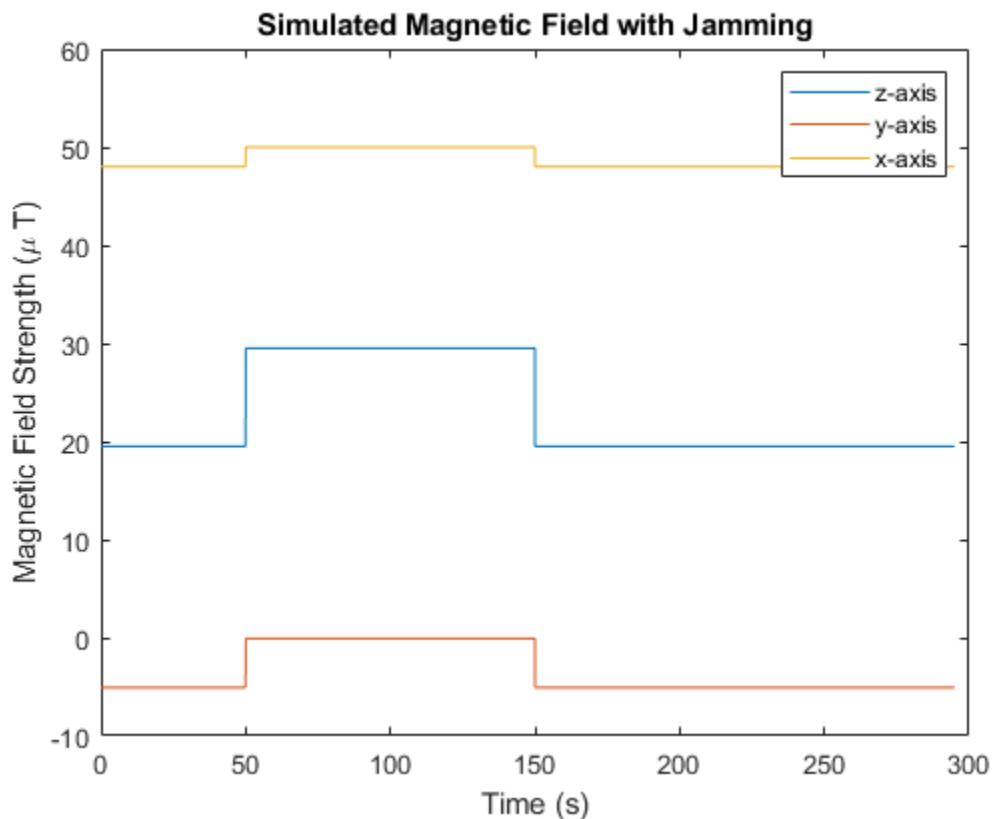
```

jamStrength = [10,5,2];
startStop = (50*SampleRate):(150*SampleRate);
jam = zeros(size(magReadings));
jam(startStop,:) = jamStrength.*ones(numel(startStop),3);

magReadings = magReadings + jam;

figure(2)
plot(time,magReadings(1:decim:end,:))
xlabel('Time (s)')
ylabel('Magnetic Field Strength ( $\mu$  T)')
title('Simulated Magnetic Field with Jamming')
legend('z-axis','y-axis','x-axis')

```



Run the simulation again using the `magReadings` with magnetic jamming. Plot the results and note the decreased performance in orientation estimation.

```

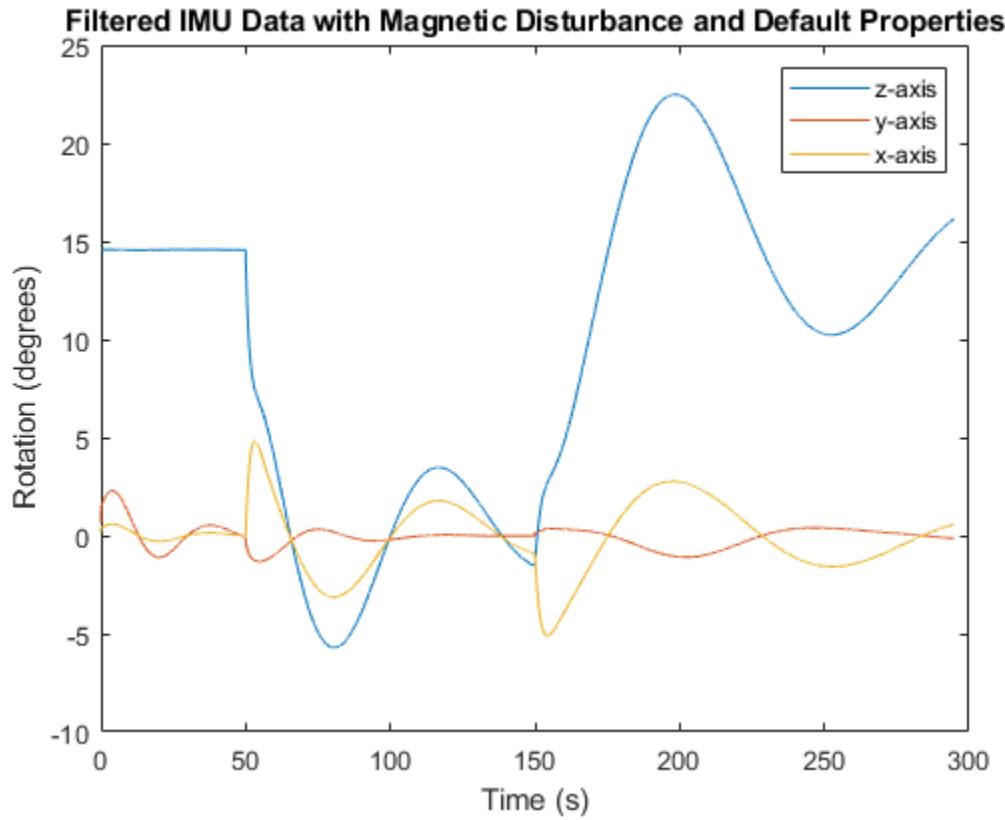
reset(FUSE)
orientation = FUSE(accelReadings,gyroReadings,magReadings);

orientationEulerAngles = eulerd(orientation,'ZYX','frame');

figure(3)
plot(time,orientationEulerAngles(:,1), ...
      time,orientationEulerAngles(:,2), ...
      time,orientationEulerAngles(:,3))
xlabel('Time (s)')
ylabel('Rotation (degrees)')

```

```
legend('z-axis','y-axis','x-axis')
title('Filtered IMU Data with Magnetic Disturbance and Default Properties')
```



The magnetic jamming was misinterpreted by the AHRS filter, and the sensor body orientation was incorrectly estimated. You can compensate for jamming by increasing the `MagneticDisturbanceNoise` property. Increasing the `MagneticDisturbanceNoise` property increases the assumed noise range for magnetic disturbance, and the entire magnetometer signal is weighted less in the underlying fusion algorithm of `ahrsfilter`.

Set the `MagneticDisturbanceNoise` to 200 and run the simulation again.

The orientation estimation output from `ahrsfilter` is more accurate and less affected by the magnetic transient. However, because the magnetometer signal is weighted less in the underlying fusion algorithm, the algorithm may take more time to restabilize.

```
reset(FUSE)
FUSE.MagneticDisturbanceNoise = 20;

orientation = FUSE(accelReadings,gyroReadings,magReadings);

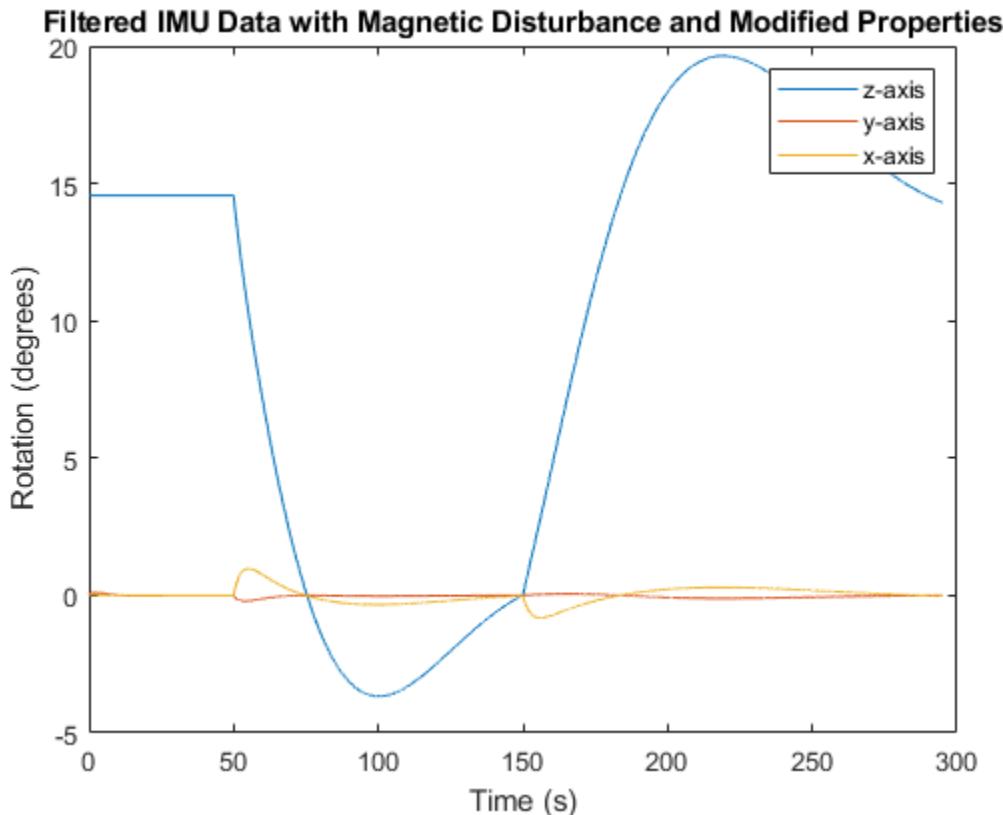
orientationEulerAngles = eulerd(orientation,'ZYX','frame');

figure(4)
plot(time,orientationEulerAngles(:,1), ...
      time,orientationEulerAngles(:,2), ...
      time,orientationEulerAngles(:,3))
xlabel('Time (s)')
```

```

ylabel('Rotation (degrees)')
legend('z-axis','y-axis','x-axis')
title('Filtered IMU Data with Magnetic Disturbance and Modified Properties')

```



### Track Shaking 9-Axis IMU

This example uses the `ahrsfilter` System object™ to fuse 9-axis IMU data from a sensor body that is shaken. Plot the quaternion distance between the object and its final resting position to visualize performance and how quickly the filter converges to the correct resting position. Then tune parameters of the `ahrsfilter` so that the filter converges more quickly to the ground-truth resting position.

Load `IMUReadingsShaken` into your current workspace. This data was recorded from an IMU that was shaken then laid in a resting position. Visualize the acceleration, magnetic field, and angular velocity as recorded by the sensors.

```

load 'IMUReadingsShaken' accelReadings gyroReadings magReadings SampleRate
numSamples = size(accelReadings,1);
time = (0:(numSamples-1))/SampleRate;

figure(1)
subplot(3,1,1)
plot(time,accelReadings)
title('Accelerometer Reading')
ylabel('Acceleration (m/s^2)')

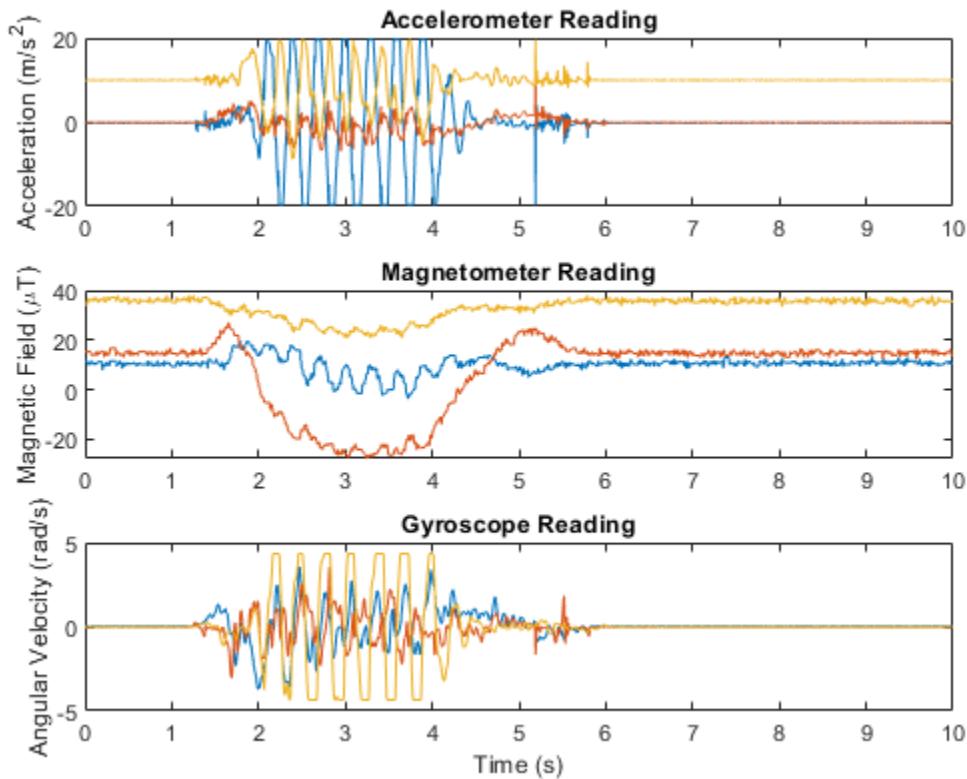
```

```

subplot(3,1,2)
plot(time,magReadings)
title('Magnetometer Reading')
ylabel('Magnetic Field (\muT)')

subplot(3,1,3)
plot(time,gyroReadings)
title('Gyroscope Reading')
ylabel('Angular Velocity (rad/s)')
xlabel('Time (s)')

```



Create an `ahrsfilter` and then fuse the IMU data to determine orientation. The orientation is returned as a vector of quaternions; convert the quaternions to Euler angles in degrees. Visualize the orientation of the sensor body over time by plotting the Euler angles required, at each time step, to rotate the global coordinate system to the sensor body coordinate system.

```

fuse = ahrsfilter('SampleRate',SampleRate);
orientation = fuse(accelReadings,gyroReadings,magReadings);

orientationEulerAngles = eulerd(orientation,'ZYX','frame');

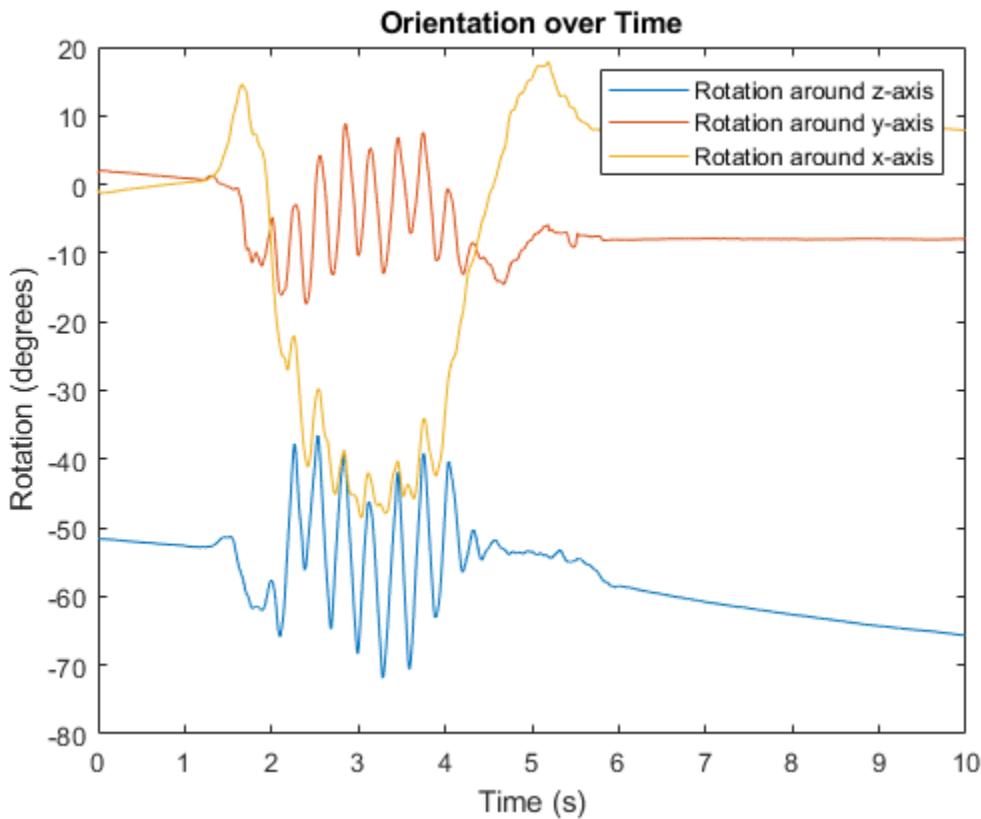
figure(2)
plot(time,orientationEulerAngles(:,1), ...
      time,orientationEulerAngles(:,2), ...
      time,orientationEulerAngles(:,3))
xlabel('Time (s)')

```

```

ylabel('Rotation (degrees)')
title('Orientation over Time')
legend('Rotation around z-axis', ...
'Rotation around y-axis', ...
'Rotation around x-axis')

```



In the IMU recording, the shaking stops after approximately six seconds. Determine the resting orientation so that you can characterize how fast the `ahrsfilter` converges.

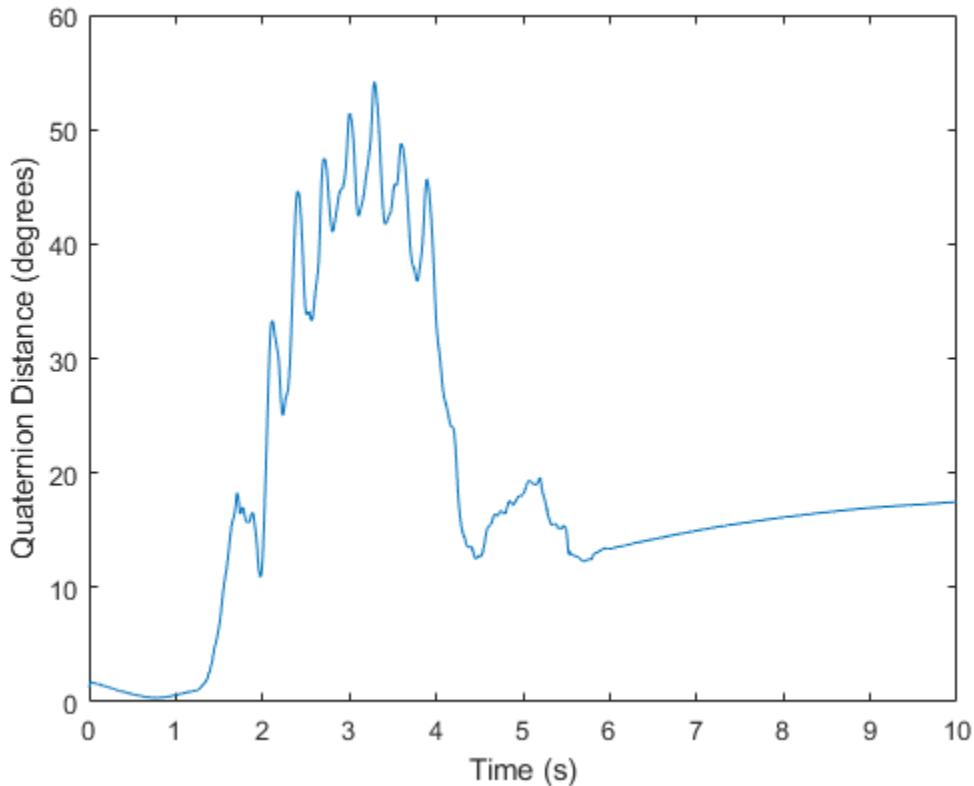
To determine the resting orientation, calculate the averages of the magnetic field and acceleration for the final four seconds and then use the `ecompass` function to fuse the data.

Visualize the quaternion distance from the resting position over time.

```

restingOrientation = ecompass(mean(accelReadings(6*SampleRate:end,:)), ...
mean(magReadings(6*SampleRate:end,:)));
figure(3)
plot(time, rad2deg(dist(restingOrientation,orientation)))
hold on
xlabel('Time (s)')
ylabel('Quaternion Distance (degrees)')

```



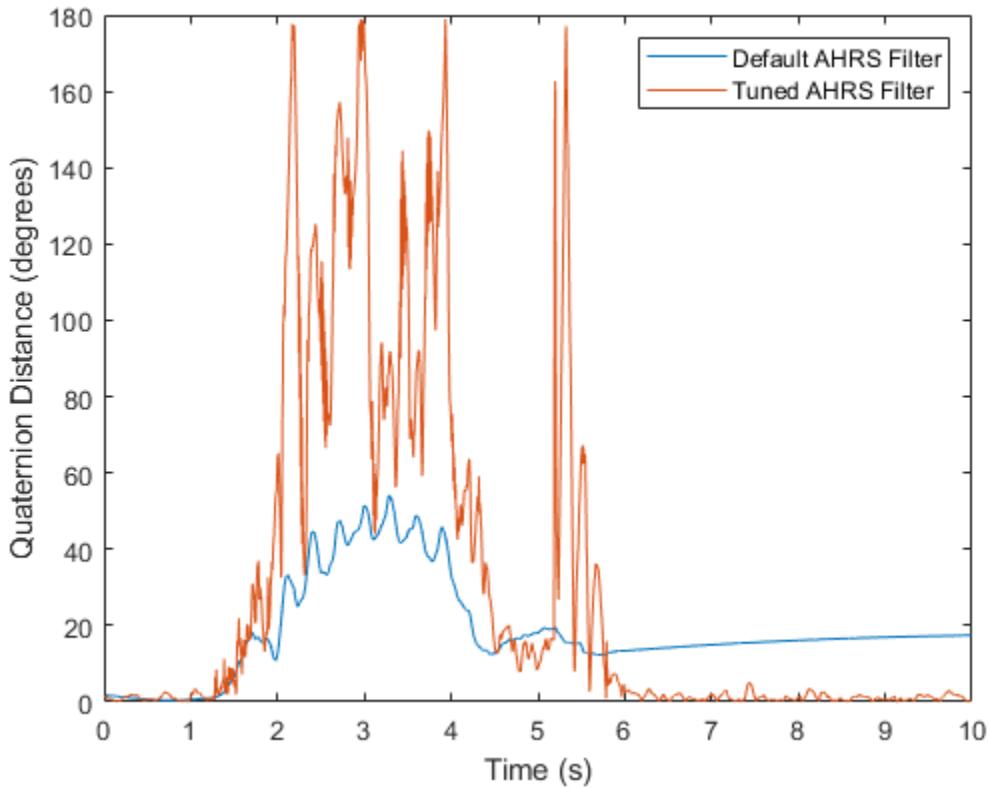
Modify the default `ahrsfilter` properties so that the filter converges to gravity more quickly. Increase the `GyroscopeDriftNoise` to `1e-2` and decrease the `LinearAccelerationNoise` to `1e-4`. This instructs the `ahrsfilter` algorithm to weigh gyroscope data less and accelerometer data more. Because the `accelerometer` data provides the stabilizing and consistent gravity vector, the resulting orientation converges more quickly.

Reset the filter, fuse the data, and plot the results.

```
fuse.LinearAccelerationNoise = 1e-4;
fuse.GyroscopeDriftNoise    = 1e-2;
reset(fuse)

orientation = fuse(accelReadings,gyroReadings,magReadings);

figure(3)
plot(time,rad2deg(dist(restingOrientation,orientation)))
legend('Default AHRS Filter','Tuned AHRS Filter')
```



## Algorithms

*Note: The following algorithm only applies to an NED reference frame.*

The `ahrsfilter` uses the nine-axis Kalman filter structure described in [1]. The algorithm attempts to track the errors in orientation, gyroscope offset, linear acceleration, and magnetic disturbance to output the final orientation and angular velocity. Instead of tracking the orientation directly, the indirect Kalman filter models the error process,  $x$ , with a recursive update:

$$x_k = \begin{bmatrix} \theta_k \\ b_k \\ a_k \\ d_k \end{bmatrix} = F_k \begin{bmatrix} \theta_{k-1} \\ b_{k-1} \\ a_{k-1} \\ d_{k-1} \end{bmatrix} + w_k$$

where  $x_k$  is a 12-by-1 vector consisting of:

- $\theta_k$  -- 3-by-1 orientation error vector, in degrees, at time  $k$
- $b_k$  -- 3-by-1 gyroscope zero angular rate bias vector, in deg/s, at time  $k$
- $a_k$  -- 3-by-1 acceleration error vector measured in the sensor frame, in g, at time  $k$
- $d_k$  -- 3-by-1 magnetic disturbance error vector measured in the sensor frame, in  $\mu\text{T}$ , at time  $k$

and where  $w_k$  is a 12-by-1 additive noise vector, and  $F_k$  is the state transition model.

Because  $x_k$  is defined as the error process, the *a priori* estimate is always zero, and therefore the state transition model,  $F_k$ , is zero. This insight results in the following reduction of the standard Kalman equations:

Standard Kalman equations:

$$\begin{aligned}x_k^- &= F_k x_{k-1}^+ \\P_k^- &= F_k P_{k-1}^+ F_k^T + Q_k \\y_k &= z_k - H_k x_k^- \\S_k &= R_k + H_k P_k^- H_k^T \\K_k &= P_k^- H_k^T (S_k)^{-1} \\x_k^+ &= x_k^- + K_k y_k \\P_k^+ &= P_k^- - K_k H_k P_k^-\end{aligned}$$

Kalman equations used in this algorithm:

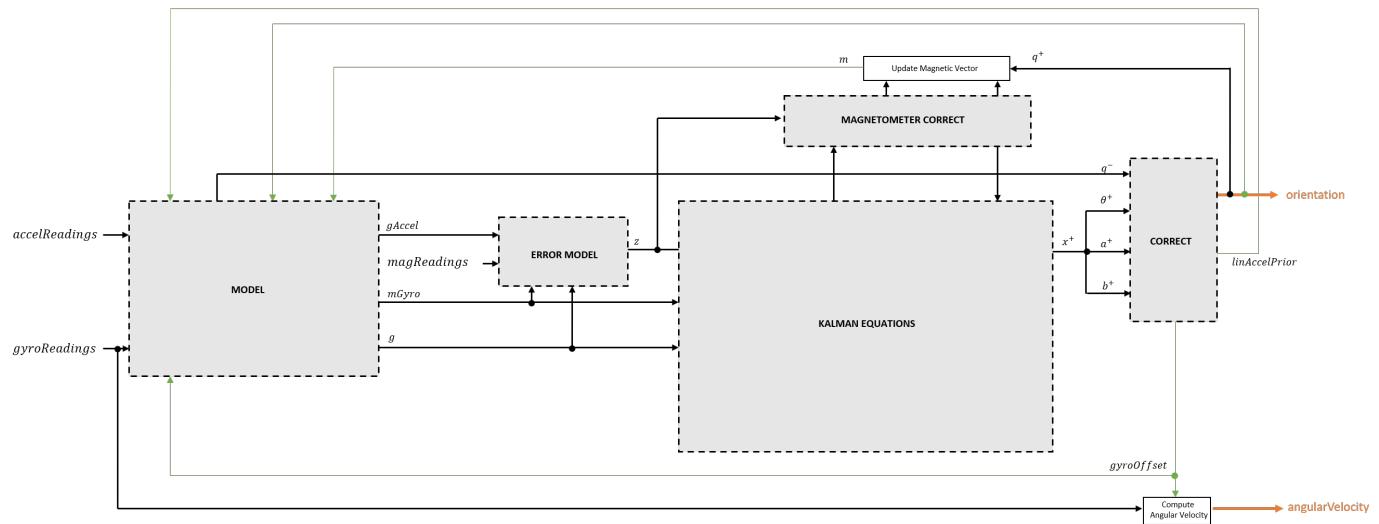
$$\begin{aligned}x_k^- &= 0 \\P_k^- &= Q_k \\y_k &= z_k \\S_k &= R_k + H_k P_k^- H_k^T \\K_k &= P_k^- H_k^T (S_k)^{-1} \\x_k^+ &= K_k y_k \\P_k^+ &= P_k^- - K_k H_k P_k^-\end{aligned}$$

where:

- $x_k^-$  -- predicted (*a priori*) state estimate; the error process
- $P_k^-$  -- predicted (*a priori*) estimate covariance
- $y_k$  -- innovation
- $S_k$  -- innovation covariance
- $K_k$  -- Kalman gain
- $x_k^+$  -- updated (*a posteriori*) state estimate
- $P_k^+$  -- updated (*a posteriori*) estimate covariance

$k$  represents the iteration, the superscript  $+$  represents an *a posteriori* estimate, and the superscript  $-$  represents an *a priori* estimate.

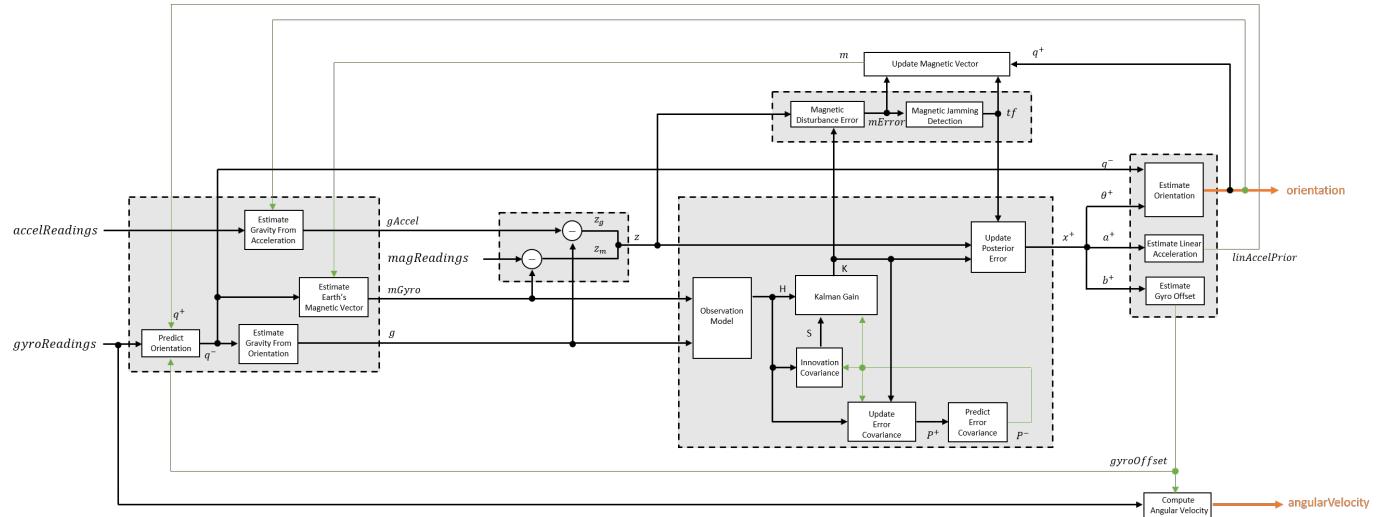
The graphic and following steps describe a single frame-based iteration through the algorithm.



Before the first iteration, the `accelReadings`, `gyroReadings`, and `magReadings` inputs are chunked into `DecimationFactor`-by-3 frames. For each chunk, the algorithm uses the most current accelerometer and magnetometer readings corresponding to the chunk of gyroscope readings.

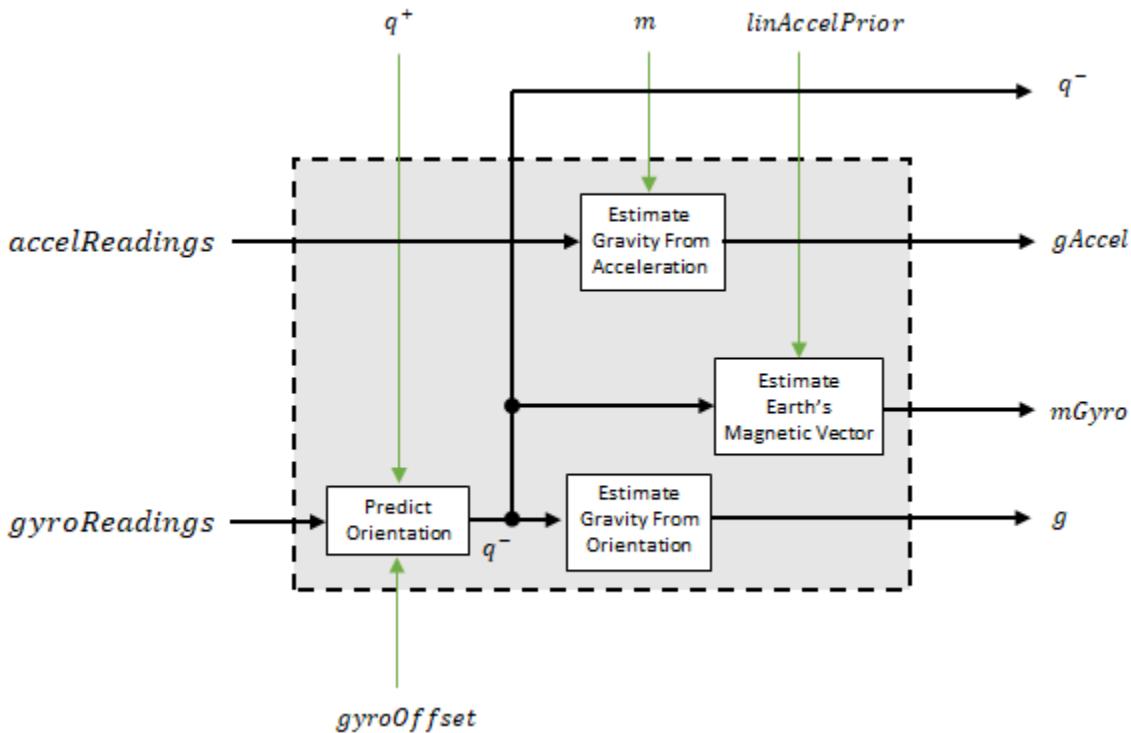
## Detailed Overview

Walk through the algorithm for an explanation of each stage of the detailed overview.



## Model

The algorithm models acceleration and angular change as linear processes.



#### Predict Orientation

The orientation for the current frame is predicted by first estimating the angular change from the previous frame:

$$\Delta\phi_{N \times 3} = \frac{(gyroReadings_{N \times 3} - gyroOffset_{1 \times 3})}{fs}$$

where *N* is the decimation factor specified by the *DecimationFactor* property and *fs* is the sample rate specified by the *SampleRate* property.

The angular change is converted into quaternions using the *rotvec* *quaternion* construction syntax:

$$\Delta Q_{N \times 1} = \text{quaternion}(\Delta\phi_{N \times 3}, 'rotvec')$$

The previous orientation estimate is updated by rotating it by  $\Delta Q$ :

$$q_{1 \times 1}^{-} = (q_{1 \times 1}^{+}) \left( \prod_{n=1}^N \Delta Q_n \right)$$

During the first iteration, the orientation estimate, *q<sup>-</sup>*, is initialized by *ecompass*.

#### Estimate Gravity from Orientation

The gravity vector is interpreted as the third column of the quaternion, *q<sup>-</sup>*, in rotation matrix form:

$$g_{1 \times 3} = (rPrior(:, 3))^T$$

See [1] for an explanation of why the third column of *rPrior* can be interpreted as the gravity vector.

### Estimate Gravity from Acceleration

A second gravity vector estimation is made by subtracting the decayed linear acceleration estimate of the previous iteration from the accelerometer readings:

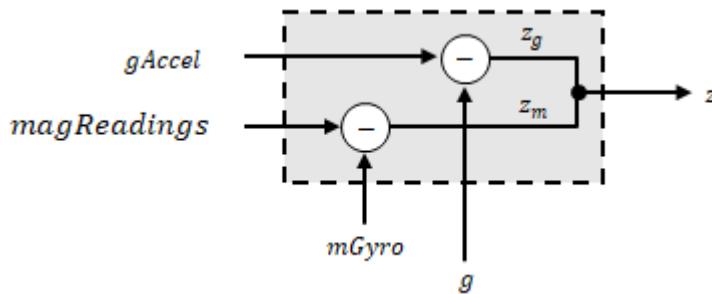
$$gAccel_{1 \times 3} = accelReadings_{1 \times 3} - linAccelprior_{1 \times 3}$$

### Estimate Earth's Magnetic Vector

Earth's magnetic vector is estimated by rotating the magnetic vector estimate from the previous iteration by the *a priori* orientation estimate, in rotation matrix form:

$$mGyro_{1 \times 3} = ((rPrior)(m^T))^T$$

### Error Model

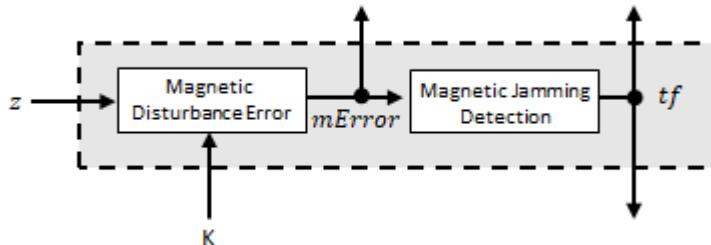


The error model combines two differences:

- The difference between the gravity estimate from the accelerometer readings and the gravity estimate from the gyroscope readings:  $z_g = g - gAccel$
- The difference between the magnetic vector estimate from the gyroscope readings and the magnetic vector estimate from the magnetometer:  $z_m = mGyro - magReadings$

### Magnetometer Correct

The magnetometer correct estimates the error in the magnetic vector estimate and detects magnetic jamming.



### Magnetometer Disturbance Error

The magnetic disturbance error is calculated by matrix multiplication of the Kalman gain associated with the magnetic vector with the error signal:

$$mError_{3 \times 1} = ((K(10:12, :)_3 \times 6)(z_{1 \times 6}))^T$$

The Kalman gain,  $K$ , is the Kalman gain calculated in the current iteration.

### Magnetic Jamming Detection

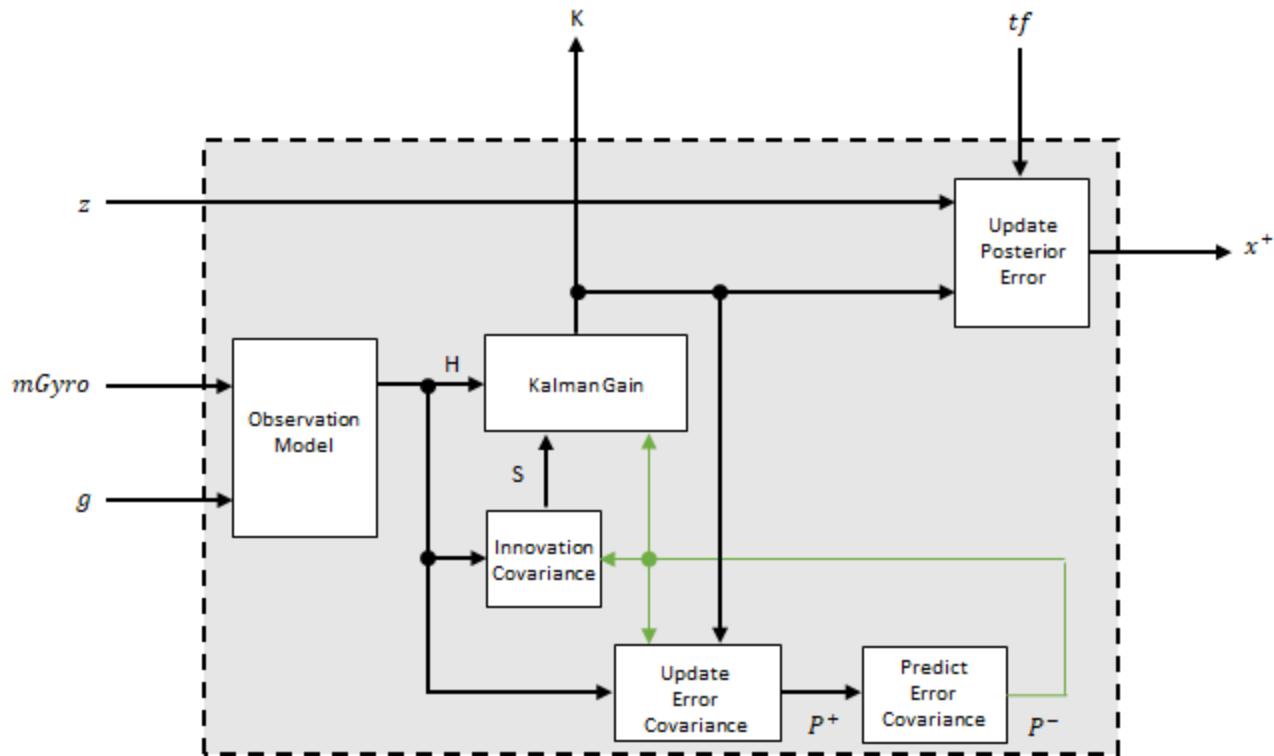
Magnetic jamming is determined by verifying that the power of the detected magnetic disturbance is less than or equal to four times the power of the expected magnetic field strength:

$$tf = \begin{cases} \text{true} & \text{if } \sum |mError|^2 > (4)(\text{ExpectedMagneticFieldStrength})^2 \\ \text{false} & \text{else} \end{cases}$$

ExpectedMagneticFieldStrength is a property of `ahrsfilter`.

### Kalman Equations

The Kalman equations use the gravity estimate derived from the gyroscope readings,  $g$ , the magnetic vector estimate derived from the gyroscope readings,  $mGyro$ , and the observation of the error process,  $z$ , to update the Kalman gain and intermediary covariance matrices. The Kalman gain is applied to the error signal,  $z$ , to output an *a posteriori* error estimate,  $x^+$ .



### Observation Model

The observation model maps the 1-by-3 observed states,  $g$  and  $mGyro$ , into the 6-by-12 true state,  $H$ .

The observation model is constructed as:

$$H_{3 \times 9} = \begin{bmatrix} 0 & g_z & -g_y & 0 & -\kappa g_z & \kappa g_y & 1 & 0 & 0 & 0 & 0 & 0 \\ -g_z & 0 & g_x & \kappa g_z & 0 & -\kappa g_x & 0 & 1 & 0 & 0 & 0 & 0 \\ g_y & -g_x & 0 & -\kappa g_y & \kappa g_x & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & m_z & -m_y & 0 & -\kappa m_z & -\kappa m_y & 0 & 0 & 0 & -1 & 0 & 0 \\ -m_z & 0 & m_x & \kappa m_z & 0 & -\kappa m_x & 0 & 0 & 0 & 0 & -1 & 0 \\ m_y & -m_x & 0 & -\kappa m_y & \kappa m_x & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

where  $g_x$ ,  $g_y$ , and  $g_z$  are the  $x$ -,  $y$ -, and  $z$ -elements of the gravity vector estimated from the *a priori* orientation, respectively.  $m_x$ ,  $m_y$ , and  $m_z$  are the  $x$ -,  $y$ -, and  $z$ -elements of the magnetic vector estimated from the *a priori* orientation, respectively.  $\kappa$  is a constant determined by the SampleRate and DecimationFactor properties:  $\kappa = \text{DecimationFactor}/\text{SampleRate}$ .

See sections 7.3 and 7.4 of [1] for a derivation of the observation model.

### Innovation Covariance

The innovation covariance is a 6-by-6 matrix used to track the variability in the measurements. The innovation covariance matrix is calculated as:

$$S_{6 \times 6} = R_{6 \times 6} + (H_{6 \times 12})(P_{12 \times 12}^-)(H_{6 \times 12})^T$$

where

- $H$  is the observation model matrix
- $P^-$  is the predicted (*a priori*) estimate of the covariance of the observation model calculated in the previous iteration
- $R$  is the covariance of the observation model noise, calculated as:

$$R_{6 \times 6} = \begin{bmatrix} \text{accel}_{\text{noise}} & 0 & 0 & 0 & 0 & 0 \\ 0 & \text{accel}_{\text{noise}} & 0 & 0 & 0 & 0 \\ 0 & 0 & \text{accel}_{\text{noise}} & 0 & 0 & 0 \\ 0 & 0 & 0 & \text{mag}_{\text{noise}} & 0 & 0 \\ 0 & 0 & 0 & 0 & \text{mag}_{\text{noise}} & 0 \\ 0 & 0 & 0 & 0 & 0 & \text{mag}_{\text{noise}} \end{bmatrix}$$

where

$$\begin{aligned} \text{accel}_{\text{noise}} &= \text{AccelerometerNoise} + \text{LinearAccelerationNoise} + \kappa^2 \\ &\quad (\text{GyroscopeDriftNoise} + \text{GyroscopeNoise}) \end{aligned}$$

and

$$\begin{aligned} \text{mag}_{\text{noise}} &= \text{MagnetometerNoise} + \text{MagneticDisturbanceNoise} + \kappa^2 \\ &\quad (\text{GyroscopeDriftNoise} + \text{GyroscopeNoise}) \end{aligned}$$

The following properties define the observation model noise variance:

- $\kappa$  -- DecimationFactor/SampleRate

- AccelerometerNoise
- LinearAccelerationNoise
- GyroscopeDriftNoise
- GyroscopeNoise
- MagneticDisturbanceNoise
- MagnetometerNoise

**Update Error Estimate Covariance**

The error estimate covariance is a 12-by-12 matrix used to track the variability in the state.

The error estimate covariance matrix is updated as:

$$P_{12 \times 12}^+ = P_{12 \times 12}^- - (K_{12 \times 6})(H_{6 \times 12})(P_{12 \times 12}^-)$$

where  $K$  is the Kalman gain,  $H$  is the measurement matrix, and  $P^-$  is the error estimate covariance calculated during the previous iteration.

**Predict Error Estimate Covariance**

The error estimate covariance is a 12-by-12 matrix used to track the variability in the state. The *a priori* error estimate covariance,  $P^-$ , is set to the process noise covariance,  $Q$ , determined during the previous iteration.  $Q$  is calculated as a function of the *a posteriori* error estimate covariance,  $P^+$ . When calculating  $Q$ , it is assumed that the cross-correlation terms are negligible compared to the autocorrelation terms, and are set to zero:

$$Q =$$

$$\left[ \begin{array}{ccccc} P^+(1) + \kappa^2 P^+(40) + \beta + \eta & 0 & 0 & -\kappa(P^+(40) + \beta) & 0 \\ 0 & P^+(14) + \kappa^2 P^+(53) + \beta + \eta & 0 & 0 & -\kappa(P^+(53) + \beta) \\ 0 & 0 & P^+(27) + \kappa^2 P^+(66) + \beta + \eta & 0 & 0 \\ -\kappa(P^+(40) + \beta) & 0 & 0 & P^+(40) + \beta & 0 \\ 0 & -\kappa(P^+(53) + \beta) & 0 & 0 & P^+(53) + \beta \\ 0 & 0 & -\kappa(P^+(66) + \beta) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right]$$

where

- $P^+$  -- is the updated (*a posteriori*) error estimate covariance
- $\kappa$  -- DecimationFactor/SampleRate
- $\beta$  -- GyroscopeDriftNoise
- $\eta$  -- GyroscopeNoise
- $\nu$  -- LinearAccelerationDecayFactor
- $\xi$  -- LinearAccelerationNoise
- $\sigma$  -- MagneticDisturbanceDecayFactor
- $\gamma$  -- MagneticDisturbanceNoise

See section 10.1 of [1] for a derivation of the terms of the process error matrix.

### Kalman Gain

The Kalman gain matrix is a 12-by-6 matrix used to weight the innovation. In this algorithm, the innovation is interpreted as the error process,  $z$ .

The Kalman gain matrix is constructed as:

$$K_{12 \times 6} = (P_{12 \times 12}^-)^T (H_{6 \times 12})^T (S_{6 \times 6})^{-1}$$

where

- $P^-$  -- predicted error covariance
- $H$  -- observation model
- $S$  -- innovation covariance

### Update a Posteriori Error

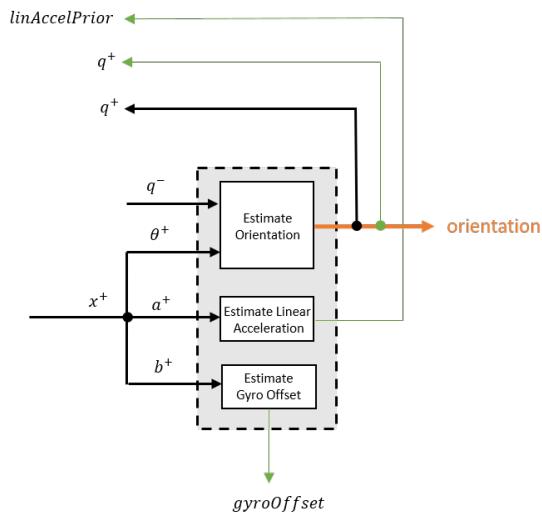
The *a posteriori* error estimate is determined by combining the Kalman gain matrix with the error in the gravity vector and magnetic vector estimations:

$$x_{12 \times 1} = (K_{12 \times 6})(z_{1 \times 6})^T$$

If magnetic jamming is detected in the current iteration, the magnetic vector error signal is ignored, and the *a posteriori* error estimate is calculated as:

$$x_{9 \times 1} = (K(1:9, 1:3)(z_g)^T$$

## Correct



### Estimate Orientation

The orientation estimate is updated by multiplying the previous estimation by the error:

$$q^+ = (q^-)(\theta^+)$$

### Estimate Linear Acceleration

The linear acceleration estimation is updated by decaying the linear acceleration estimation from the previous iteration and subtracting the error:

$$\text{linAccelPrior} = (\text{linAccelPrior}_{k-1})\nu - b^+$$

where

- $\nu$  -- LinearAccelerationDecayFactor

### Estimate Gyroscope Offset

The gyroscope offset estimation is updated by subtracting the gyroscope offset error from the gyroscope offset from the previous iteration:

$$\text{gyroOffset} = \text{gyroOffset}_{k-1} - a^+$$

### Compute Angular Velocity

To estimate angular velocity, the frame of `gyroReadings` are averaged and the gyroscope offset computed in the previous iteration is subtracted:

$$\text{angularVelocity}_{1 \times 3} = \frac{\sum \text{gyroReadings}_{N \times 3}}{N} - \text{gyroOffset}_{1 \times 3}$$

where  $N$  is the decimation factor specified by the `DecimationFactor` property.

The gyroscope offset estimation is initialized to zeros for the first iteration.

## Update Magnetic Vector

If magnetic jamming was not detected in the current iteration, the magnetic vector estimate,  $m$ , is updated using the *a posteriori* magnetic disturbance error and the *a posteriori* orientation.

The magnetic disturbance error is converted to the navigation frame:

$$mErrorNED_{1 \times 3} = \left( (rPost_{3 \times 3})^T (mError_{1 \times 3})^T \right)^T$$

The magnetic disturbance error in the navigation frame is subtracted from the previous magnetic vector estimate and then interpreted as inclination:

$$M = m - mErrorNED$$

$$inclination = \text{atan2}(M(3), M(1))$$

The inclination is converted to a constrained magnetic vector estimate for the next iteration:

$$m(1) = (\text{ExpectedMagneticFieldStrength})(\cos(inclination))$$

$$m(2) = 0$$

$$m(3) = (\text{ExpectedMagneticFieldStrength})(\sin(inclination))$$

`ExpectedMagneticFieldStrength` is a property of `ahrsfilter`.

## References

- [1] Open Source Sensor Fusion. <https://github.com/memsindustrygroup/Open-Source-Sensor-Fusion/tree/master/docs>
- [2] Roetenberg, D., H.J. Luinge, C.T.M. Baten, and P.H. Veltink. "Compensation of Magnetic Disturbances Improves Inertial and Magnetic Sensing of Human Body Segment Orientation." *IEEE Transactions on Neural Systems and Rehabilitation Engineering*. Vol. 13. Issue 3, 2005, pp. 395-405.

## 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

`ecompass` | `gpsSensor` | `imuSensor` | `imufilter` | `quaternion`

### Topics

"Determine Orientation Using Inertial Sensors"

### Introduced in R2018b

# tune

Tune `ahrsfilter` parameters to reduce estimation error

## Syntax

```
tune(filter,sensorData,groundTruth)
tune(___,config)
```

## Description

`tune(filter,sensorData,groundTruth)` adjusts the properties of the `ahrsfilter` filter object, `filter`, to reduce the root-mean-squared (RMS) quaternion distance error between the fused sensor data and the ground truth. The function uses the property values in the filter as the initial estimate for the optimization algorithm.

`tune(___,config)` specifies the tuning configuration based on a `tunerconfig` object, `config`.

## Examples

### Tune `ahrsfilter` to Improve Orientation Estimate

Load recorded sensor data and ground truth data.

```
ld = load('ahrsfilterTuneData.mat');
qTrue = ld.groundTruth.Orientation; % true orientation
```

Create an `ahrsfilter` object.

```
fuse = ahrsfilter;
```

Fuse the sensor data using the default, untuned filter.

```
qEstUntuned = fuse(ld.sensorData.Accelerometer, ...
    ld.sensorData.Gyroscope, ld.sensorData.Magnetometer);
```

Create a `tunerconfig` object. Tune the `ahrsfilter` object to improve the orientation estimation based on the configuration.

```
config = tunerconfig('ahrsfilter');
tune(fuse,ld.sensorData,ld.groundTruth,config);
```

Iteration	Parameter	Metric
1	AccelerometerNoise	0.1345
1	GyroscopeNoise	0.1342
1	MagnetometerNoise	0.1341
1	GyroscopeDriftNoise	0.1341
1	LinearAccelerationNoise	0.1332
1	MagneticDisturbanceNoise	0.1324
1	LinearAccelerationDecayFactor	0.1317
1	MagneticDisturbanceDecayFactor	0.1316

2	AccelerometerNoise	0.1316
2	GyroscopeNoise	0.1312
2	MagnetometerNoise	0.1311
2	GyroscopeDriftNoise	0.1311
2	LinearAccelerationNoise	0.1300
2	MagneticDisturbanceNoise	0.1292
2	LinearAccelerationDecayFactor	0.1285
2	MagneticDisturbanceDecayFactor	0.1285
3	AccelerometerNoise	0.1285
3	GyroscopeNoise	0.1280
3	MagnetometerNoise	0.1279
3	GyroscopeDriftNoise	0.1279
3	LinearAccelerationNoise	0.1267
3	MagneticDisturbanceNoise	0.1258
3	LinearAccelerationDecayFactor	0.1253
3	MagneticDisturbanceDecayFactor	0.1253
4	AccelerometerNoise	0.1252
4	GyroscopeNoise	0.1247
4	MagnetometerNoise	0.1246
4	GyroscopeDriftNoise	0.1246
4	LinearAccelerationNoise	0.1233
4	MagneticDisturbanceNoise	0.1224
4	LinearAccelerationDecayFactor	0.1220
4	MagneticDisturbanceDecayFactor	0.1220
5	AccelerometerNoise	0.1220
5	GyroscopeNoise	0.1213
5	MagnetometerNoise	0.1212
5	GyroscopeDriftNoise	0.1212
5	LinearAccelerationNoise	0.1200
5	MagneticDisturbanceNoise	0.1190
5	LinearAccelerationDecayFactor	0.1187
5	MagneticDisturbanceDecayFactor	0.1187
6	AccelerometerNoise	0.1187
6	GyroscopeNoise	0.1180
6	MagnetometerNoise	0.1178
6	GyroscopeDriftNoise	0.1178
6	LinearAccelerationNoise	0.1167
6	MagneticDisturbanceNoise	0.1156
6	LinearAccelerationDecayFactor	0.1155
6	MagneticDisturbanceDecayFactor	0.1155
7	AccelerometerNoise	0.1155
7	GyroscopeNoise	0.1147
7	MagnetometerNoise	0.1145
7	GyroscopeDriftNoise	0.1145
7	LinearAccelerationNoise	0.1137
7	MagneticDisturbanceNoise	0.1126
7	LinearAccelerationDecayFactor	0.1125
7	MagneticDisturbanceDecayFactor	0.1125
8	AccelerometerNoise	0.1125
8	GyroscopeNoise	0.1117
8	MagnetometerNoise	0.1116
8	GyroscopeDriftNoise	0.1116
8	LinearAccelerationNoise	0.1112
8	MagneticDisturbanceNoise	0.1100
8	LinearAccelerationDecayFactor	0.1099
8	MagneticDisturbanceDecayFactor	0.1099
9	AccelerometerNoise	0.1099
9	GyroscopeNoise	0.1091

9	MagnetometerNoise	0.1090
9	GyroscopeDriftNoise	0.1090
9	LinearAccelerationNoise	0.1090
9	MagneticDisturbanceNoise	0.1076
9	LinearAccelerationDecayFactor	0.1075
9	MagneticDisturbanceDecayFactor	0.1075
10	AccelerometerNoise	0.1075
10	GyroscopeNoise	0.1066
10	MagnetometerNoise	0.1064
10	GyroscopeDriftNoise	0.1064
10	LinearAccelerationNoise	0.1064
10	MagneticDisturbanceNoise	0.1049
10	LinearAccelerationDecayFactor	0.1047
10	MagneticDisturbanceDecayFactor	0.1047
11	AccelerometerNoise	0.1047
11	GyroscopeNoise	0.1038
11	MagnetometerNoise	0.1036
11	GyroscopeDriftNoise	0.1036
11	LinearAccelerationNoise	0.1036
11	MagneticDisturbanceNoise	0.1016
11	LinearAccelerationDecayFactor	0.1014
11	MagneticDisturbanceDecayFactor	0.1014
12	AccelerometerNoise	0.1014
12	GyroscopeNoise	0.1005
12	MagnetometerNoise	0.1002
12	GyroscopeDriftNoise	0.1002
12	LinearAccelerationNoise	0.1002
12	MagneticDisturbanceNoise	0.0978

Fuse the sensor data using the tuned filter.

```
qEstTuned = fuse(ld.sensorData.Accelerometer, ...
    ld.sensorData.Gyroscope, ld.sensorData.Magnetometer);
```

Compare the tuned and untuned RMS error performances.

```
dUntuned = rad2deg(dist(qEstUntuned, qTrue));
dTuned = rad2deg(dist(qEstTuned, qTrue));
rmsUntuned = sqrt(mean(dUntuned.^2))

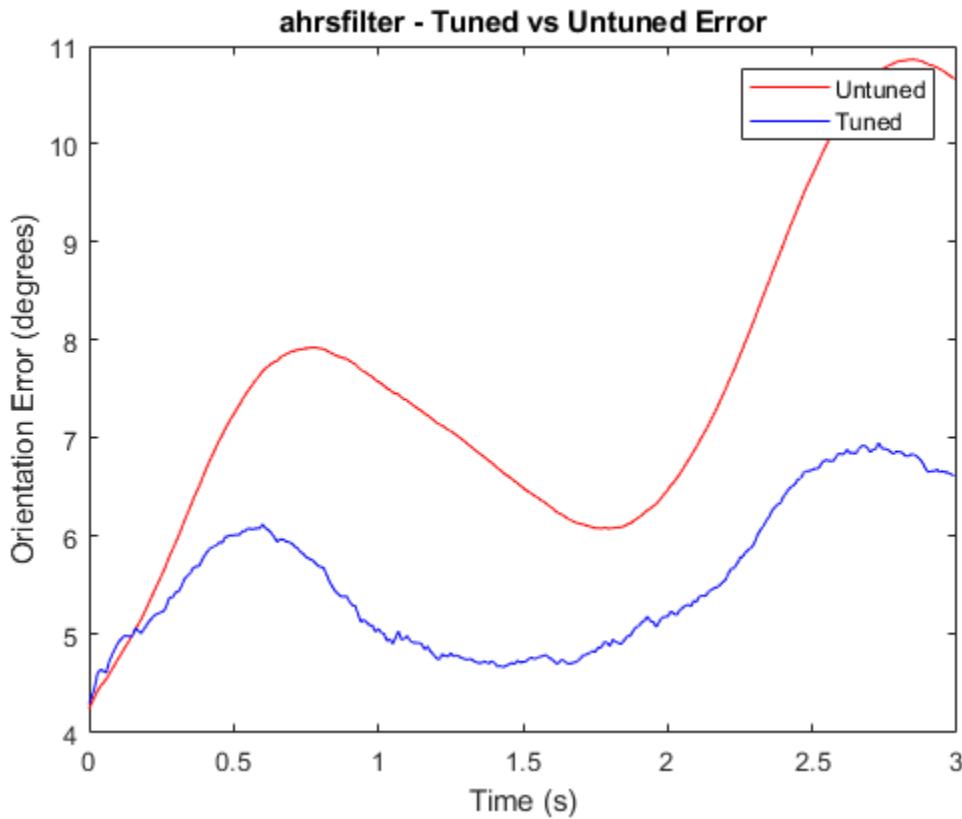
rmsUntuned = 7.7088

rmsTuned = sqrt(mean(dTuned.^2))

rmsTuned = 5.6033
```

Visualize the errors with respect to time.

```
N = numel(dUntuned);
t = (0:N-1)./ fuse.SampleRate;
plot(t, dUntuned, 'r', t, dTuned, 'b');
legend('Untuned', 'Tuned');
title('ahrsfilter - Tuned vs Untuned Error')
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



## Input Arguments

### **filter — Filter object**

ahrsfilter object

Filter object, specified as an ahrsfilter object.

### **sensorData — Sensor data**

table

Sensor data, specified as a table. In each row, the sensor data is specified as:

- **Accelerometer** — Accelerometer data, specified as a 1-by-3 vector of scalars in  $\text{m}^2/\text{s}$ .
- **Gyroscope** — Gyroscope data, specified as a 1-by-3 vector of scalars in  $\text{rad}/\text{s}$ .
- **Magnetometer** — Magnetometer data, specified as a 1-by-3 vector of scalars in  $\mu\text{T}$ .

If you set the Cost property of the tuner configuration input, config, to Custom, then you can use other data types for the sensorData input based on your choice.

### **groundTruth — Ground truth data**

timetable

Ground truth data, specified as a table. The table has only one column of Orientation data. In each row, the orientation is specified as a quaternion object or a 3-by-3 rotation matrix.

---

The function processes each row of the `sensorData` and `groundTruth` tables sequentially to calculate the state estimate and RMS error from the ground truth. Each row of the `sensorData` and the `groundTruth` tables must correspond to each other.

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `groundTruth` input based on your choice.

**config — Tuner configuration**

`tunerconfig` object

Tuner configuration, specified as a `tunerconfig` object.

## References

- [1] Abbeel, P., Coates, A., Montemerlo, M., Ng, A.Y. and Thrun, S. Discriminative Training of Kalman Filters. In *Robotics: Science and systems*, Vol. 2, pp. 1, 2005.

## See Also

**Introduced in R2020b**

# complementaryFilter

Orientation estimation from a complementary filter

## Description

The `complementaryFilter` System object fuses accelerometer, gyroscope, and magnetometer sensor data to estimate device orientation and angular velocity.

To estimate orientation using this object:

- 1 Create the `complementaryFilter` 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

```
FUSE = complementaryFilter
FUSE = complementaryFilter('ReferenceFrame',RF)
FUSE = complementaryFilter(____,Name,Value)
```

### Description

`FUSE = complementaryFilter` returns a `complementaryFilter` System object, `FUSE`, for sensor fusion of accelerometer, gyroscope, and magnetometer data to estimate device orientation and angular velocity.

`FUSE = complementaryFilter('ReferenceFrame',RF)` returns a `complementaryFilter` System object that fuses accelerometer, gyroscope, and magnetometer data to estimate device orientation relative to the reference frame `RF`. Specify `RF` as '`NED`' (North-East-Down) or '`ENU`' (East-North-Up). The default value is '`NED`'.

`FUSE = complementaryFilter(____,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

### 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 — Input sample rate of sensor data (Hz)**

100 (default) | positive scalar

Input sample rate of the sensor data in Hz, specified as a positive scalar.

**Tunable:** No

Data Types: single | double

**AccelerometerGain — Accelerometer gain**

0.01 (default) | real scalar in [0, 1]

Accelerometer gain, specified as a real scalar in the range of [0, 1]. The gain determines how much the accelerometer measurement is trusted over the gyroscope measurement for orientation estimation. This property is tunable.

Data Types: single | double

**MagnetometerGain — Magnetometer gain**

0.01 (default) | real scalar in [0, 1]

Magnetometer gain, specified as a real scalar in the range of [0, 1]. The gain determines how much the magnetometer measurement is trusted over the gyroscope measurement for orientation estimation. This property is tunable.

Data Types: single | double

**HasMagnetometer — Enable magnetometer input**

true (default) | false

Enable magnetometer input, specified as true or false.

Data Types: logical

**OrientationFormat — Output orientation format**

'quaternion' (default) | 'Rotation matrix'

Output orientation format, specified as 'quaternion' or 'Rotation matrix'. The size of the output depends on the output orientation format:

- 'quaternion' -- Output is an  $N$ -by-1 quaternion.
- 'Rotation matrix' -- Output is a 3-by-3-by- $N$  rotation matrix.

$N$  is the number of samples.

Data Types: char | string

## Usage

### Syntax

```
[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings,magReadings)
[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings)
```

## Description

`[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings,magReadings)`  
fuses accelerometer, gyroscope, and magnetometer data to compute orientation and angular velocity.  
To use this syntax, set the `HasMagnetometer` property as `true`.

`[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings)` fuses  
accelerometer and gyroscope data to compute orientation and angular velocity. To use this syntax, set  
the `HasMagnetometer` property as `false`.

## Input Arguments

**accelReadings — Accelerometer readings in sensor body coordinate system (m/s<sup>2</sup>)**  
*N*-by-3 matrix

Accelerometer readings in the sensor body coordinate system in m/s<sup>2</sup>, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `accelReadings` represent the [x y z] measurements. Accelerometer readings are assumed to correspond to the sample rate specified by the `SampleRate` property. In the filter, the gravity constant *g* is assumed to be 9.81 m/s<sup>2</sup>.

Data Types: `single` | `double`

**gyroReadings — Gyroscope readings in sensor body coordinate system (rad/s)**  
*N*-by-3 matrix

Gyroscope readings in the sensor body coordinate system in rad/s, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `gyroReadings` represent the [x y z] measurements. Gyroscope readings are assumed to correspond to the sample rate specified by the `SampleRate` property.

Data Types: `single` | `double`

**magReadings — Magnetometer readings in sensor body coordinate system (μT)**  
*N*-by-3 matrix

Magnetometer readings in the sensor body coordinate system in μT, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of `magReadings` represent the [x y z] measurements. Magnetometer readings are assumed to correspond to the sample rate specified by the `SampleRate` property.

Data Types: `single` | `double`

## Output Arguments

**orientation — Orientation that rotates quantities from local navigation coordinate system to sensor body coordinate system**

*N*-by-1 array of quaternions (default) | 3-by-3-by-*N* array

Orientation that rotates quantities from the local navigation coordinate system to the body coordinate system, returned as quaternions or an array. The size and type of `orientation` depends on whether the `OrientationFormat` property is set to '`quaternion`' or '`Rotation matrix`':

- '`quaternion`' -- the output is an *N*-by-1 vector of quaternions, where *N* is the number of samples.
- '`Rotation matrix`' -- the output is a 3-by-3-by-*N* array of rotation matrices, where *N* is the number of samples.

Data Types: quaternion | single | double

**angularVelocity — Angular velocity in sensor body coordinate system (rad/s)**  
*N*-by-3 array (default)

Angular velocity expressed in the sensor body coordinate system in rad/s, returned as an *N*-by-3 array, where *N* is the number of samples.

Data Types: single | 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
clone	Create duplicate System object
isLocked	Determine if System object is in use

## Examples

### Estimate Orientation from Recorded IMU Data

Load the `rpy_9axis` file, which contains recorded accelerometer, gyroscope, and magnetometer sensor data from a device oscillating in pitch (around *y*-axis), then yaw (around *z*-axis), and then roll (around *x*-axis). The file also contains the sample rate of the recording.

```
ld = load('rpy_9axis.mat');
accel = ld.sensorData.Acceleration;
gyro = ld.sensorData.AngularVelocity;
mag = ld.sensorData.MagneticField;
```

Create a complementary filter object with sample rate equal to the frequency of the data.

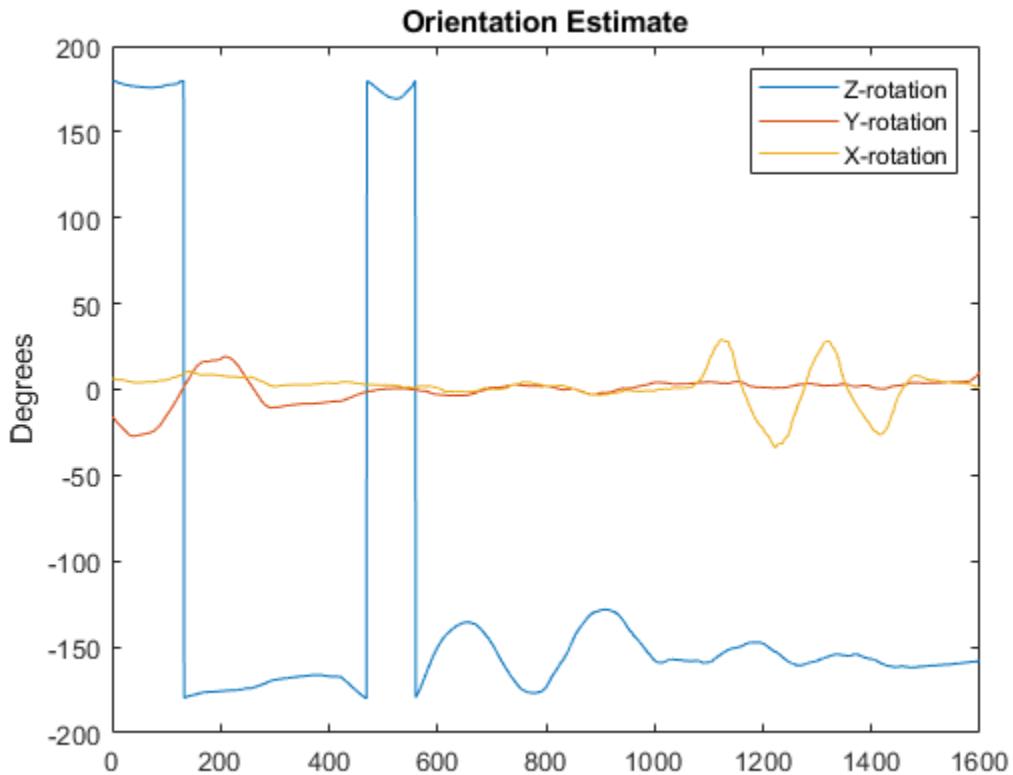
```
Fs = ld.Fs; % Hz
fuse = complementaryFilter('SampleRate', Fs);
```

Fuse accelerometer, gyroscope, and magnetometer data using the filter.

```
q = fuse(accel, gyro, mag);
```

Visualize the results.

```
plot(eulerd(q, 'ZYX', 'frame'));
title('Orientation Estimate');
legend('Z-rotation', 'Y-rotation', 'X-rotation');
ylabel('Degrees');
```



## References

- [1] Valenti, R., I. Dryanovski, and J. Xiao. "Keeping a good attitude: A quaternion-based orientation filter for IMUs and MARGs." *Sensors*. Vol. 15, Number 8, 2015, pp. 19302-19330.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[ahrsfilter](#) | [imufilter](#)

**Introduced in R2019b**

# imufilter

Orientation from accelerometer and gyroscope readings

## Description

The `imufilter` System object fuses accelerometer and gyroscope sensor data to estimate device orientation.

To estimate device orientation:

- 1 Create the `imufilter` 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

```
FUSE = imufilter
FUSE = imufilter('ReferenceFrame',RF)
FUSE = imufilter(___,Name,Value)
```

### Description

`FUSE = imufilter` returns an indirect Kalman filter System object, `FUSE`, for fusion of accelerometer and gyroscope data to estimate device orientation. The filter uses a nine-element state vector to track error in the orientation estimate, the gyroscope bias estimate, and the linear acceleration estimate.

`FUSE = imufilter('ReferenceFrame',RF)` returns an `imufilter` filter System object that fuses accelerometer and gyroscope data to estimate device orientation relative to the reference frame `RF`. Specify `RF` as '`NED`' (North-East-Down) or '`ENU`' (East-North-Up). The default value is '`NED`'.

`FUSE = imufilter(___,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

Example: `FUSE = imufilter('SampleRate',200,'GyroscopeNoise',1e-6)` creates a System object, `FUSE`, with a 200 Hz sample rate and gyroscope noise set to  $1e-6$  radians per second squared.

## 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 input sensor data (Hz)**

100 (default) | positive finite scalar

Sample rate of the input sensor data in Hz, specified as a positive finite scalar.

**Tunable:** No

Data Types: single | double | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**DecimationFactor — Decimation factor**

1 (default) | positive integer scalar

Decimation factor by which to reduce the sample rate of the input sensor data, specified as a positive integer scalar.

The number of rows of the inputs, `accelReadings` and `gyroReadings`, must be a multiple of the decimation factor.

**Tunable:** No

Data Types: single | double | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**AccelerometerNoise — Variance of accelerometer signal noise ((m/s<sup>2</sup>)<sup>2</sup>)**

0.00019247 (default) | positive real scalar

Variance of accelerometer signal noise in (m/s<sup>2</sup>)<sup>2</sup>, specified as a positive real scalar.

**Tunable:** Yes

Data Types: single | double | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**GyroscopeNoise — Variance of gyroscope signal noise ((rad/s)<sup>2</sup>)**

9.1385e-5 (default) | positive real scalar

Variance of gyroscope signal noise in (rad/s)<sup>2</sup>, specified as a positive real scalar.

**Tunable:** Yes

Data Types: single | double | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**GyroscopeDriftNoise — Variance of gyroscope offset drift ((rad/s)<sup>2</sup>)**

3.0462e-13 (default) | positive real scalar

Variance of gyroscope offset drift in (rad/s)<sup>2</sup>, specified as a positive real scalar.

**Tunable:** Yes

Data Types: single | double | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**LinearAccelerationNoise — Variance of linear acceleration noise ((m/s<sup>2</sup>)<sup>2</sup>)**

0.0096236 (default) | positive real scalar

Variance of linear acceleration noise in (m/s<sup>2</sup>)<sup>2</sup>, specified as a positive real scalar. Linear acceleration is modeled as a lowpass filtered white noise process.

**Tunable:** Yes

Data Types: single | double | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**LinearAccelerationDecayFactor — Decay factor for linear acceleration drift**

0.5 (default) | scalar in the range [0,1]

Decay factor for linear acceleration drift, specified as a scalar in the range [0,1]. If linear acceleration is changing quickly, set `LinearAccelerationDecayFactor` to a lower value. If linear acceleration changes slowly, set `LinearAccelerationDecayFactor` to a higher value. Linear acceleration drift is modeled as a lowpass-filtered white noise process.

**Tunable:** Yes

Data Types: single | double | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**InitialProcessNoise — Covariance matrix for process noise**

9-by-9 matrix

Covariance matrix for process noise, specified as a 9-by-9 matrix. The default is:

Columns 1 through 6

0.000006092348396	0	0	0	0	0	0
0	0.000006092348396	0	0	0	0	0
0	0	0.000006092348396	0	0	0	0
0	0	0	0.000076154354947	0	0	0
0	0	0	0	0.000076154354947	0	0
0	0	0	0	0	0.000076154354947	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Columns 7 through 9

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0.009623610000000	0	0	0
0	0.009623610000000	0	0
0	0	0.009623610000000	0

The initial process covariance matrix accounts for the error in the process model.

Data Types: single | double | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64

**OrientationFormat — Output orientation format**

'quaternion' (default) | 'Rotation matrix'

Output orientation format, specified as 'quaternion' or 'Rotation matrix'. The size of the output depends on the input size,  $N$ , and the output orientation format:

- 'quaternion' -- Output is an  $N$ -by-1 quaternion.
- 'Rotation matrix' -- Output is a 3-by-3-by- $N$  rotation matrix.

Data Types: char | string

## Usage

### Syntax

```
[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings)
```

### Description

[orientation,angularVelocity] = FUSE(accelReadings,gyroReadings) fuses accelerometer and gyroscope readings to compute orientation and angular velocity measurements. The algorithm assumes that the device is stationary before the first call.

### Input Arguments

**accelReadings — Accelerometer readings in sensor body coordinate system (m/s<sup>2</sup>)**

*N*-by-3 matrix

Accelerometer readings in the sensor body coordinate system in m/s<sup>2</sup>, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of accelReadings represent the [x y z] measurements. Accelerometer readings are assumed to correspond to the sample rate specified by the SampleRate property.

Data Types: single | double

**gyroReadings — Gyroscope readings in sensor body coordinate system (rad/s)**

*N*-by-3 matrix

Gyroscope readings in the sensor body coordinate system in rad/s, specified as an *N*-by-3 matrix. *N* is the number of samples, and the three columns of gyroReadings represent the [x y z] measurements. Gyroscope readings are assumed to correspond to the sample rate specified by the SampleRate property.

Data Types: single | double

### Output Arguments

**orientation — Orientation that rotates quantities from global coordinate system to sensor body coordinate system**

*M*-by-1 vector of quaternions (default) | 3-by-3-by-*M* array

Orientation that can rotate quantities from a global coordinate system to a body coordinate system, returned as quaternions or an array. The size and type of orientation depends on whether the OrientationFormat property is set to 'quaternion' or 'Rotation matrix':

- 'quaternion' -- The output is an *M*-by-1 vector of quaternions, with the same underlying data type as the inputs.
- 'Rotation matrix' -- The output is a 3-by-3-by-*M* array of rotation matrices the same data type as the inputs.

The number of input samples, *N*, and the DecimationFactor property determine *M*.

You can use orientation in a rotateframe function to rotate quantities from a global coordinate system to a sensor body coordinate system.

Data Types: quaternion | single | double

**angularVelocity — Angular velocity in sensor body coordinate system (rad/s)**  
*M*-by-3 array (default)

Angular velocity with gyroscope bias removed in the sensor body coordinate system in rad/s, returned as an *M*-by-3 array. The number of input samples, *N*, and the DecimationFactor property determine *M*.

Data Types: single | 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 imufilter

tune Tune imufilter parameters to reduce estimation error

### 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

### Estimate Orientation from IMU data

Load the *rpy\_9axis* file, which contains recorded accelerometer, gyroscope, and magnetometer sensor data from a device oscillating in pitch (around *y*-axis), then yaw (around *z*-axis), and then roll (around *x*-axis). The file also contains the sample rate of the recording.

```
load 'rpy_9axis.mat' sensorData Fs
accelerometerReadings = sensorData.Acceleration;
gyroscopeReadings = sensorData.AngularVelocity;
```

Create an *imufilter* System object™ with sample rate set to the sample rate of the sensor data. Specify a decimation factor of two to reduce the computational cost of the algorithm.

```
decim = 2;
fuse = imufilter('SampleRate',Fs,'DecimationFactor',decim);
```

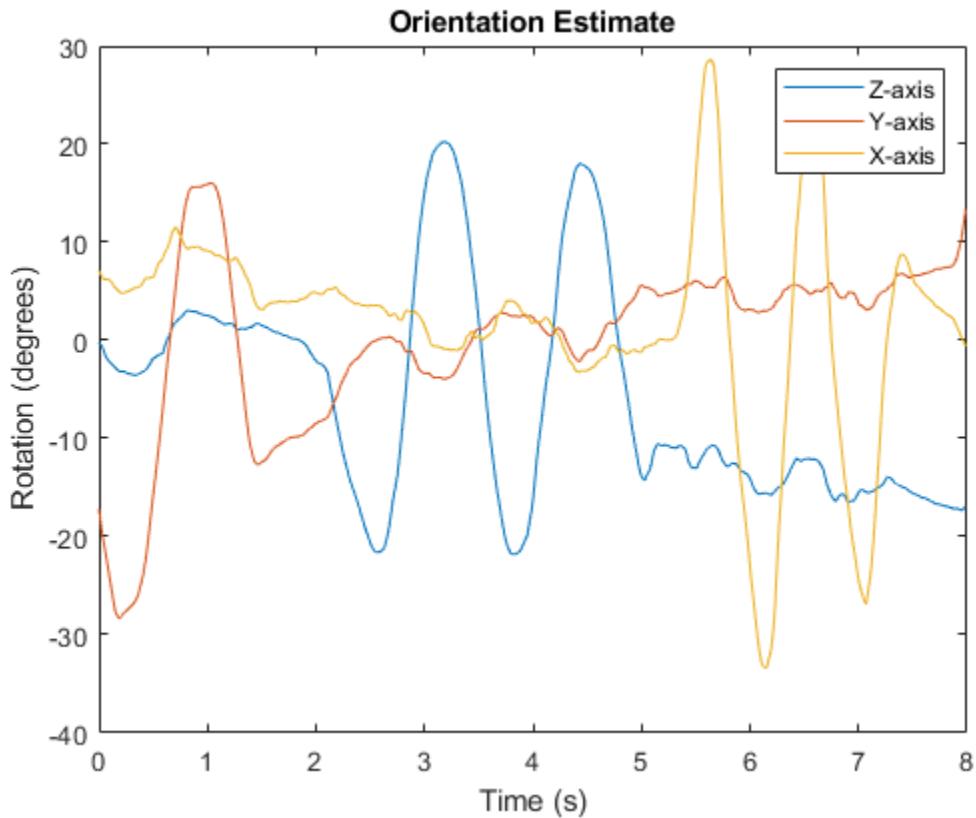
Pass the accelerometer readings and gyroscope readings to the *imufilter* object, *fuse*, to output an estimate of the sensor body orientation over time. By default, the orientation is output as a vector of quaternions.

```
q = fuse(accelerometerReadings,gyroscopeReadings);
```

Orientation is defined by the angular displacement required to rotate a parent coordinate system to a child coordinate system. Plot the orientation in Euler angles in degrees over time.

`imufilter` fusion correctly estimates the change in orientation from an assumed north-facing initial orientation. However, the device's x-axis was pointing southward when recorded. To correctly estimate the orientation relative to the true initial orientation or relative to NED, use `ahrsfilter`.

```
time = (0:decim:size(accelerometerReadings,1)-1)/Fs;  
  
plot(time,eulerd(q,'ZYX','frame'))  
title('Orientation Estimate')  
legend('Z-axis', 'Y-axis', 'X-axis')  
xlabel('Time (s)')  
ylabel('Rotation (degrees)')
```



### Model Tilt Using Gyroscope and Accelerometer Readings

Model a tilting IMU that contains an accelerometer and gyroscope using the `imuSensor` System object™. Use ideal and realistic models to compare the results of orientation tracking using the `imufilter` System object.

Load a struct describing ground-truth motion and a sample rate. The motion struct describes sequential rotations:

- 1 yaw: 120 degrees over two seconds
- 2 pitch: 60 degrees over one second

- 3** roll: 30 degrees over one-half second
- 4** roll: -30 degrees over one-half second
- 5** pitch: -60 degrees over one second
- 6** yaw: -120 degrees over two seconds

In the last stage, the motion struct combines the 1st, 2nd, and 3rd rotations into a single-axis rotation. The acceleration, angular velocity, and orientation are defined in the local NED coordinate system.

```
load y120p60r30.mat motion fs
accNED = motion.Acceleration;
angVelNED = motion.AngularVelocity;
orientationNED = motion.Orientation;

numSamples = size(motion.Orientation,1);
t = (0:(numSamples-1)).'/fs;
```

Create an ideal IMU sensor object and a default IMU filter object.

```
IMU = imuSensor('accel-gyro','SampleRate',fs);
aFilter = imufilter('SampleRate',fs);
```

In a loop:

- 1** Simulate IMU output by feeding the ground-truth motion to the IMU sensor object.
- 2** Filter the IMU output using the default IMU filter object.

```
orientation = zeros(numSamples,1,'quaternion');
for i = 1:numSamples

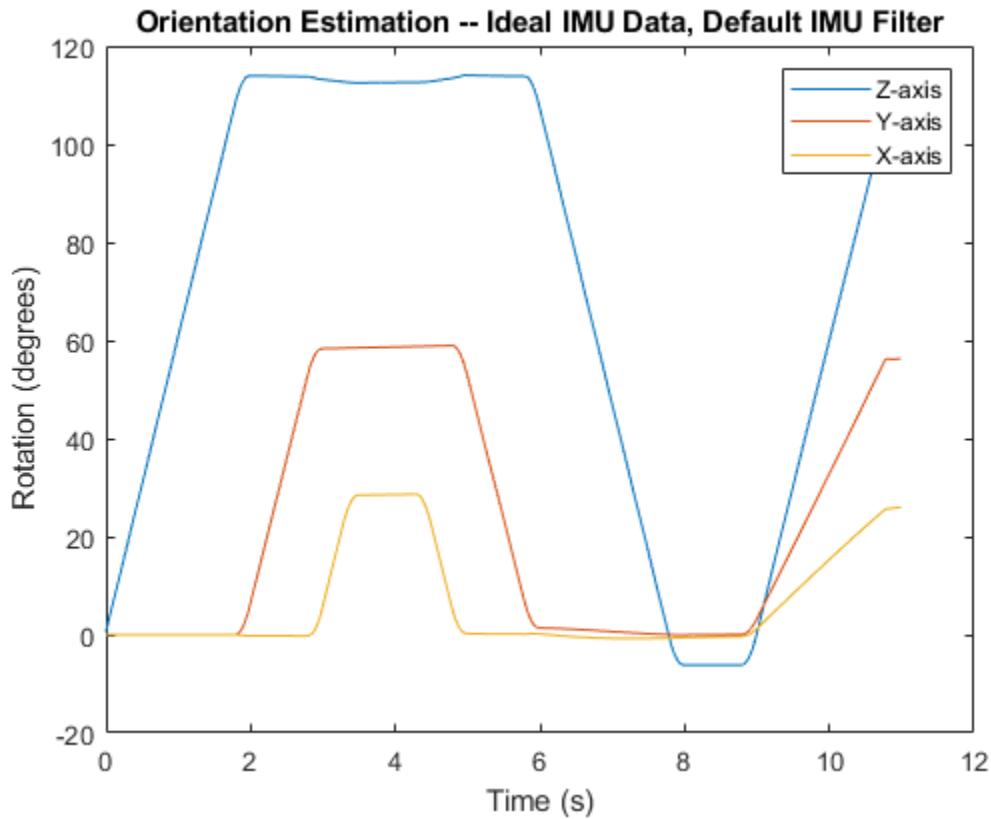
    [accelBody,gyroBody] = IMU(accNED(i,:),angVelNED(i,:),orientationNED(i,:));

    orientation(i) = aFilter(accelBody,gyroBody);

end
release(aFilter)
```

Plot the orientation over time.

```
figure(1)
plot(t,eulerd(orientation,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Ideal IMU Data, Default IMU Filter')
legend('Z-axis','Y-axis','X-axis')
```



Modify properties of your `imuSensor` to model real-world sensors. Run the loop again and plot the orientation estimate over time.

```

IMU.Accelerometer = accelparams( ...
    'MeasurementRange',19.62, ...
    'Resolution',0.00059875, ...
    'ConstantBias',0.4905, ...
    'AxesMisalignment',2, ...
    'NoiseDensity',0.003924, ...
    'BiasInstability',0, ...
    'TemperatureBias',[0.34335 0.34335 0.5886], ...
    'TemperatureScaleFactor',0.02);
IMU.Gyroscope = gyroparams( ...
    'MeasurementRange',4.3633, ...
    'Resolution',0.00013323, ...
    'AxesMisalignment',2, ...
    'NoiseDensity',8.7266e-05, ...
    'TemperatureBias',0.34907, ...
    'TemperatureScaleFactor',0.02, ...
    'AccelerationBias',0.00017809, ...
    'ConstantBias',[0.3491,0.5,0]);

orientationDefault = zeros(numSamples,1,'quaternion');
for i = 1:numSamples
    [accelBody,gyroBody] = IMU(accNED(i,:),angVelNED(i,:),orientationNED(i,:));

```

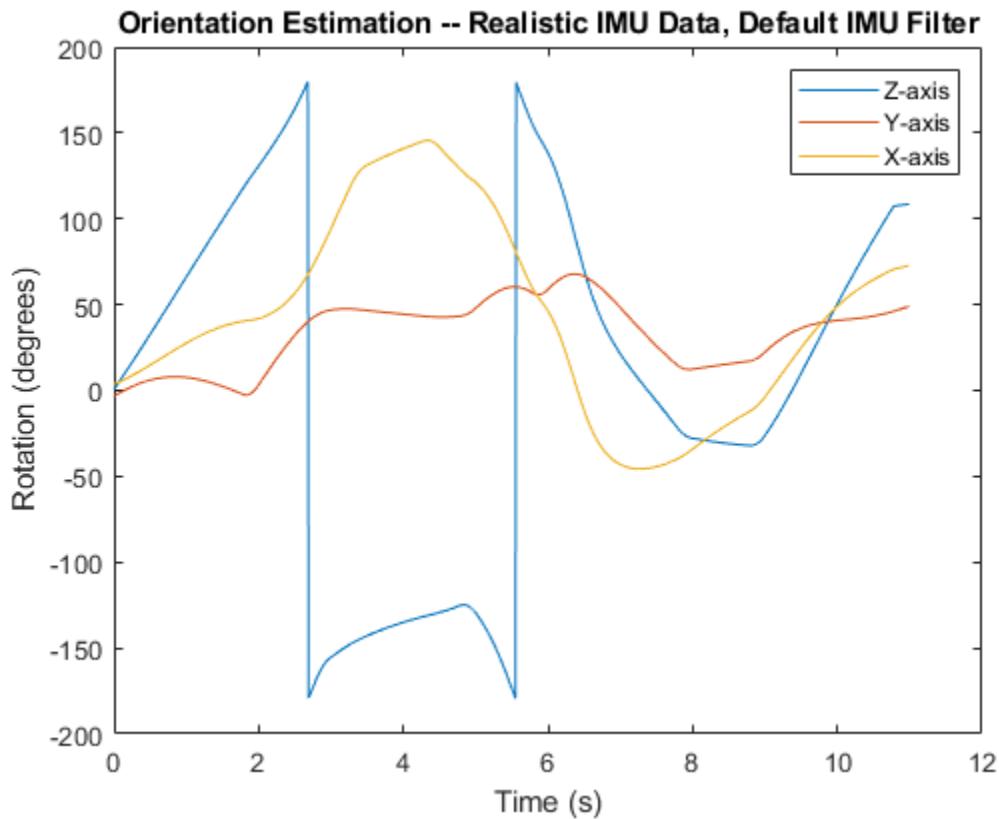
```

orientationDefault(i) = aFilter(accelBody,gyroBody);

end
release(aFilter)

figure(2)
plot(t,eulerd(orientationDefault,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Realistic IMU Data, Default IMU Filter')
legend('Z-axis','Y-axis','X-axis')

```



The ability of the `imufilter` to track the ground-truth data is significantly reduced when modeling a realistic IMU. To improve performance, modify properties of your `imufilter` object. These values were determined empirically. Run the loop again and plot the orientation estimate over time.

```

aFilter.GyroscopeNoise      = 7.6154e-7;
aFilter.AccelerometerNoise = 0.0015398;
aFilter.GyroscopeDriftNoise = 3.0462e-12;
aFilter.LinearAccelerationNoise = 0.00096236;
aFilter.InitialProcessNoise = aFilter.InitialProcessNoise*10;

orientationNondefault = zeros(numSamples,1,'quaternion');
for i = 1:numSamples
    [accelBody,gyroBody] = IMU(accNED(i,:),angVelNED(i,:),orientationNED(i,:));
    orientationNondefault(i) = aFilter(accelBody,gyroBody);

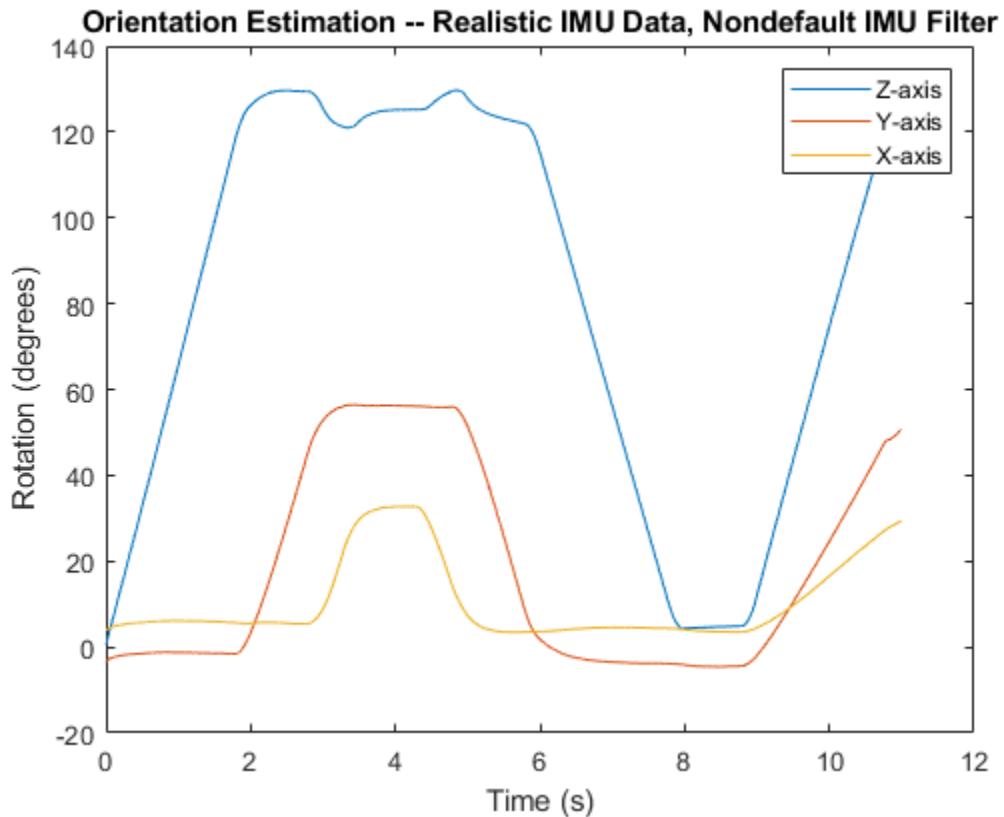
```

```

end
release(aFilter)

figure(3)
plot(t,eulerd(orientationNondefault,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Realistic IMU Data, Nondefault IMU Filter')
legend('Z-axis','Y-axis','X-axis')

```



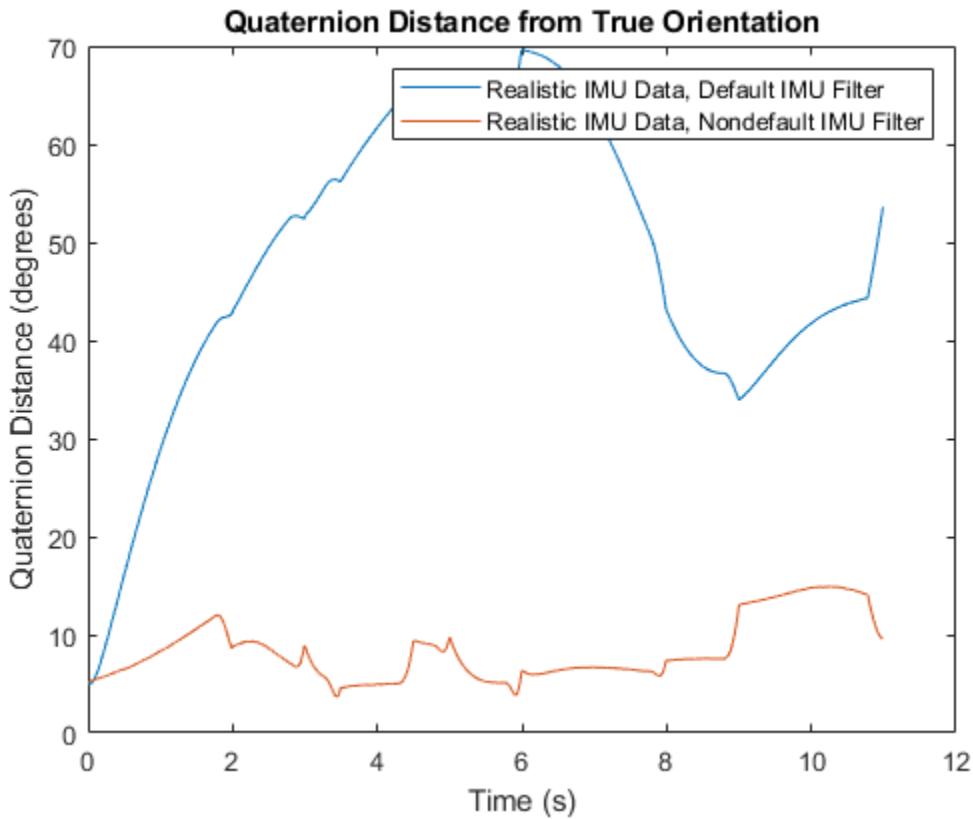
To quantify the improved performance of the modified `imufilter`, plot the quaternion distance between the ground-truth motion and the orientation as returned by the `imufilter` with default and nondefault properties.

```

qDistDefault = rad2deg(dist(orientationNED,orientationDefault));
qDistNondefault = rad2deg(dist(orientationNED,orientationNondefault));

figure(4)
plot(t,[qDistDefault,qDistNondefault])
title('Quaternion Distance from True Orientation')
legend('Realistic IMU Data, Default IMU Filter', ...
       'Realistic IMU Data, Nondefault IMU Filter')
xlabel('Time (s)')
ylabel('Quaternion Distance (degrees)')

```



### Remove Bias from Angular Velocity Measurement

This example shows how to remove gyroscope bias from an IMU using `imufilter`.

Use `kinematicTrajectory` to create a trajectory with two parts. The first part has a constant angular velocity about the y- and z-axes. The second part has a varying angular velocity in all three axes.

```

duration = 60*8;
fs = 20;
numSamples = duration * fs;
rng('default') % Seed the RNG to reproduce noisy sensor measurements.

initialAngVel = [0,0.5,0.25];
finalAngVel = [-0.2,0.6,0.5];
constantAngVel = repmat(initialAngVel,floor(numSamples/2),1);
varyingAngVel = [linspace(initialAngVel(1), finalAngVel(1), ceil(numSamples/2)).',...
    linspace(initialAngVel(2), finalAngVel(2), ceil(numSamples/2)).',...
    linspace(initialAngVel(3), finalAngVel(3), ceil(numSamples/2)).'];

angVelBody = [constantAngVel; varyingAngVel];
accBody = zeros(numSamples,3);

traj = kinematicTrajectory('SampleRate',fs);

[~,qNED,~,accNED,angVelNED] = traj(accBody,angVelBody);

```

Create an `imuSensor` System object™, `IMU`, with a nonideal gyroscope. Call `IMU` with the ground-truth acceleration, angular velocity, and orientation.

```
IMU = imuSensor('accel-gyro', ...
    'Gyroscope',gyroparams('RandomWalk',0.003,'ConstantBias',0.3), ...
    'SampleRate',fs);

[accelReadings, gyroReadingsBody] = IMU(accNED,angVelNED,qNED);
```

Create an `imufilter` System object, `fuse`. Call `fuse` with the modeled accelerometer readings and gyroscope readings.

```
fuse = imufilter('SampleRate',fs, 'GyroscopeDriftNoise', 1e-6);

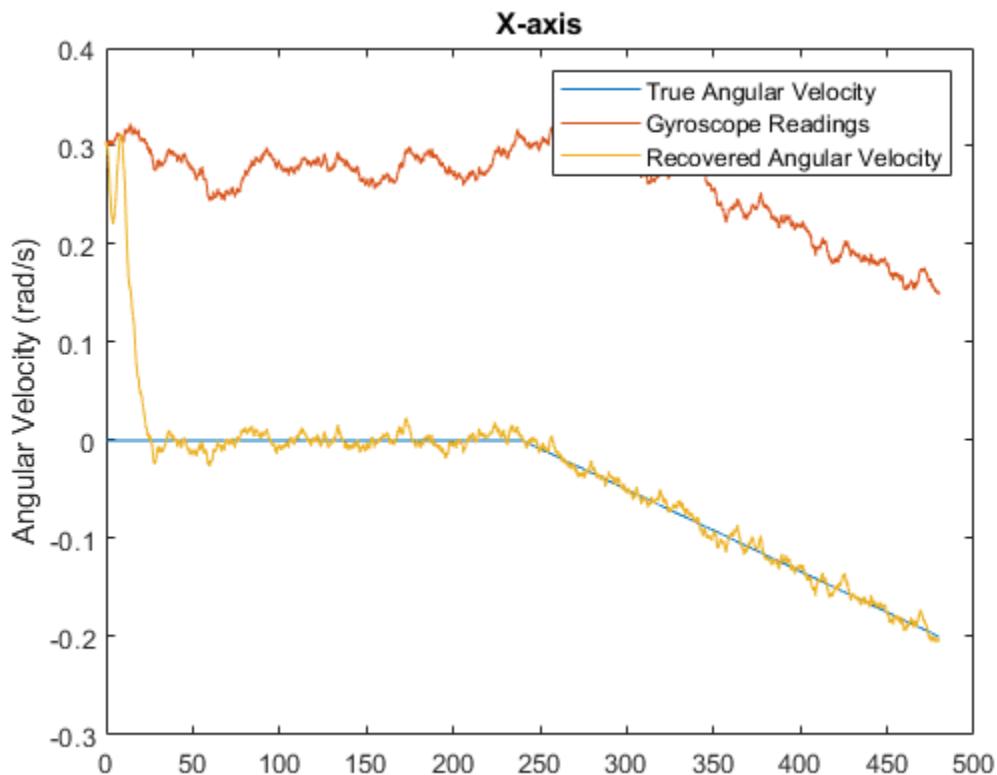
[~,angVelBodyRecovered] = fuse(accelReadings,gyroReadingsBody);
```

Plot the ground-truth angular velocity, the gyroscope readings, and the recovered angular velocity for each axis.

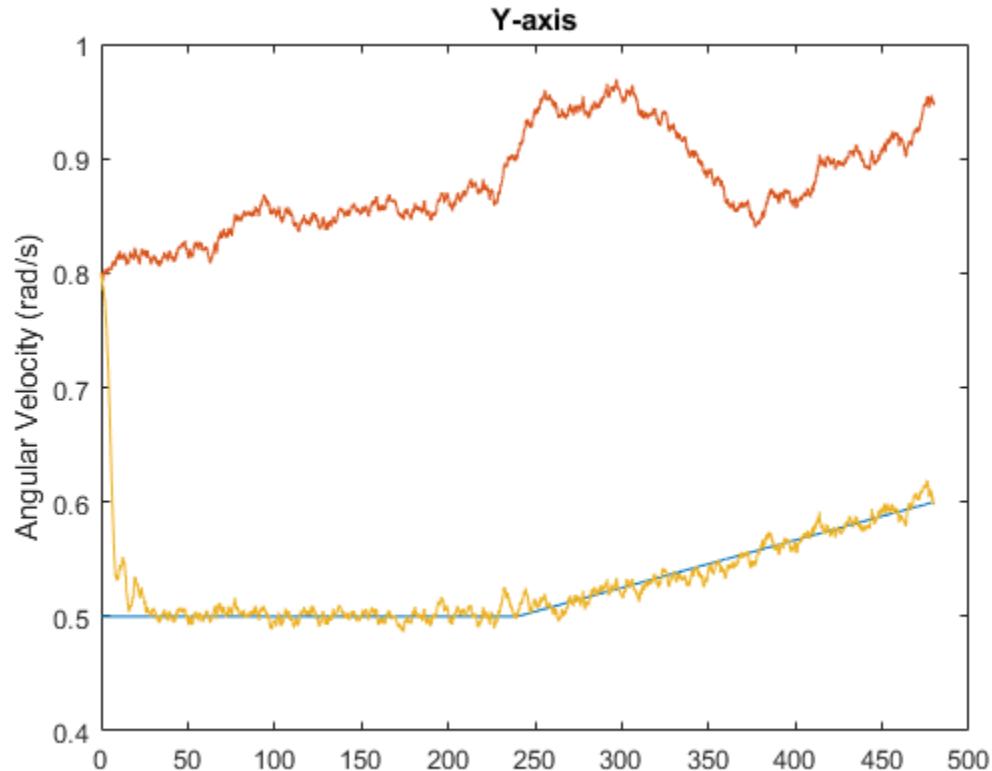
The angular velocity returned from the `imufilter` compensates for the effect of the gyroscope bias over time and converges to the true angular velocity.

```
time = (0:numSamples-1)/fs;

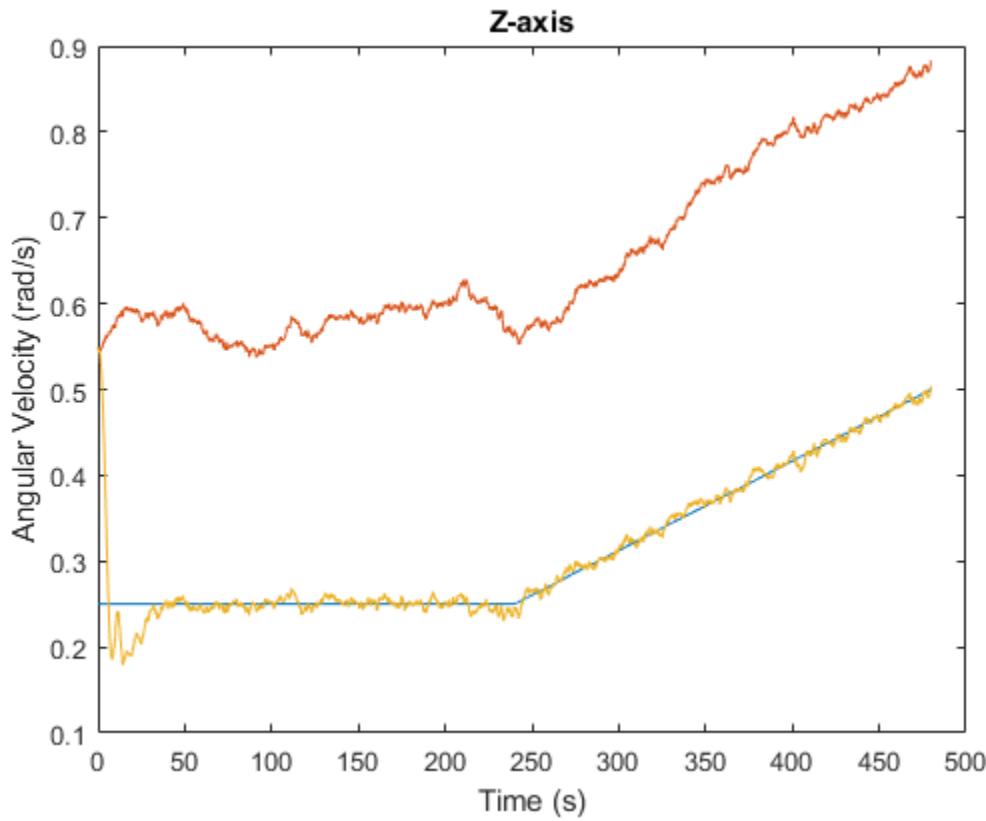
figure(1)
plot(time,angVelBody(:,1), ...
    time,gyroReadingsBody(:,1), ...
    time,angVelBodyRecovered(:,1))
title('X-axis')
legend('True Angular Velocity', ...
    'Gyroscope Readings', ...
    'Recovered Angular Velocity')
ylabel('Angular Velocity (rad/s)')
```



```
figure(2)
plot(time,angVelBody(:,2), ...
      time,gyroReadingsBody(:,2), ...
      time,angVelBodyRecovered(:,2))
title('Y-axis')
ylabel('Angular Velocity (rad/s)')
```



```
figure(3)
plot(time,angVelBody(:,3), ...
      time,gyroReadingsBody(:,3), ...
      time,angVelBodyRecovered(:,3))
title('Z-axis')
ylabel('Angular Velocity (rad/s)')
xlabel('Time (s)')
```



## Algorithms

*Note: The following algorithm only applies to an NED reference frame.*

The `imufilter` uses the six-axis Kalman filter structure described in [1]. The algorithm attempts to track the errors in orientation, gyroscope offset, and linear acceleration to output the final orientation and angular velocity. Instead of tracking the orientation directly, the indirect Kalman filter models the error process,  $x$ , with a recursive update:

$$x_k = \begin{bmatrix} \theta_k \\ b_k \\ a_k \end{bmatrix} = F_k \begin{bmatrix} \theta_{k-1} \\ b_{k-1} \\ a_{k-1} \end{bmatrix} + w_k$$

where  $x_k$  is a 9-by-1 vector consisting of:

- $\theta_k$  -- 3-by-1 orientation error vector, in degrees, at time  $k$
- $b_k$  -- 3-by-1 gyroscope zero angular rate bias vector, in deg/s, at time  $k$
- $a_k$  -- 3-by-1 acceleration error vector measured in the sensor frame, in g, at time  $k$
- $w_k$  -- 9-by-1 additive noise vector
- $F_k$  -- state transition model

Because  $x_k^-$  is defined as the error process, the *a priori* estimate is always zero, and therefore the state transition model,  $F_k$ , is zero. This insight results in the following reduction of the standard Kalman equations:

Standard Kalman equations:

$$\begin{aligned}x_k^- &= F_k x_k^+ - 1 \\P_k^- &= F_k P_{k-1}^+ F_k^T + Q_k \\y_k &= z_k - H_k x_k^- \\S_k &= R_k + H_k P_k^- H_k^T \\K_k &= P_k^- H_k^T (S_k)^{-1} \\x_k^+ &= x_k^- + K_k y_k \\P_k^+ &= P_k^- - K_k H_k P_k^-\end{aligned}$$

Kalman equations used in this algorithm:

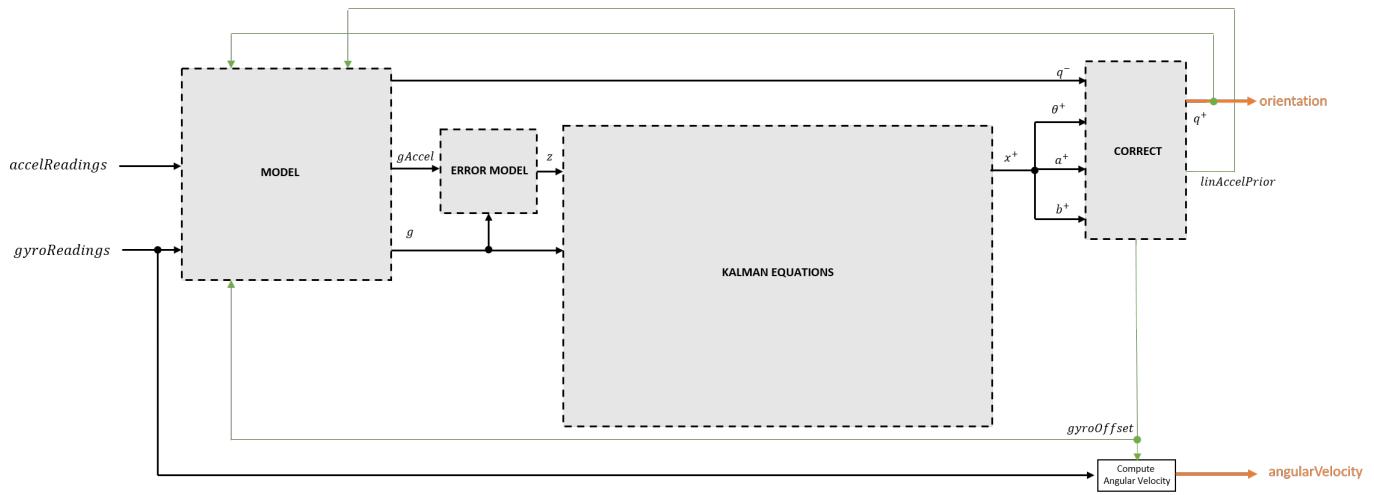
$$\begin{aligned}x_k^- &= 0 \\P_k^- &= Q_k \\y_k &= z_k \\S_k &= R_k + H_k P_k^- H_k^T \\K_k &= P_k^- H_k^T (S_k)^{-1} \\x_k^+ &= K_k y_k \\P_k^+ &= P_k^- - K_k H_k P_k^-\end{aligned}$$

where

- $x_k^-$  -- predicted (*a priori*) state estimate; the error process
- $P_k^-$  -- predicted (*a priori*) estimate covariance
- $y_k$  -- innovation
- $S_k$  -- innovation covariance
- $K_k$  -- Kalman gain
- $x_k^+$  -- updated (*a posteriori*) state estimate
- $P_k^+$  -- updated (*a posteriori*) estimate covariance

$k$  represents the iteration, the superscript  $+$  represents an *a posteriori* estimate, and the superscript  $-$  represents an *a priori* estimate.

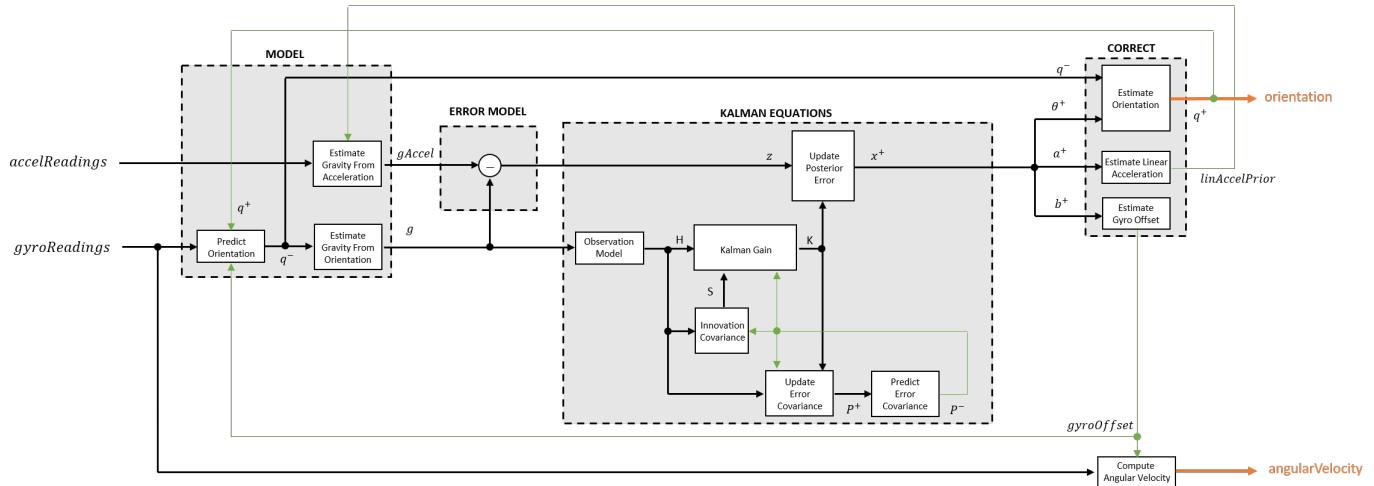
The graphic and following steps describe a single frame-based iteration through the algorithm.



Before the first iteration, the `accelReadings` and `gyroReadings` inputs are chunked into 1-by-3 frames and `DecimationFactor`-by-3 frames, respectively. The algorithm uses the most current accelerometer readings corresponding to the chunk of gyroscope readings.

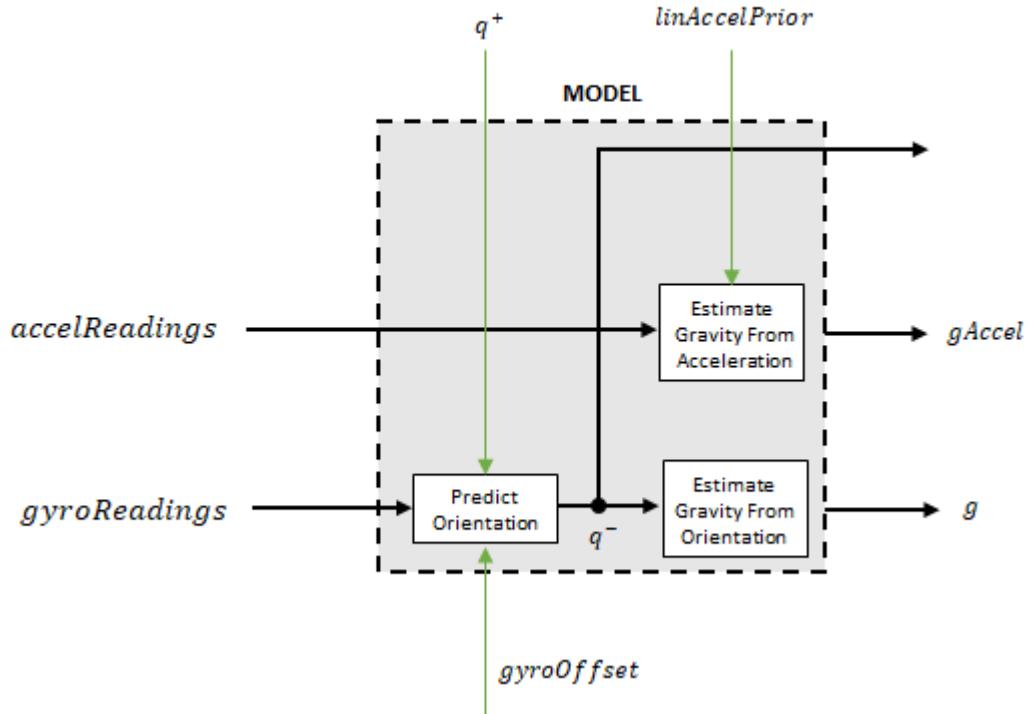
## Detailed Overview

Step through the algorithm for an explanation of each stage of the detailed overview.



## Model

The algorithm models acceleration and angular change as linear processes.



#### Predict Orientation

The orientation for the current frame is predicted by first estimating the angular change from the previous frame:

$$\Delta\varphi_{N \times 3} = \frac{(gyroReadings_{N \times 3} - gyroOffset_{1 \times 3})}{fs}$$

where  $N$  is the decimation factor specified by the `DecimationFactor` property, and  $fs$  is the sample rate specified by the `SampleRate` property.

The angular change is converted into quaternions using the `rotvec quaternion` construction syntax:

$$\Delta Q_{N \times 1} = \text{quaternion}(\Delta\varphi_{N \times 3}, 'rotvec')$$

The previous orientation estimate is updated by rotating it by  $\Delta Q$ :

$$q_{1 \times 1}^- = (q_{1 \times 1}^+ \left( \prod_{n=1}^N \Delta Q_n \right))$$

During the first iteration, the orientation estimate,  $q^-$ , is initialized by `ecompass` with an assumption that the x-axis points north.

#### Estimate Gravity from Orientation

The gravity vector is interpreted as the third column of the quaternion,  $q^-$ , in rotation matrix form:

$$g_{1 \times 3} = (rPrior(:, 3))^T$$

See `ecompass` for an explanation of why the third column of `rPrior` can be interpreted as the gravity vector.

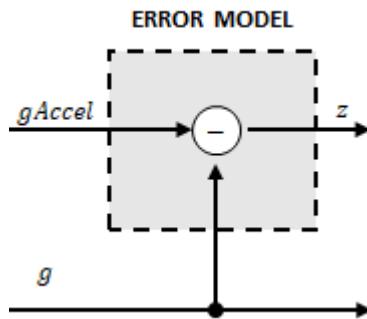
### Estimate Gravity from Acceleration

A second gravity vector estimation is made by subtracting the decayed linear acceleration estimate of the previous iteration from the accelerometer readings:

$$gAccel_{1 \times 3} = accelReadings_{1 \times 3} - linAccelprior_{1 \times 3}$$

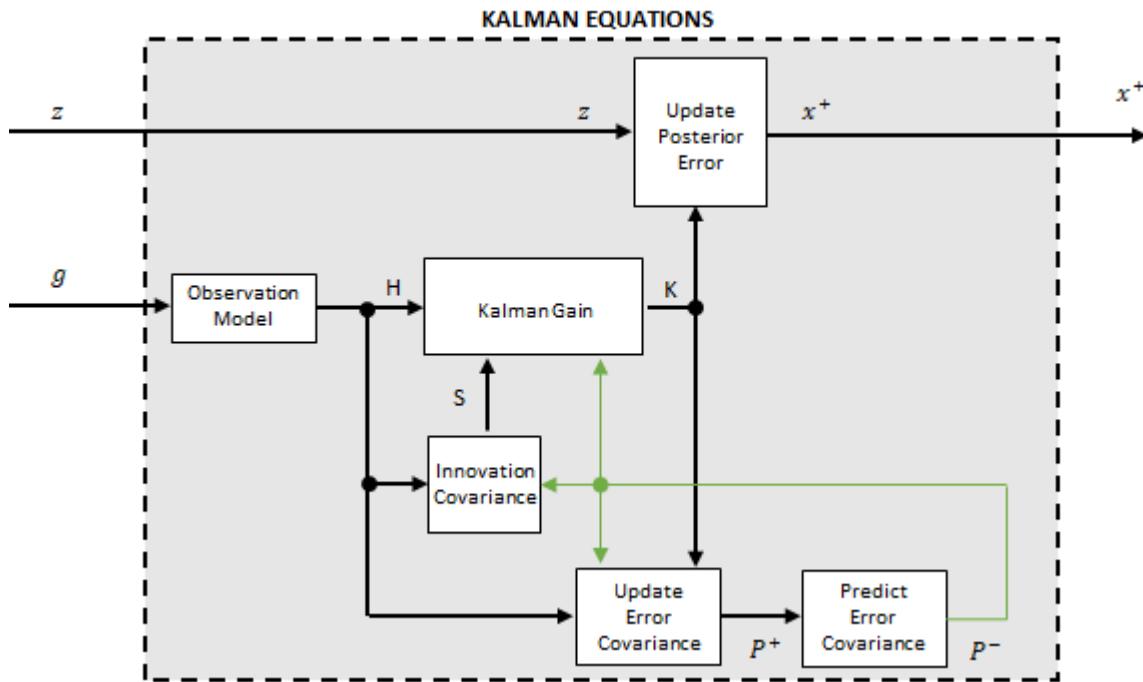
### Error Model

The error model is the difference between the gravity estimate from the accelerometer readings and the gravity estimate from the gyroscope readings:  $z = g - gAccel$ .



### Kalman Equations

The Kalman equations use the gravity estimate derived from the gyroscope readings,  $g$ , and the observation of the error process,  $z$ , to update the Kalman gain and intermediary covariance matrices. The Kalman gain is applied to the error signal,  $z$ , to output an *a posteriori* error estimate,  $x^+$ .



#### Observation Model

The observation model maps the 1-by-3 observed state,  $g$ , into the 3-by-9 true state,  $H$ .

The observation model is constructed as:

$$H_{3 \times 9} = \begin{bmatrix} 0 & g_z & -g_y & 0 & -\kappa g_z & \kappa g_y & 1 & 0 & 0 \\ -g_z & 0 & g_x & \kappa g_z & 0 & -\kappa g_x & 0 & 1 & 0 \\ g_y & -g_x & 0 & -\kappa g_y & \kappa g_x & 0 & 0 & 0 & 1 \end{bmatrix}$$

where  $g_x$ ,  $g_y$ , and  $g_z$  are the x-, y-, and z-elements of the gravity vector estimated from the orientation, respectively.  $\kappa$  is a constant determined by the SampleRate and DecimationFactor properties:  $\kappa = \text{DecimationFactor}/\text{SampleRate}$ .

See sections 7.3 and 7.4 of [1] for a derivation of the observation model.

#### Innovation Covariance

The innovation covariance is a 3-by-3 matrix used to track the variability in the measurements. The innovation covariance matrix is calculated as:

$$S_{3 \times 3} = R_{3 \times 3} + (H_{3 \times 9})(P_{9 \times 9})^T$$

where

- $H$  is the observation model matrix
- $P^-$  is the predicted (*a priori*) estimate of the covariance of the observation model calculated in the previous iteration

- $R$  is the covariance of the observation model noise, calculated as:

$$R_{3 \times 3} = (\lambda + \xi + \kappa(\beta + \eta)) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The following properties define the observation model noise variance:

- $\kappa$  -- (DecimationFactor/SampleRate)<sup>2</sup>
- $\beta$  -- GyroscopeDriftNoise
- $\eta$  -- GyroscopeNoise
- $\lambda$  -- AccelerometerNoise
- $\xi$  -- LinearAccelerationNoise

### **Update Error Estimate Covariance**

The error estimate covariance is a 9-by-9 matrix used to track the variability in the state.

The error estimate covariance matrix is updated as:

$$P_{9 \times 9}^+ = P_{9 \times 9}^- - (K_{9 \times 3})(H_{3 \times 9})(P_{9 \times 9}^-)$$

where  $K$  is the Kalman gain,  $H$  is the measurement matrix, and  $P^-$  is the error estimate covariance calculated during the previous iteration.

### **Predict Error Estimate Covariance**

The error estimate covariance is a 9-by-9 matrix used to track the variability in the state. The *a priori* error estimate covariance,  $P^-$ , is set to the process noise covariance,  $Q$ , determined during the previous iteration.  $Q$  is calculated as a function of the *a posteriori* error estimate covariance,  $P^+$ . When calculating  $Q$ , the cross-correlation terms are assumed to be negligible compared to the autocorrelation terms, and are set to zero:

$$Q =$$

$$\left[ \begin{array}{cccc} P^+(1) + \kappa^2 P^+(31) + \beta + \eta & 0 & 0 & -\kappa(P^+(31) + \beta) & 0 \\ 0 & P^+(11) + \kappa^2 P^+(41) + \beta + \eta & 0 & 0 & P^+(41) + \beta \\ 0 & 0 & P^+(21) + \kappa^2 P^+(51) + \beta + \eta & 0 & 0 & P^+(51) + \beta \\ -\kappa(P^+(31) + \beta) & 0 & 0 & P^+(31) + \beta & 0 \\ 0 & P^+(41) + \beta & 0 & 0 & P^+(41) + \beta \\ 0 & 0 & P^+(51) + \beta & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right]$$

where

- $P^+$  -- is the updated (*a posteriori*) error estimate covariance
- $\kappa$  -- DecimationFactor/SampleRate
- $\beta$  -- GyroscopeDriftNoise
- $\eta$  -- GyroscopeNoise
- $\nu$  -- LinearAccelerationDecayFactor
- $\xi$  -- LinearAccelerationNoise

See section 10.1 of [1] for a derivation of the terms of the process error matrix.

### Kalman Gain

The Kalman gain matrix is a 9-by-3 matrix used to weight the innovation. In this algorithm, the innovation is interpreted as the error process,  $z$ .

The Kalman gain matrix is constructed as:

$$K_{9 \times 3} = (P_{9 \times 9}^-)^{-1} (H_{3 \times 9})^T ((S_{3 \times 3})^T)^{-1}$$

where

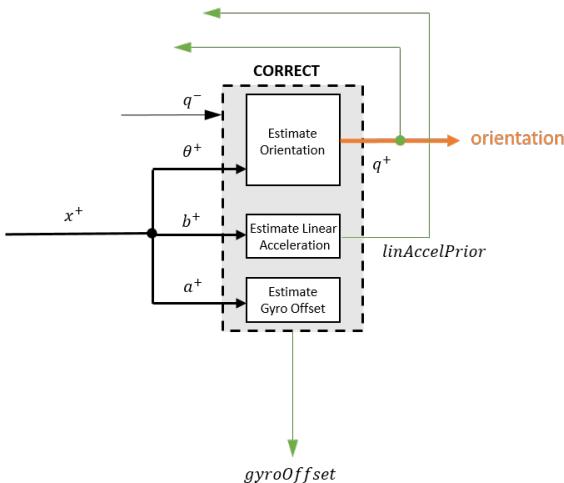
- $P^-$  -- predicted error covariance
- $H$  -- observation model
- $S$  -- innovation covariance

### Update a Posteriori Error

The *a posteriori* error estimate is determined by combining the Kalman gain matrix with the error in the gravity vector estimations:

$$x_{9 \times 1} = (K_{9 \times 3})(z_{1 \times 3})^T$$

### Correct



### Estimate Orientation

The orientation estimate is updated by multiplying the previous estimation by the error:

$$q^+ = (q^-)(\theta^+)$$

### Estimate Linear Acceleration

The linear acceleration estimation is updated by decaying the linear acceleration estimation from the previous iteration and subtracting the error:

$$linAccelPrior = (linAccelPrior_{k-1})\nu - b^+$$

where

- $\nu$  -- LinearAccelerationDecayFactor

#### Estimate Gyroscope Offset

The gyroscope offset estimation is updated by subtracting the gyroscope offset error from the gyroscope offset from the previous iteration:

$$gyroOffset = gyroOffset_{k-1} - a^+$$

#### Compute Angular Velocity

To estimate angular velocity, the frame of `gyroReadings` are averaged and the gyroscope offset computed in the previous iteration is subtracted:

$$\text{angularVelocity}_{1 \times 3} = \frac{\sum \text{gyroReadings}_{N \times 3}}{N} - gyroOffset_{1 \times 3}$$

where  $N$  is the decimation factor specified by the `DecimationFactor` property.

The gyroscope offset estimation is initialized to zeros for the first iteration.

## References

- [1] Open Source Sensor Fusion. <https://github.com/memsindustrygroup/Open-Source-Sensor-Fusion/tree/master/docs>
- [2] Roetenberg, D., H.J. Luinge, C.T.M. Baten, and P.H. Veltink. "Compensation of Magnetic Disturbances Improves Inertial and Magnetic Sensing of Human Body Segment Orientation." *IEEE Transactions on Neural Systems and Rehabilitation Engineering*. Vol. 13. Issue 3, 2005, pp. 395-405.

## 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

`ahrsfilter` | `ecompass` | `gpsSensor` | `imuSensor` | `quaternion`

### Topics

"Determine Orientation Using Inertial Sensors"

### Introduced in R2018b

# tune

Tune `imufilter` parameters to reduce estimation error

## Syntax

```
tune(filter,sensorData,groundTruth)
tune(___,config)
```

## Description

`tune(filter,sensorData,groundTruth)` adjusts the properties of the `imufilter` filter object, `filter`, to reduce the root-mean-squared (RMS) quaternion distance error between the fused sensor data and the ground truth. The function fuses the sensor data to estimate the orientation, which is compared to the orientation in the ground truth. The function uses the property values in the filter as the initial estimate for the optimization algorithm.

`tune(___,config)` specifies the tuning configuration based on a `tunerconfig` object, `config`.

## Examples

### Tune `imufilter` to Optimize Orientation Estimate

Load recorded sensor data and ground truth data.

```
ld = load('imufilterTuneData.mat');
qTrue = ld.groundTruth.Orientation; % true orientation
```

Create an `imufilter` object and fuse the filter with the sensor data.

```
fuse = imufilter;
qEstUntuned = fuse(ld.sensorData.Accelerometer, ...
    ld.sensorData.Gyroscope);
```

Create a `tunerconfig` object and tune the `imufilter` to improve the orientation estimate.

```
cfg = tunerconfig('imufilter');
tune(fuse, ld.sensorData, ld.groundTruth, cfg);
```

Iteration	Parameter	Metric
1	AccelerometerNoise	0.1149
1	GyroscopeNoise	0.1146
1	GyroscopeDriftNoise	0.1146
1	LinearAccelerationNoise	0.1122
1	LinearAccelerationDecayFactor	0.1103
2	AccelerometerNoise	0.1102
2	GyroscopeNoise	0.1098
2	GyroscopeDriftNoise	0.1098
2	LinearAccelerationNoise	0.1070
2	LinearAccelerationDecayFactor	0.1053
3	AccelerometerNoise	0.1053

3	GyroscopeNoise	0.1048
3	GyroscopeDriftNoise	0.1048
3	LinearAccelerationNoise	0.1016
3	LinearAccelerationDecayFactor	0.1002
4	AccelerometerNoise	0.1001
4	GyroscopeNoise	0.0996
4	GyroscopeDriftNoise	0.0996
4	LinearAccelerationNoise	0.0962
4	LinearAccelerationDecayFactor	0.0950
5	AccelerometerNoise	0.0950
5	GyroscopeNoise	0.0943
5	GyroscopeDriftNoise	0.0943
5	LinearAccelerationNoise	0.0910
5	LinearAccelerationDecayFactor	0.0901
6	AccelerometerNoise	0.0900
6	GyroscopeNoise	0.0893
6	GyroscopeDriftNoise	0.0893
6	LinearAccelerationNoise	0.0862
6	LinearAccelerationDecayFactor	0.0855
7	AccelerometerNoise	0.0855
7	GyroscopeNoise	0.0848
7	GyroscopeDriftNoise	0.0848
7	LinearAccelerationNoise	0.0822
7	LinearAccelerationDecayFactor	0.0818
8	AccelerometerNoise	0.0817
8	GyroscopeNoise	0.0811
8	GyroscopeDriftNoise	0.0811
8	LinearAccelerationNoise	0.0791
8	LinearAccelerationDecayFactor	0.0789
9	AccelerometerNoise	0.0788
9	GyroscopeNoise	0.0782
9	GyroscopeDriftNoise	0.0782
9	LinearAccelerationNoise	0.0769
9	LinearAccelerationDecayFactor	0.0768
10	AccelerometerNoise	0.0768
10	GyroscopeNoise	0.0762
10	GyroscopeDriftNoise	0.0762
10	LinearAccelerationNoise	0.0754
10	LinearAccelerationDecayFactor	0.0753
11	AccelerometerNoise	0.0753
11	GyroscopeNoise	0.0747
11	GyroscopeDriftNoise	0.0747
11	LinearAccelerationNoise	0.0741
11	LinearAccelerationDecayFactor	0.0740
12	AccelerometerNoise	0.0740
12	GyroscopeNoise	0.0734
12	GyroscopeDriftNoise	0.0734
12	LinearAccelerationNoise	0.0728
12	LinearAccelerationDecayFactor	0.0728
13	AccelerometerNoise	0.0728
13	GyroscopeNoise	0.0721
13	GyroscopeDriftNoise	0.0721
13	LinearAccelerationNoise	0.0715
13	LinearAccelerationDecayFactor	0.0715
14	AccelerometerNoise	0.0715
14	GyroscopeNoise	0.0706
14	GyroscopeDriftNoise	0.0706
14	LinearAccelerationNoise	0.0700

```

14      LinearAccelerationDecayFactor    0.0700
15      AccelerometerNoise            0.0700
15      GyroscopeNoise              0.0690
15      GyroscopeDriftNoise         0.0690
15      LinearAccelerationNoise     0.0684
15      LinearAccelerationDecayFactor 0.0684
16      AccelerometerNoise            0.0684
16      GyroscopeNoise              0.0672
16      GyroscopeDriftNoise         0.0672
16      LinearAccelerationNoise     0.0668
16      LinearAccelerationDecayFactor 0.0667
17      AccelerometerNoise            0.0667
17      GyroscopeNoise              0.0655
17      GyroscopeDriftNoise         0.0655
17      LinearAccelerationNoise     0.0654
17      LinearAccelerationDecayFactor 0.0654
18      AccelerometerNoise            0.0654
18      GyroscopeNoise              0.0641
18      GyroscopeDriftNoise         0.0641
18      LinearAccelerationNoise     0.0640
18      LinearAccelerationDecayFactor 0.0639
19      AccelerometerNoise            0.0639
19      GyroscopeNoise              0.0627
19      GyroscopeDriftNoise         0.0627
19      LinearAccelerationNoise     0.0627
19      LinearAccelerationDecayFactor 0.0624
20      AccelerometerNoise            0.0624
20      GyroscopeNoise              0.0614
20      GyroscopeDriftNoise         0.0614
20      LinearAccelerationNoise     0.0613
20      LinearAccelerationDecayFactor 0.0613

```

Fuse the sensor data again using the tuned filter.

```
qEstTuned = fuse(ld.sensorData.Accelerometer, ...
    ld.sensorData.Gyroscope);
```

Compare the tuned and untuned filter RMS error performances.

```
dUntuned = rad2deg(dist(qEstUntuned, qTrue));
dTuned = rad2deg(dist(qEstTuned, qTrue));
rmsUntuned = sqrt(mean(dUntuned.^2))

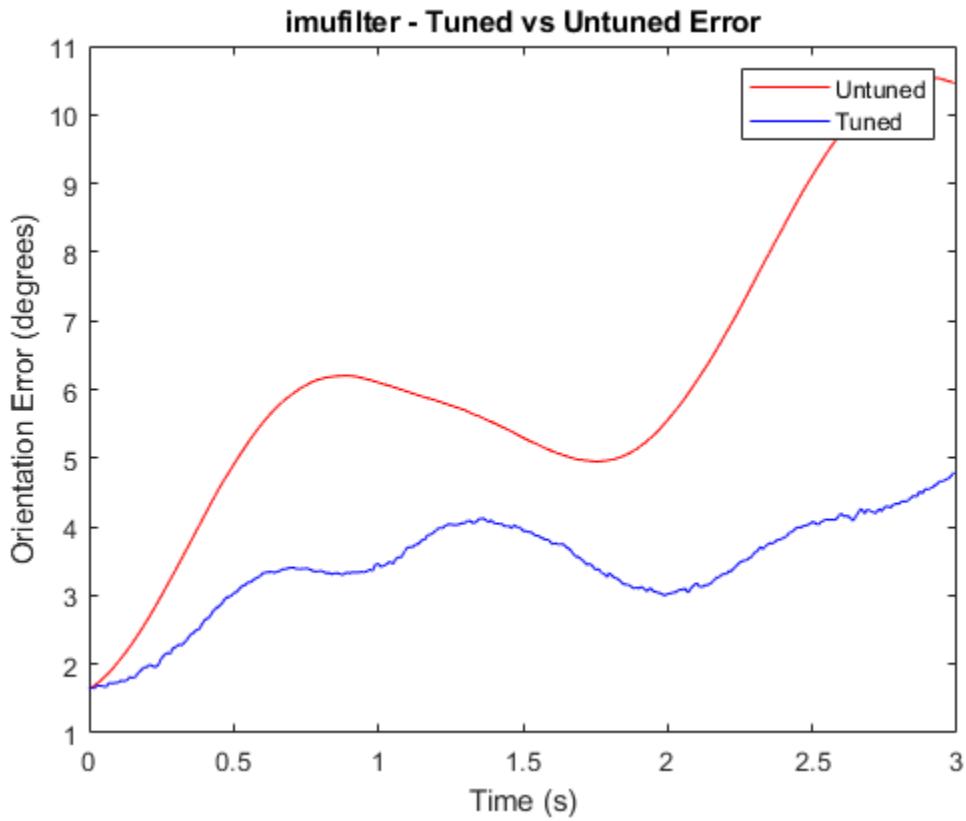
rmsUntuned = 6.5864

rmsTuned = sqrt(mean(dTuned.^2))

rmsTuned = 3.5098
```

Visualize the results.

```
N = numel(dUntuned);
t = (0:N-1)./ fuse.SampleRate;
plot(t, dUntuned, 'r', t, dTuned, 'b');
legend('Untuned', 'Tuned');
title('imufilter - Tuned vs Untuned Error')
xlabel('Time (s)');
ylabel('Orientation Error (degrees)');
```



## Input Arguments

### **filter — Filter object**

imufilter object

Filter object, specified as an `imufilter` object.

### **sensorData — Sensor data**

table

Sensor data, specified as a `table`. In each row, the sensor data is specified as:

- **Accelerometer** — Accelerometer data, specified as a 1-by-3 vector of scalars in  $\text{m}^2/\text{s}$ .
- **Gyroscope** — Gyroscope data, specified as a 1-by-3 vector of scalars in  $\text{rad}/\text{s}$ .

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `sensorData` input based on your choice.

### **groundTruth — Ground truth data**

timetable

Ground truth data, specified as a `table`. The table has only one column of `Orientation` data. In each row, the orientation is specified as a `quaternion` object or a 3-by-3 rotation matrix.

---

The function processes each row of the `sensorData` and `groundTruth` tables sequentially to calculate the state estimate and RMS error from the ground truth. Each row of the `sensorData` and the `groundTruth` tables must correspond to each other.

If you set the `Cost` property of the tuner configuration input, `config`, to `Custom`, then you can use other data types for the `groundTruth` input based on your choice.

**config — Tuner configuration**

`tunerconfig` object

Tuner configuration, specified as a `tunerconfig` object.

## References

- [1] Abbeel, P., Coates, A., Montemerlo, M., Ng, A.Y. and Thrun, S. Discriminative Training of Kalman Filters. In *Robotics: Science and systems*, Vol. 2, pp. 1, 2005.

## See Also

**Introduced in R2020b**

# insSensor

Inertial navigation and GPS simulation model

## Description

The `insSensor` System object models data output from an inertial navigation and GPS.

To model output from an inertial navigation and GPS:

- 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 an inertial navigation and GPS reading based on an inertial input signal.

`INS = insSensor(Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

## 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](#).

### **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 defined as rotation around the x-axis of the sensor body. Roll noise is modeled as a white noise process. `RollAccuracy` sets the standard deviation, in degrees, 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 defined as rotation around the  $y$ -axis of the sensor body. Pitch noise is modeled as a white noise process. **PitchAccuracy** defines the standard deviation, in degrees, 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 defined as rotation around the  $z$ -axis of the sensor body. Yaw noise is modeled as a white noise process. **YawAccuracy** defines the standard deviation, in degrees, of the yaw measurement noise.

**Tunable:** Yes

Data Types: single | double

**PositionAccuracy — Accuracy of position measurement (m)**

1 (default) | nonnegative real scalar

Accuracy of the position measurement of the sensor body in meters, specified as a nonnegative real scalar.

Position noise is modeled as a white noise process. **PositionAccuracy** defines the standard deviation, in meters, 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, in meters per second, of the velocity measurement noise.

**Tunable:** Yes

Data Types: single | double

**RandomStream — Random number source**

'Global stream' (default) | 'mt19937ar with seed'

Random number source, specified as a character vector:

- 'Global stream' -- Random numbers are generated using the current global random number stream.
- 'mt19937ar with seed' -- Random numbers are generated using the mt19937ar algorithm with the seed specified by the **Seed** property.

Data Types: char | string

**Seed — Initial seed**

67 (default) | nonnegative integer scalar

Initial seed of an mt19937ar random number generator algorithm, specified as a real, nonnegative integer scalar.

**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(motion)
```

### Description

`measurement = INS(motion)` models the data received from an inertial navigation and GPS reading. The measurement is based on the input signal, `motion`.

### Input Arguments

**motion — Ground-truth sensor body motion in local NED**

struct

The `motion` is a struct with the following fields:

- 'Position' -- Position of the sensor body in the local NED coordinate system specified as a real finite  $N$ -by-3 array in meters.  $N$  is the number of samples in the current frame.
- 'Velocity' -- Velocity of the sensor body in the local NED coordinate system specified as a real finite  $N$ -by-3 array in meters per second.  $N$  is the number of samples in the current frame.
- 'Orientation' -- Orientation of the sensor body with respect to the local NED coordinate system specified as a quaternion  $N$ -element column vector or a single or double 3-by-3-by- $N$  rotation matrix. Each quaternion or rotation matrix is a frame rotation from the local NED coordinate system to the current sensor body coordinate system.  $N$  is the number of samples in the current frame.

Example: `motion = struct('Position',[0,0,0],'Velocity',[0,0,0],'Orientation',quaternion([1,0,0,0]))`

## Output Arguments

**measurement — Measurement of sensor body motion in local NED**  
**struct**

**measurement** is a struct with the following fields:

- '**Position**' -- Position measurement of the sensor body in the local NED coordinate system specified as a real finite  $N$ -by-3 array in meters.  $N$  is the number of samples in the current frame.
- '**Velocity**' -- Velocity measurement of the sensor body in the local NED coordinate system specified as a real finite  $N$ -by-3 array in meters per second.  $N$  is the number of samples in the current frame.
- '**Orientation**' -- Orientation measurement of the sensor body with respect to the local NED coordinate system specified as a quaternion  $N$ -element column vector or a single or double 3-by-3 by- $N$  rotation matrix. Each quaternion or rotation matrix is a frame rotation from the local NED coordinate system to the current sensor body coordinate system.  $N$  is the number of samples in the current frame.

## 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

<b>perturbations</b>	Perturbation defined on object
<b>perturb</b>	Apply perturbations to object

### Common to All System Objects

<b>step</b>	Run System object algorithm
<b>release</b>	Release resources and allow changes to System object property values and input characteristics
<b>reset</b>	Reset internal states of System object

## Examples

### Generate INS Measurements from Stationary Input

Create a motion struct that defines a stationary position at the local NED origin. Because the platform is stationary, you only need to define 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 struct 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.

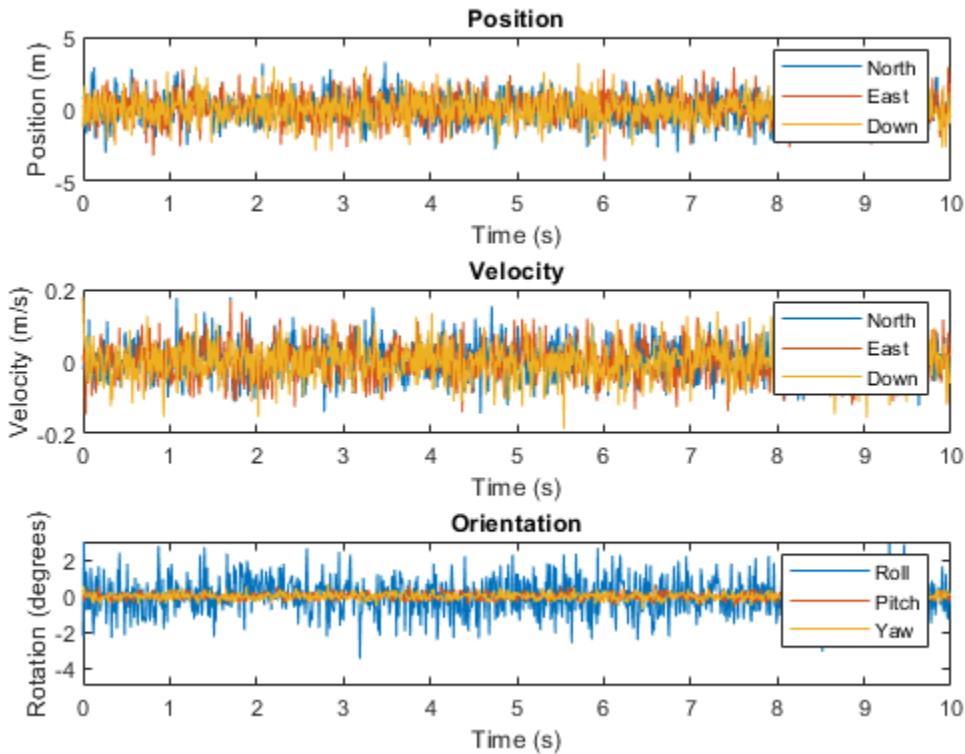
```
orientationMeasurements = eulerd(orientationMeasurements, 'ZYX','frame');

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 a Scenario

Generate INS measurements using the `insSensor` System object™. Use `waypointTrajectory` to generate the ground-truth path. Use `trackingScenario` 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`.

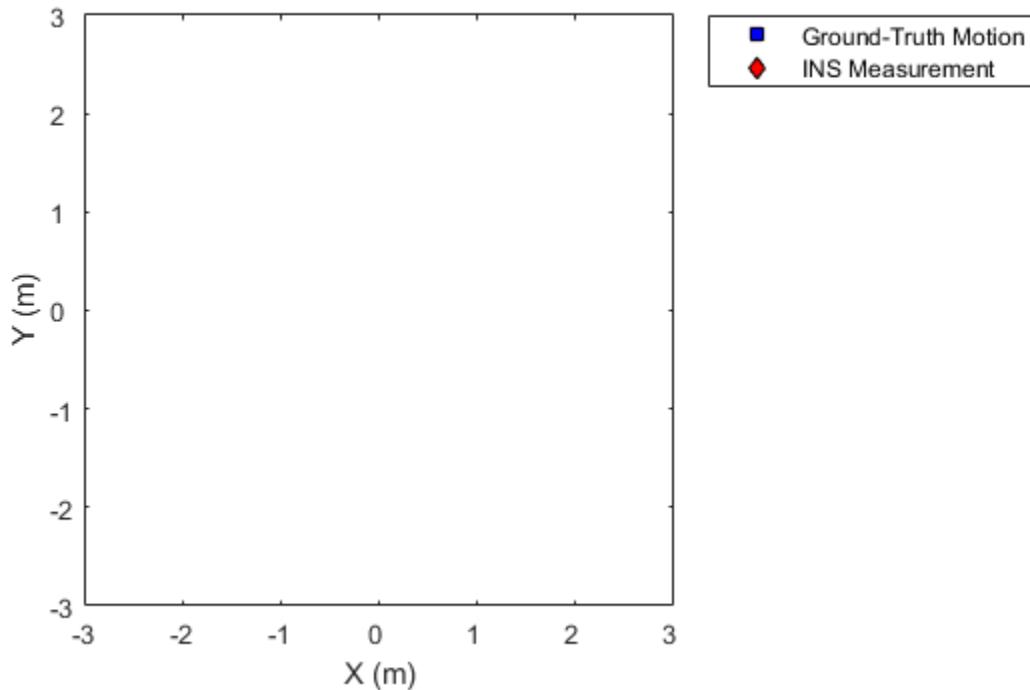
```
ins = insSensor('PositionAccuracy',0.1);
```

Create a tracking scenario with a single platform whose motion is defined by `path`.

```
scenario = trackingScenario('UpdateRate',Fs);
quadcopter = platform(scenario);
quadcopter.Trajectory = path;
```

Create a theater plot to visualize the ground-truth `quadcopter` motion and the quadcopter motion measurements modeled by `insSensor`.

```
tp = theaterPlot('XLimits',[-3, 3], 'YLimits', [-3, 3]);
quadPlotter = 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');
```

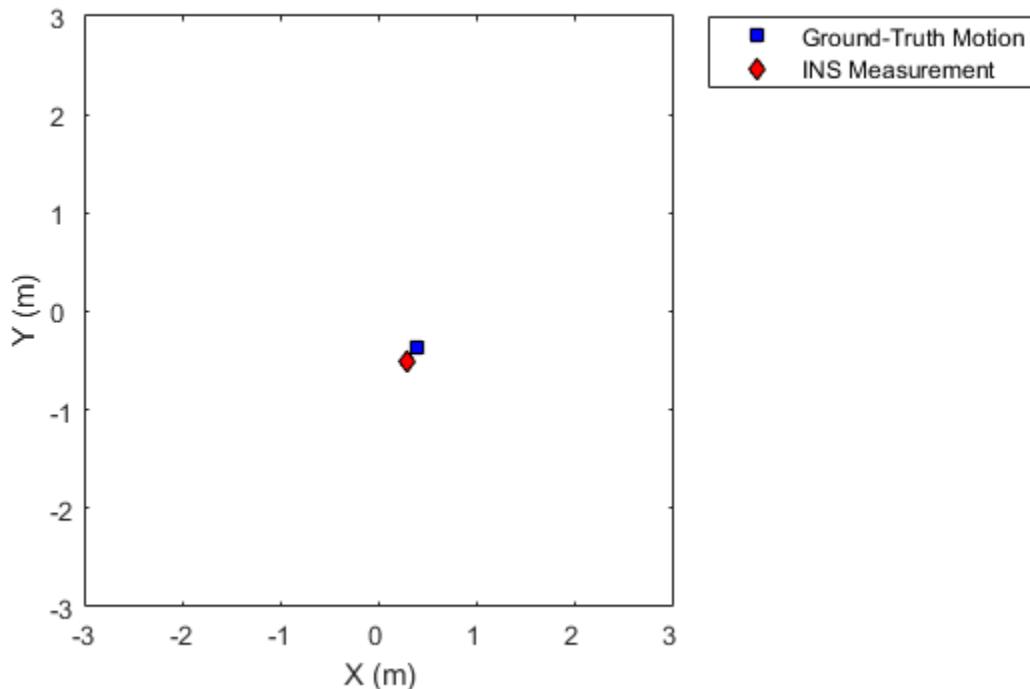
```

insMeas = ins(motion);

plotPlatform(quadPlotter,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.

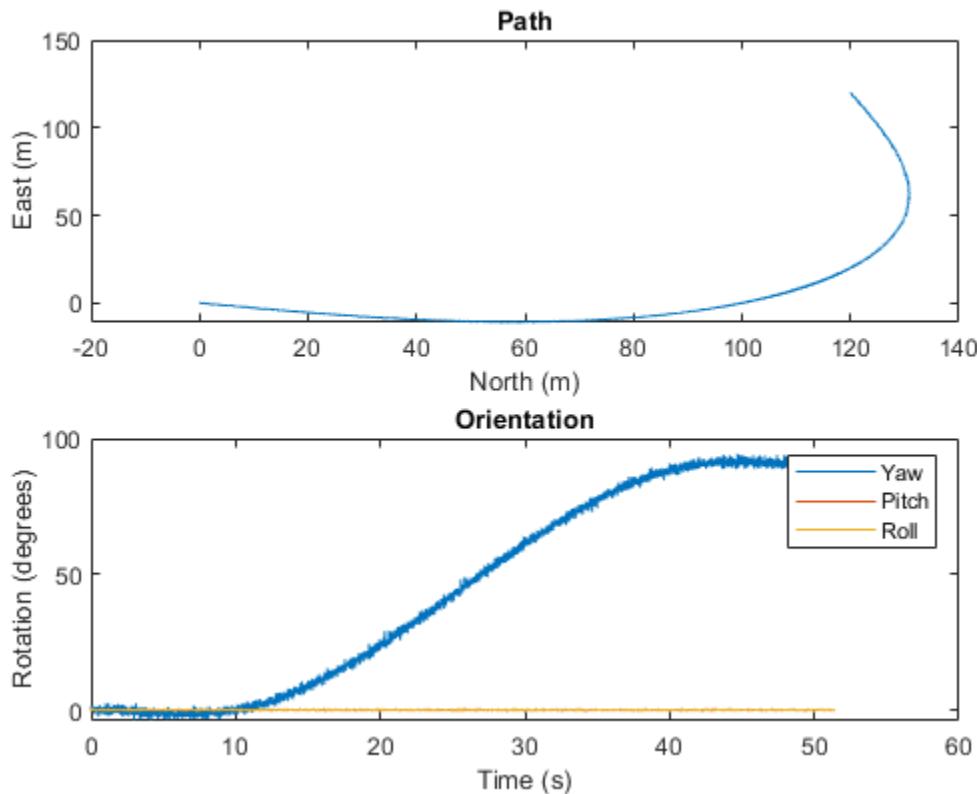
```
[motion.Position,motion.Orientation,motion.Velocity] = path();
insMeas = ins(motion);
```

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™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Objects

[gpsSensor](#) | [imuSensor](#)

### Topics

“Model IMU, GPS, and INS/GPS”

**Introduced in R2018b**

## gpsSensor

GPS receiver simulation model

### Description

The `gpsSensor` System object models data output from a Global Positioning System (GPS) receiver.

To model a GPS receiver:

- 1 Create the `gpsSensor` 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

```
GPS = gpsSensor  
GPS = gpsSensor('ReferenceFrame',RF)  
GPS = gpsSensor(___,Name,Value)
```

#### Description

`GPS = gpsSensor` returns a `gpsSensor` System object that computes a Global Positioning System receiver reading based on a local position and velocity input signal. The default reference position in geodetic coordinates is

- latitude:  $0^{\circ}$  N
- longitude:  $0^{\circ}$  E
- altitude: 0 m

`GPS = gpsSensor('ReferenceFrame',RF)` returns a `gpsSensor` System object that computes a global positioning system receiver reading relative to the reference frame RF. Specify RF as '`NED`' (North-East-Down) or '`ENU`' (East-North-Up). The default value is '`NED`'.

`GPS = gpsSensor(___,Name,Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values.

#### 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 — Update rate of receiver (Hz)**

1 (default) | positive real scalar

Update rate of the receiver in Hz, specified as a positive real scalar.

Data Types: `single` | `double`

**ReferenceLocation — Origin of local navigation reference frame**

[0 0 0] (default) | [degrees degrees meters]

Reference location, specified as a 3-element row vector in geodetic coordinates (latitude, longitude, and altitude). Altitude is the height above the reference ellipsoid model, WGS84. The reference location is in [degrees degrees meters]. The degree format is decimal degrees (DD).

Data Types: `single` | `double`

**HorizontalPositionAccuracy — Horizontal position accuracy (m)**

1.6 (default) | nonnegative real scalar

Horizontal position accuracy in meters, specified as a nonnegative real scalar. The horizontal position accuracy specifies the standard deviation of the noise in the horizontal position measurement.

**Tunable:** Yes

Data Types: `single` | `double`

**VerticalPositionAccuracy — Vertical position accuracy (m)**

3 (default) | nonnegative real scalar

Vertical position accuracy in meters, specified as a nonnegative real scalar. The vertical position accuracy specifies the standard deviation of the noise in the vertical position measurement.

**Tunable:** Yes

Data Types: `single` | `double`

**VelocityAccuracy — Velocity accuracy (m/s)**

0.1 (default) | nonnegative real scalar

Velocity accuracy in meters per second, specified as a nonnegative real scalar. The velocity accuracy specifies the standard deviation of the noise in the velocity measurement.

**Tunable:** Yes

Data Types: `single` | `double`

**DecayFactor — Global position noise decay factor**

0.999 (default) | scalar in the range [0,1]

Global position noise decay factor, specified as a scalar in the range [0,1].

A decay factor of 0 models the global position noise as a white noise process. A decay factor of 1 models the global position noise as a random walk process.

**Tunable:** Yes

Data Types: single | double

**RandomStream — Random number source**

'Global stream' (default) | 'mt19937ar with seed'

Random number source, specified as a character vector or string:

- 'Global stream' -- Random numbers are generated using the current global random number stream.
- 'mt19937ar with seed' -- Random numbers are generated using the mt19937ar algorithm with the seed specified by the **Seed** property.

Data Types: char | string

**Seed — Initial seed**

67 (default) | nonnegative integer scalar

Initial seed of an mt19937ar random number generator algorithm, specified as a nonnegative integer scalar.

**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

```
[position,velocity,groundspeed,course] = GPS(truePosition,trueVelocity)
```

### Description

```
[position,velocity,groundspeed,course] = GPS(truePosition,trueVelocity)
```

computes global navigation satellite system receiver readings from the position and velocity inputs.

### Input Arguments

**truePosition — Position of GPS receiver in local navigation coordinate system (m)**

*N*-by-3 matrix

Position of the GPS receiver in the local navigation coordinate system in meters, specified as a real finite *N*-by-3 matrix.

*N* is the number of samples in the current frame.

Data Types: single | double

**trueVelocity — Velocity of GPS receiver in local navigation coordinate system (m/s)**

*N*-by-3 matrix

Velocity of GPS receiver in the local navigation coordinate system in meters per second, specified as a real finite *N*-by-3 matrix.

*N* is the number of samples in the current frame.

Data Types: single | double

### Output Arguments

#### **position — Position in LLA coordinate system**

*N*-by-3 matrix

Position of the GPS receiver in the geodetic latitude, longitude, and altitude (LLA) coordinate system, returned as a real finite *N*-by-3 array. Latitude and longitude are in degrees with North and East being positive. Altitude is in meters.

*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 of the GPS receiver in the local navigation coordinate system in meters per second, returned as a real finite *N*-by-3 array.

*N* is the number of samples in the current frame.

Data Types: single | double

#### **groundspeed — Magnitude of horizontal velocity in local navigation coordinate system (m/s)**

*N*-by-1 column vector

Magnitude of the horizontal velocity of the GPS receiver in the local navigation coordinate system in meters per second, returned as a real finite *N*-by-1 column vector.

*N* is the number of samples in the current frame.

Data Types: single | double

#### **course — Direction of horizontal velocity in local navigation coordinate system (°)**

*N*-by-1 column vector

Direction of the horizontal velocity of the GPS receiver in the local navigation coordinate system in degrees, returned as a real finite *N*-by-1 column of values between 0 and 360. North corresponds to 360 degrees and East corresponds to 90 degrees.

*N* is the number of samples in the current frame.

Data Types: single | 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

### Generate GPS Position Measurements From Stationary Input

Create a `gpsSensor` System object™ to model GPS receiver data. Assume a typical one Hz sample rate and a 1000-second simulation time. Define the reference location in terms of latitude, longitude, and altitude (LLA) of Natick, MA (USA). Define the sensor as stationary by specifying the true position and velocity with zeros.

```
fs = 1;
duration = 1000;
numSamples = duration*fs;

refLoc = [42.2825 -71.343 53.0352];

truePosition = zeros(numSamples,3);
trueVelocity = zeros(numSamples,3);

gps = gpsSensor('SampleRate',fs,'ReferenceLocation',refLoc);
```

Call `gps` with the specified `truePosition` and `trueVelocity` to simulate receiving GPS data for a stationary platform.

```
position = gps(truePosition,trueVelocity);
```

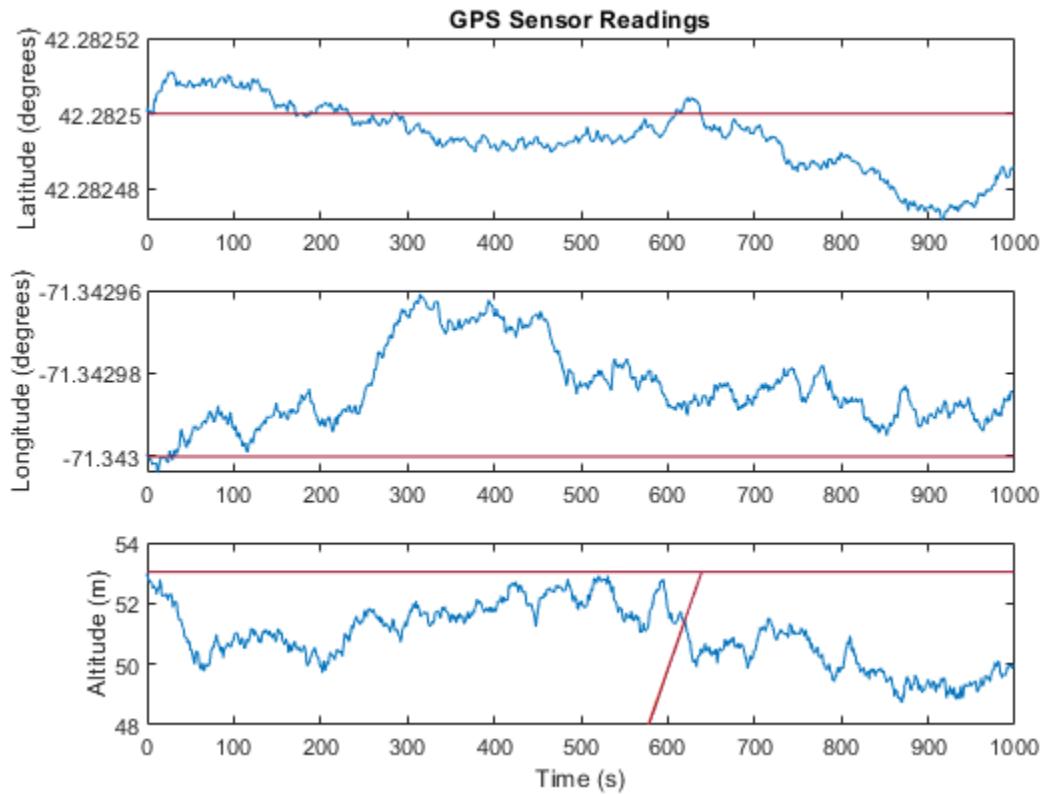
Plot the true position and the GPS sensor readings for position.

```
t = (0:(numSamples-1))/fs;

subplot(3, 1, 1)
plot(t, position(:,1), ...
      t, ones(numSamples)*refLoc(1))
title('GPS Sensor Readings')
ylabel('Latitude (degrees)')

subplot(3, 1, 2)
plot(t, position(:,2), ...
      t, ones(numSamples)*refLoc(2))
ylabel('Longitude (degrees)')

subplot(3, 1, 3)
plot(t, position(:,3), ...
      t, ones(numSamples)*refLoc(3))
ylabel('Altitude (m)')
xlabel('Time (s)')
```



The position readings have noise controlled by `HorizontalPositionAccuracy`, `VerticalPositionAccuracy`, `VelocityAccuracy`, and `DecayFactor`. The `DecayFactor` property controls the drift in the noise model. By default, `DecayFactor` is set to 0.999, which approaches a random walk process. To observe the effect of the `DecayFactor` property:

- 1** Reset the `gps` object.
- 2** Set `DecayFactor` to 0.5.
- 3** Call `gps` with variables specifying a stationary position.
- 4** Plot the results.

The GPS position readings now oscillate around the true position.

```
reset(gps)
gps.DecayFactor = 0.5;
position = gps(truePosition,trueVelocity);

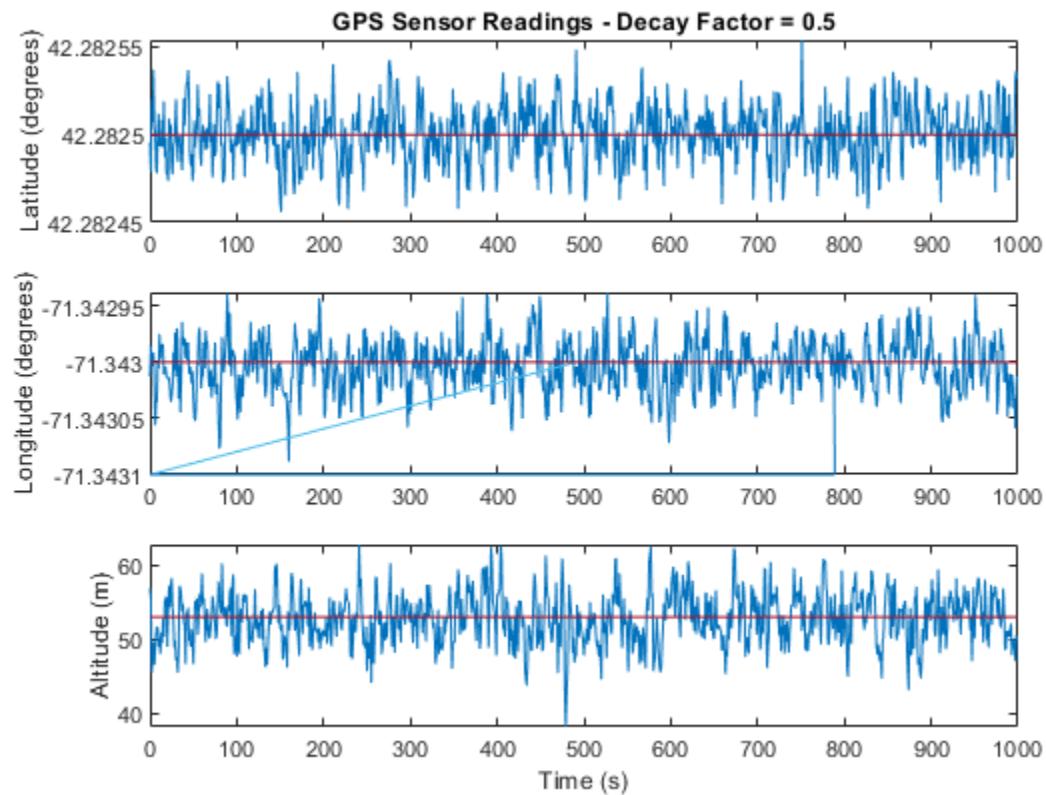
subplot(3, 1, 1)
plot(t, position(:,1), ...
      t, ones(numSamples)*refLoc(1))
title('GPS Sensor Readings - Decay Factor = 0.5')
ylabel('Latitude (degrees)')

subplot(3, 1, 2)
plot(t, position(:,2), ...
      t, ones(numSamples)*refLoc(2))
ylabel('Longitude (degrees)')
```

```

subplot(3, 1, 3)
plot(t, position(:,3), ...
      t, ones(numSamples)*refLoc(3))
ylabel('Altitude (m)')
xlabel('Time (s)')

```



### Relationship Between Groundspeed and Course Accuracy

GPS receivers achieve greater course accuracy as groundspeed increases. In this example, you create a GPS receiver simulation object and simulate the data received from a platform that is accelerating from a stationary position.

Create a default `gpsSensor` System object™ to model data returned by a GPS receiver.

```

GPS = gpsSensor
GPS =
    gpsSensor with properties:

        SampleRate: 1
        ReferenceLocation: [0 0 0]
        HorizontalPositionAccuracy: 1.6
        VerticalPositionAccuracy: 3
        VelocityAccuracy: 0.1

```

SampleRate:	1	Hz
ReferenceLocation:	[0 0 0]	[deg deg m]
HorizontalPositionAccuracy:	1.6	m
VerticalPositionAccuracy:	3	m
VelocityAccuracy:	0.1	m/s

```
RandomStream: 'Global stream'
DecayFactor: 0.999
```

Create matrices to describe the position and velocity of a platform in the NED coordinate system. The platform begins from a stationary position and accelerates to 60 m/s North-East over 60 seconds, then has a vertical acceleration to 2 m/s over 2 seconds, followed by a 2 m/s rate of climb for another 8 seconds. Assume a constant velocity, such that the velocity is the simple derivative of the position.

```
duration = 70;
numSamples = duration*GPS.SampleRate;

course = 45*ones(duration,1);
groundspeed = [(1:60)';60*ones(10,1)];

Nvelocity = groundspeed.*sind(course);
Evelocity = groundspeed.*cosd(course);
Dvelocity = [zeros(60,1);-1;-2*ones(9,1)];
NEDvelocity = [Nvelocity,Evelocity,Dvelocity];

Ndistance = cumsum(Nvelocity);
Edistance = cumsum(Evelocity);
Ddistance = cumsum(Dvelocity);
NEDposition = [Ndistance,Edistance,Ddistance];
```

Model GPS measurement data by calling the GPS object with your velocity and position matrices.

```
[~,~,groundspeedMeasurement,courseMeasurement] = GPS(NEDposition,NEDvelocity);
```

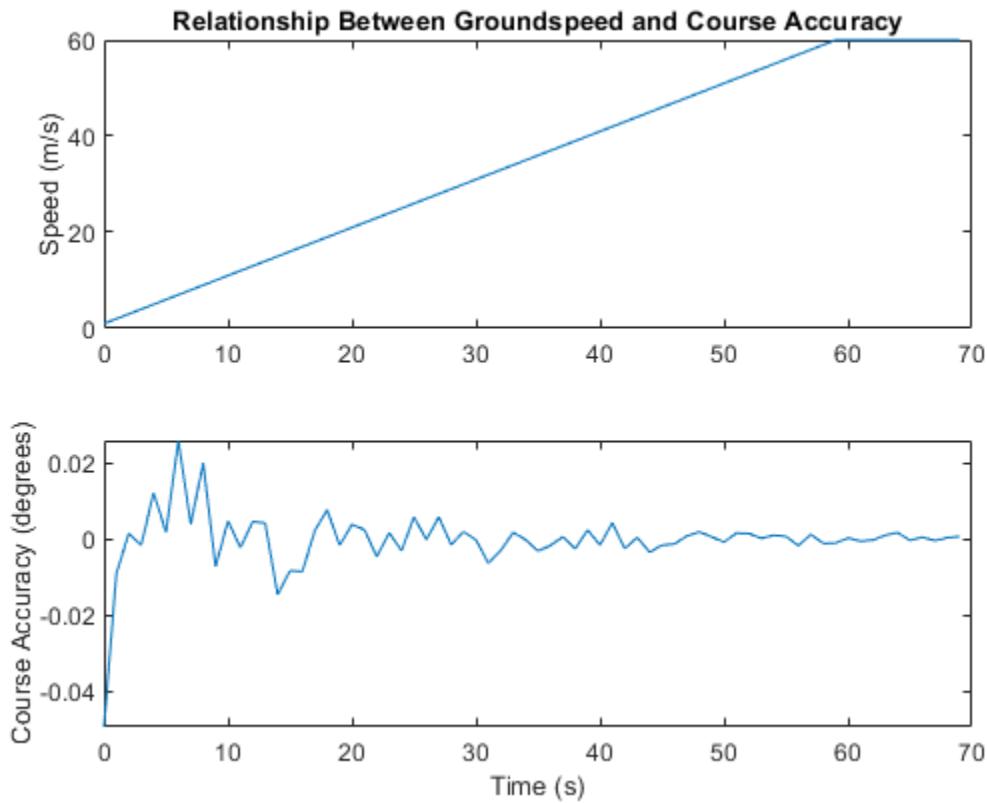
Plot the groundspeed and the difference between the true course and the course returned by the GPS simulator.

As groundspeed increases, the accuracy of the course increases. Note that the velocity increase during the last ten seconds has no effect, because the additional velocity is not in the ground plane.

```
t = (0:numSamples-1)/GPS.SampleRate;

subplot(2,1,1)
plot(t,groundspeed);
ylabel('Speed (m/s)')
title('Relationship Between Groundspeed and Course Accuracy')

subplot(2,1,2)
courseAccuracy = courseMeasurement - course;
plot(t,courseAccuracy)
xlabel('Time (s)');
ylabel('Course Accuracy (degrees)')
```



### Model GPS Receiver Data

Simulate GPS data received during a trajectory from the city of Natick, MA, to Boston, MA.

Define the decimal degree latitude and longitude for the city of Natick, MA USA, and Boston, MA USA. For simplicity, set the altitude for both locations to zero.

```
NatickLLA = [42.27752809999999, -71.34680909999997, 0];  
BostonLLA = [42.3600825, -71.05888010000001, 0];
```

Define a motion that can take a platform from Natick to Boston in 20 minutes. Set the origin of the local NED coordinate system as Natick. Create a `waypointTrajectory` object to output the trajectory 10 samples at a time.

```
fs = 1;  
duration = 60*20;  
  
bearing = 68; % degrees  
distance = 25.39e3; % meters  
distanceEast = distance*sind(bearing);  
distanceNorth = distance*cosd(bearing);  
  
NatickNED = [0,0,0];  
BostonNED = [distanceNorth,distanceEast,0];
```

```
trajectory = waypointTrajectory( ...
    'Waypoints', [NatickNED;BostonNED], ...
    'TimeOfArrival',[0;duration], ...
    'SamplesPerFrame',10, ...
    'SampleRate',fs);
```

Create a `gpsSensor` object to model receiving GPS data for the platform. Set the `HorizontalPositionAccuracy` to 25 and the `DecayFactor` to 0.25 to emphasize the noise. Set the `ReferenceLocation` to the Natick coordinates in LLA.

```
GPS = gpsSensor( ...
    'HorizontalPositionAccuracy',25, ...
    'DecayFactor',0.25, ...
    'SampleRate',fs, ...
    'ReferenceLocation',NatickLLA);
```

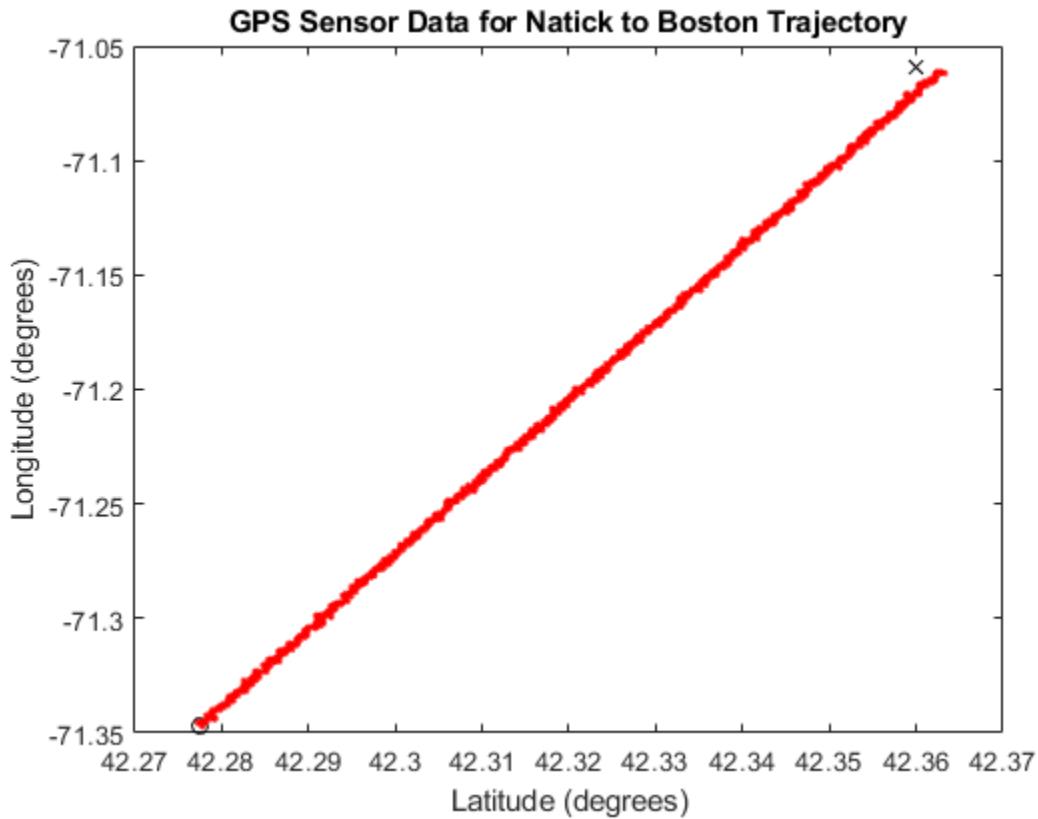
Open a figure and plot the position of Natick and Boston in LLA. Ignore altitude for simplicity.

In a loop, call the `gpsSensor` object with the ground-truth trajectory to simulate the received GPS data. Plot the ground-truth trajectory and the model of received GPS data.

```
figure(1)
plot(NatickLLA(1),NatickLLA(2),'ko', ...
    BostonLLA(1),BostonLLA(2),'kx')
xlabel('Latitude (degrees)')
ylabel('Longitude (degrees)')
title('GPS Sensor Data for Natick to Boston Trajectory')
hold on

while ~isDone(trajectory)
    [truePositionNED,~,trueVelocityNED] = trajectory();
    reportedPositionLLA = GPS(truePositionNED,trueVelocityNED);

    figure(1)
    plot(reportedPositionLLA(:,1),reportedPositionLLA(:,2),'r.')
end
```



As a best practice, release System objects when complete.

```
release(GPS)  
release(trajectory)
```

## 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

### Objects

[imuSensor](#) | [insSensor](#)

### Topics

[“Model IMU, GPS, and INS/GPS”](#)

## Introduced in R2018b

# radarSensor

Generate detections from radar emissions

## Description

The `radarSensor` System object returns a statistical model to generate detections from radar emissions. You can generate detections from monostatic radar, bistatic radar and Electronic Support Measures (ESM). You can use the `radarSensor` object in a scenario that models moving and stationary platforms using `trackingScenario`. The radar sensor can simulate real detections with added random noise and also generate false alarm detections. In addition, you can use this object to create input to trackers such as `trackerGNN`, `trackerJPDA` and `trackerTOMHT`.

This object enables you to configure a scanning radar. A scanning radar 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 scan limits, `MechanicalScanLimits`, and electronic scan limits, `ElectronicScanLimits`. If the scanning 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.

Using a single-exponential mode, 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 [1] and [2] for more details.

To generate radar detections:

- 1 Create the `radarSensor` 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

```
sensor = radarSensor(SensorIndex)

sensor = radarSensor(SensorIndex,'No scanning')
sensor = radarSensor(SensorIndex,'Raster')
sensor = radarSensor(SensorIndex,'Rotator')
sensor = radarSensor(SensorIndex,'Sector')

sensor = radarSensor(____,Name,Value)
```

## Description

`sensor = radarSensor(SensorIndex)` creates a radar detection generator object with a specified sensor index, `SensorIndex`, and default property values.

`sensor = radarSensor(SensorIndex, 'No scanning')` is a convenience syntax that creates a `radarSensor` that stares along the radar antenna boresight direction. No mechanical or electronic scanning is performed. This syntax sets the `ScanMode` property to '`No scanning`'.

`sensor = radarSensor(SensorIndex, 'Raster')` is a convenience syntax that creates a `radarSensor` object that mechanically scans a raster pattern. The raster span is 90° in azimuth from -45° to +45° and in elevation from the horizon to 10° above the horizon. See "Convenience Syntaxes" on page 3-115 for the properties set by this syntax.

`sensor = radarSensor(SensorIndex, 'Rotator')` is a convenience syntax that creates a `radarSensor` object that mechanically scans 360° 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 3-115 for the properties set by this syntax.

`sensor = radarSensor(SensorIndex, 'Sector')` is a convenience syntax to create a `radarSensor` object that mechanically scans a 90° azimuth sector from -45° to +45°. 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 3-115 for the properties set by this syntax.

`sensor = radarSensor(____, Name, Value)` sets properties using one or more name-value pairs after all other input arguments. Enclose each property name in quotes. For example, `radarSensor(1, 'DetectionCoordinates', 'Sensor cartesian', 'MaxRange', 200)` creates a radar detection generator that reports detections in the sensor Cartesian coordinate system and has a maximum detection range of 200 meters. If you specify the sensor index using the `SensorIndex` property, you can omit the `SensorIndex` 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.

### **SensorIndex — Unique sensor identifier**

positive integer

Unique sensor identifier, specified as a positive integer. This property distinguishes detections that come from different sensors in a multi-sensor system. When creating a `radarSensor` system object, you must either specify the `SensorIndex` as the first input argument in the creation syntax, or specify it as the value for the `SensorIndex` property in the creation syntax.

Example: 2

Data Types: double

#### **UpdateRate — Sensor update rate**

1 (default) | positive scalar

Sensor update rate, specified as a positive scalar. This interval must be an integer multiple of the simulation time interval defined by `trackingScenario`. The `trackingScenario` object calls the radar sensor at simulation time intervals. The radar generates new detections at intervals defined by the reciprocal of the `UpdateRate` property. Any update requested to the sensor between update intervals contains no detections. Units are in hertz.

Example: 5

Data Types: double

#### **DetectionMode — Detection mode**

'ESM' (default) | 'monostatic' | 'bistatic'

Detection mode, specified as 'ESM', 'monostatic' or 'bistatic'. When set to 'ESM', the sensor operates passively and can model ESM and RWR systems. When set to 'monostatic', the sensor generates detections from reflected signals originating from a collocated radar emitter. 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 3-110.

Example: 'Monostatic'

Data Types: char | string

#### **EmitterIndex — Unique monostatic emitter index**

positive integer

Unique monostatic emitter index, specified as a positive integer. The emitter index identifies the monostatic emitter providing the reference signal to the sensor.

Example: 404

#### **Dependencies**

To enable this property, set the `DetectionMode` property to 'Monostatic'.

Data Types: double

#### **HasElevation — Enable elevation scan and measurements**

false (default) | true

Enable the sensor to measure target elevation angles and to scan in elevation, specified as `false` or `true`. Set this property to `true` to model a radar sensor that can estimate target elevation and scan in elevation.

Data Types: logical

#### **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 dBm.

Example: -10

Data Types: double

**DetectionThreshold — Minimum SNR required to declare a detection**

5 (default) | scalar

Minimum SNR required to declare a detection, specified as a scalar. Units are in dB.

Example: -1

Data Types: double

**FalseAlarmRate — False alarm rate**

1e-6 (default) | positive scalar

False alarm report rate within each sensor resolution cell, specified as a positive scalar in the range of  $[10^{-7}, 10^{-3}]$ . Units are dimensionless. Resolution cells are determined from the AzimuthResolution and RangeResolution properties, and the ElevationResolution and RangeRateResolution properties when they are enabled.

Example: 1e-5

Data Types: double

**AzimuthResolution — Azimuth resolution**

1 (default) | positive 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 two targets. The azimuth resolution is typically the 3-dB downpoint of the azimuth angle beamwidth of the radar. Units are in degrees.

Data Types: double

**ElevationResolution — Elevation resolution**

1 (default) | positive 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 two targets. The elevation resolution is typically the 3dB-downpoint in elevation angle beamwidth of the radar. Units are in degrees.

**Dependencies**

To enable this property, set the HasElevation property to true.

Data Types: double

**AzimuthBiasFraction — Azimuth bias fraction**

0 .1 (default) | nonnegative scalar

Azimuth bias fraction of the radar, specified as a nonnegative scalar. The azimuth bias is expressed as a fraction of the azimuth resolution specified in AzimuthResolution. This value sets a lower bound on the azimuthal accuracy of the radar. This value is dimensionless.

Data Types: double

**ElevationBiasFraction — Elevation bias fraction**

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 value of the `ElevationResolution` property. This value sets a lower bound on the elevation accuracy of the radar. This value is dimensionless.

### **Dependencies**

To enable this property, set the `HasElevation` property to `true`.

Data Types: double

### **HasINS — Enable inertial navigation system (INS) input**

`false` (default) | `true`

Enable the optional input argument that passes the current estimate of the sensor platform pose to the sensor, specified as `false` or `true`. When `true`, pose information is added to the `MeasurementParameters` structure of the reported detections. Pose information lets tracking and fusion algorithms estimate the state of the target detections in the north-east-down (NED) frame.

Data Types: logical

### **HasNoise — Enable addition of noise to sensor measurements**

`true` (default) | `false`

Enable addition of noise to sensor measurements, specified as `true` or `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 object still computes the `MeasurementNoise` property of each detection.

Data Types: logical

### **HasFalseAlarms — Enable creating false alarm detections**

`true` (default) | `false`

Enable creating false alarm measurements, specified as `true` or `false`. Set this property to `true` to report false alarms. Otherwise, only actual detections are reported.

Data Types: logical

### **MaxNumDetectionsSource — Source of maximum number of detections reported**

`'Auto'` (default) | `'Property'`

Source of maximum number of detections reported by the sensor, specified as `'Auto'` or `'Property'`. When this property is set to `'Auto'`, the sensor reports all detections. When this property is set to `'Property'`, the sensor reports up to the number of detections specified by the `MaxNumDetections` property.

Data Types: char

### **MaxNumDetections — Maximum number of reported detections**

`50` (default) | positive integer

Maximum number of detections reported by the sensor, specified as a positive integer. If the `DetectionMode` is set to `'monostatic'` or `'bistatic'`, detections are reported in order of distance to the sensor until the maximum number is reached. If the `DetectionMode` is set to `'ESM'`, detections are reported from highest SNR to lowest SNR.

**Dependencies**

To enable this property, set the `MaxNumDetectionsSource` property to 'Property'.

Data Types: double

**HasOcclusion — Enable occlusion from extended objects**

true (default) | false

Enable occlusion from extended objects, specified as `true` or `false`. Set this property 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 this property 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

**DetectionCoordinates — Coordinate system of reported detections**

'Scenario' | 'Body' | 'Sensor rectangular' | 'Sensor spherical'

Coordinate system of reported detections, specified as:

- 'Scenario' — Detections are reported in the rectangular scenario coordinate frame. The scenario coordinate system is defined as the local NED 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 can not be changed.

Example: 'Sensor spherical'

Data Types: char

**ESM and Bistatic Sensor Properties****MountingLocation — Sensor location on platform**

[0 0 0] (default) | 1-by-3 real-valued vector

Sensor location on platform, specified as a 1-by-3 real-valued vector. This property defines the coordinates of the sensor with respect to the platform origin. The default value specifies that the sensor origin is at the origin of its platform. Units are in meters.

Example: [ .2 0.1 0]

#### **Dependencies**

To enable this property, set the **DetectionMode** property to 'ESM' or 'bistatic'.

Data Types: double

#### **MountingAngles — Orientation of sensor**

[0 0 0] (default) | 3-element real-valued vector

Orientation of the sensor 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 sensor axes. The three elements define the rotations around the z-, y-, and x-axes, 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 the frame around the carried x-axis. Units are in degrees.

Example: [10 20 -15]

#### **Dependencies**

To enable this property, set the **DetectionMode** property to 'ESM' or 'bistatic'.

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 field of view **azfov** must lie in the interval (0,360]. The elevation field 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 sensor scans mechanically across the azimuth and elevation limits specified by the <b>MechanicalScanLimits</b> property. 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 <b>ElectronicScanLimits</b> property. 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 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 sensor beam points along the antenna boresight defined by the <b>mountingAngles</b> property.

Example: 'No scanning'

### Dependencies

To enable this property, set the **DetectionMode** property to 'ESM' or 'bistatic'.

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 sensor can mechanically scan. The sensor 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', and set the `DetectionMode` property to 'ESM' or 'bistatic'.

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', and set the `DetectionMode` property to 'ESM' or 'bistatic'.

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', and set the `DetectionMode` property to 'ESM' or 'bistatic'.

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`', and set the `DetectionMode` property to '`ESM`' or '`bistatic`'.

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`', and set the `DetectionMode` property to '`ESM`' or '`bistatic`'.

Data Types: `double`

#### **LookAngle — Look angle of sensor**

scalar | real-valued 2-by-1 vector

This property is read-only.

Look angle of sensor, 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.

ScanMode	LookAngle
'Mechanical'	MechanicalAngle
'Electronic'	ElectronicAngle
'Mechanical and Electronic'	MechanicalAngle + ElectronicAngle
'No scanning'	0

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.

#### Dependencies

To enable this property, set the `DetectionMode` property to '`ESM`' or '`bistatic`'.

**CenterFrequency — Center frequency of radar band**

positive scalar

Center frequency of radar band, specified as a positive scalar. Units are in hertz.

Example: 100e6

**Dependencies**To enable this property, set the **DetectionMode** property to 'ESM' or 'bistatic'.

Data Types: double

**Bandwidth — Radar waveform bandwidth**

positive scalar

Radar waveform bandwidth, specified as a positive scalar. Units are in hertz.

Example: 100e3

**Dependencies**To enable this property, set the **DetectionMode** property to 'ESM' or 'bistatic'.

Data Types: double

**WaveformTypes — Types of detected waveforms**0 (default) | nonnegative integer-valued  $L$ -element vectorTypes of detected waveforms, specified as a nonnegative integer-valued  $L$ -element vector.

Example: [1 4 5]

**Dependencies**To enable this property, set the **DetectionMode** property to 'ESM' or 'bistatic'.

Data Types: double

**ConfusionMatrix — Probability of correct classification of detected waveform**positive scalar | real-valued nonnegative  $L$ -element vector | real-valued nonnegative  $L$ -by- $L$  matrix

Probability of correct classification of a detected waveform, specified as a positive scalar, a real-valued nonnegative  $L$ -element vector, or a real-valued nonnegative  $L$ -by- $L$  matrix. Matrix values lie from 0 through 1 and matrix rows must sum to 1.  $L$  is the number of waveform types detectable by the sensor, as indicated by the value set in the **WaveformTypes** property. The  $(i,j)$  matrix element represents the probability of classifying the  $i$ th waveform as the  $j$ th waveform. When specified as a scalar from 0 through 1, the value is expanded along the diagonal of the confusion matrix. When specified as a vector, it must have the same number of elements as the **WaveformTypes** property. When defined as a scalar or a vector, the off diagonal values are set to  $(1-\text{val})/(L-1)$ .

**Dependencies**To enable this property, set the **DetectionMode** property to 'ESM' or 'bistatic'.

Data Types: double

**Monostatic and Bistatic Sensor Properties****RangeResolution — Range resolution of radar**

100 (default) | positive scalar

Range resolution of the radar, specified as a positive scalar. The range resolution defines the minimum separation in range at which the radar can distinguish between two targets. Units are in meters.

**Dependencies**

To enable this property, set the `DetectionMode` property to '`monostatic`' or '`bistatic`'.

Data Types: double

**RangeRateResolution — Range rate resolution of radar**

10 (default) | positive scalar

Range rate resolution of the radar, specified as a positive 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 property, set the `HasRangeRate` property to `true`, and set the `DetectionMode` property to '`monostatic`' or '`bistatic`'.

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 in `RangeResolution`. This property sets a lower bound on the range accuracy of the radar. This value is dimensionless.

**Dependencies**

To enable this property, set the `DetectionMode` property to '`monostatic`' or '`bistatic`'.

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 in `RangeRateResolution`. This property sets a lower bound on the range-rate accuracy of the radar. This value is dimensionless.

**Dependencies**

To enable this property, set the `HasRangeRate` property to `true`, and set the `DetectionMode` property to '`monostatic`' or '`bistatic`'.

Data Types: double

**HasRangeRate — Enable radar to measure range rate**

false (default) | true

Enable the radar to measure target range rates, specified as `false` or `true`. Set this property to `true` to model a radar sensor that can measure target range rate. Set this property to `false` to model a radar sensor that cannot measure range rate.

## Dependencies

To enable this property, set the `DetectionMode` property to 'ESM' or 'bistatic'.

Data Types: logical

### **HasRangeAmbiguities — Enable range ambiguities**

false (default) | true

Enable range ambiguities, specified as `false` or `true`. Set this property to `true` to enable range ambiguities by the sensor. In this case, the sensor cannot resolve range ambiguities for targets at ranges beyond the `MaxUnambiguousRange` are wrapped into the interval  $[0, \text{MaxUnambiguousRange}]$ . When `false`, targets are reported at their unambiguous range.

## Dependencies

To enable this property, set the `DetectionMode` property to 'ESM' or 'bistatic'.

Data Types: logical

### **HasRangeRateAmbiguities — Enable range-rate ambiguities**

false (default) | true

Enable range-rate ambiguities, specified as `false` or `true`. Set to `true` to enable range-rate ambiguities by the sensor. When `true`, the sensor does not resolve range rate ambiguities and target range rates beyond the `MaxUnambiguousRadialSpeed` are wrapped into the interval  $[0, \text{MaxUnambiguousRadialSpeed}]$ . When `false`, targets are reported at their unambiguous range rate.

## Dependencies

To enable this property, set the `HasRangeRate` property to `true` and set the `DetectionMode` property to 'ESM' or 'bistatic'.

Data Types: logical

### **MaxUnambiguousRange — Maximum unambiguous detection range**

100e3 (default) | positive scalar

Maximum unambiguous range, specified as a positive scalar. 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 applies to true target detections when you set the `HasRangeAmbiguities` property to `true`.

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 for false alarms.

Units are in meters.

Example: 5e3

## Dependencies

To enable this property, set the `HasRangeAmbiguities` property or the `HasFalseAlarms` property to `true`. Meanwhile, set the `DetectionMode` property to 'ESM' or 'bistatic'.

Data Types: double

**MaxUnambiguousRadialSpeed — Maximum unambiguous radial speed**

200 (default) | positive scalar

Maximum unambiguous radial speed, specified as a positive scalar. 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 `[-MaxUnambiguousRadialSpeed, MaxUnambiguousRadialSpeed]`. This property applies to true target detections when you set `HasRangeRateAmbiguities` property to `true`.

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 for which false alarms can be generated.

Units are in meters per second.

**Dependencies**

To enable this property, set `HasRangeRate` and `HasRangeRateAmbiguities` to `true` and/or set `HasRangeRate` and `HasFalseAlarms` to `true`. Meanwhile, set the `DetectionMode` property to '`ESM`' or '`bistatic`'.

Data Types: double

## Usage

### Syntax

```
dets = sensor(radarsigs,simTime)
dets = sensor(radarsigs,txconfigs,simTime)
dets = sensor(___,ins,simTime)
[dets,numDets,config] = sensor(___)
```

### Description

`dets = sensor(radarsigs,simTime)` creates ESM or bistatic radar detections, `dets`, from radar emissions, `radarsigs`, at the current simulation time, `simTime`. The sensor generates detections at the rate defined by the `UpdateRate` property. To use this syntax, set `ScanMode` property to '`ESM`' or '`bistatic`'.

`dets = sensor(radarsigs,txconfigs,simTime)` also specifies emitter configurations, `txconfigs`, of the monostatic sensor at the current simulation time. To use this syntax, set `ScanMode` property to '`Monostatic`'.

`dets = sensor(___,ins,simTime)` also specifies the inertial navigation system (INS) estimated sensor platform pose, `ins`. INS information is used by tracking and fusion algorithms to estimate the target positions in the NED frame.

To use this syntax, set the `HasINS` property to `true`.

`[dets,numDets,config] = sensor(___)` also returns the number of valid detections reported, `numDets`, and the configuration of the sensor, `config`, at the current simulation time.

## Input Arguments

### **radarsigs — Radar emissions**

array of radar emission objects

Radar emissions, specified as an array or a cell array of `radarEmission` objects.

### **txconfigs — Emitter configurations**

array of structures

Emitter configurations, specified as an array of structures. This array must contain the configuration of the `radarEmitter` whose `EmitterIndex` matches the value of the `EmitterIndex` property of the `radarSensor`. Each structure has these fields:

Field	Description
<code>EmitterIndex</code>	Unique emitter index
<code>IsValidTime</code>	<code>IsValidTime</code> is 0 when emitter updates are requested at times that are between update intervals specified by <code>UpdateInterval</code> .
<code>IsScanDone</code>	<code>IsScanDone</code> is <code>true</code> when the emitter has completed a scan.
<code>FieldOfView</code>	Field of view of emitter.
<code>MeasurementParameters</code>	<code>MeasurementParameters</code> 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 3-112.

Data Types: `struct`

### **ins — Platform pose from INS**

structure

Sensor platform pose obtained from the inertial navigation system (INS), specified as a structure. The INS information can be used by tracking and fusion algorithms to estimate the platform's pose and velocity in the NED frame.

Platform pose information from an inertial navigation system (INS) is a structure with these fields:

Field	Definition
<code>Position</code>	Position in the navigation frame, specified as a real-valued 1-by-3 vector. Units are in meters.
<code>Velocity</code>	Velocity in the navigation frame, specified as a real-valued 1-by-3 vector. Units are in meters per second.

Orientation	Orientation with respect to the navigation 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.
-------------	---

**Dependencies**

To enable this argument, set the `HasINS` property to `true`.

Data Types: struct

**simTime — Current simulation time**

nonnegative scalar

Current simulation time, specified as a positive scalar. The `trackingScenario` object calls the scan radar sensor at regular time intervals. The radar sensor generates new detections 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 sensor between update intervals contain no detections. Units are in seconds.

Example: 10.5

Data Types: double

**Output Arguments****dets — sensor detections**

cell array of `objectDetection` objects

Sensor detections, returned as a cell array of `objectDetection` objects. Each object has 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

`Measurement` and `MeasurementNoise` are reported in the coordinate system specified by the `DetectionCoordinates` property. For details on `Measurement`, `MeasurementParameters`, and `ObjectAttributes` of `radarSensor`, please see “Object Detections” on page 3-111.

**numDets — Number of detections**

nonnegative integer

Number of detections reported, returned as a nonnegative integer.

- When the `MaxNumDetectionsSource` property is set to 'Auto', `numDets` is set to the length of `dets`.
- When the `MaxNumDetectionsSource` property is set to 'Property', `dets` is a cell array with length determined by the `MaxNumDetections` property. No more than `MaxNumDetections` number of detections are returned. If the number of detections is fewer than `MaxNumDetections`, the first `numDets` elements of `dets` hold valid detections. The remaining elements of `dets` are set to the default value.

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
<code>SensorIndex</code>	Unique sensor index, returned as a positive integer.
<code>IsValidTime</code>	<code>IsValidTime</code> is <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>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.

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 radarSensor**

<code>coverageConfig</code>	Sensor and emitter coverage configuration
<code>perturbations</code>	Perturbation defined on object
<code>perturb</code>	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

### Detect Radar Emission with ESM Sensor

Create an radar emission and then detect the emission using a `radarSensor` object.

First, create an 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 `radarSensor`.

```
sensor = radarSensor(1);
```

Detect the RF emission.

```
time = 0;
[dets,numDets,config] = sensor(rfSig,time)

dets = 1x1 cell array
    {1x1 objectDetection}

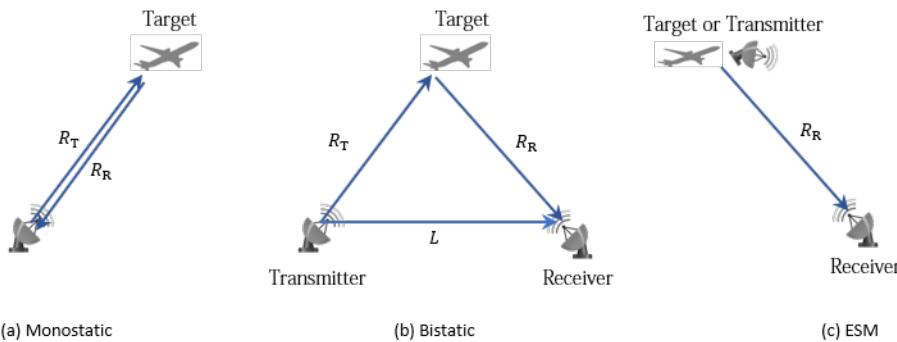
numDets = 1

config = struct with fields:
    SensorIndex: 1
    IsValidTime: 1
    IsScanDone: 0
    FieldOfView: [1 5]
    MeasurementParameters: [1x1 struct]
```

## More About

### Radar Sensor Detection Modes

The `radarSensor` 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 also 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 eventually 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 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 azimuth and elevation of the target or transmitter. These measurements can only be reported in a spherical coordinate system.

## Object Detections

### Measurements

The sensor measures the coordinates of the target. The `Measurement` and `MeasurementNoise` values are reported in the coordinate system specified by the `DetectionCoordinates` property of the sensor.

When the `DetectionCoordinates` property is '`Scenario`', '`Body`', or '`Sensor rectangular`', the `Measurement` and `MeasurementNoise` values are reported in rectangular coordinates. Velocities are only reported when the range rate property, `HasRangeRate`, is `true`.

When the `DetectionCoordinates` property is '`Sensor spherical`', the `Measurement` and `MeasurementNoise` values are reported in a spherical coordinate system. Measurements are ordered as [azimuth, elevation, range, range rate]. Angles are in degrees, range is in meters, and range rate is in meters per second. Elevation and range rate are only reported when `HasElevation` and `HasRangeRate` are `true`.

Note:

- When the **DetectionMode** is set to 'ESM' or 'bistatic', the detections can only be reported in 'Sensor spherical' coordinate system.
- When the **DetectionMode** is set to 'monostatic', the reported 'range' is the range measurement from the target to the radar sensor.
- When the **DetectionMode** is set to 'bistatic', the reported 'range' is the bistatic range measurement (see "Radar Sensor Detection Modes" on page 3-110).

### Measurement Coordinates

<b>DetectionCoordinates</b>	<b>Measurement and Measurement Noise Coordinates</b>																
'Scenario'	<b>Coordinate Dependence on HasRangeRate</b>																
'Body'	<table border="1"> <thead> <tr> <th><b>HasRangeRate</b></th><th><b>Coordinates</b></th></tr> </thead> <tbody> <tr> <td>true</td><td>[x; y; z; vx; vy; vz]</td></tr> <tr> <td>false</td><td>[x; y; z]</td></tr> </tbody> </table>		<b>HasRangeRate</b>	<b>Coordinates</b>	true	[x; y; z; vx; vy; vz]	false	[x; y; z]									
<b>HasRangeRate</b>	<b>Coordinates</b>																
true	[x; y; z; vx; vy; vz]																
false	[x; y; z]																
'Sensor rectangular'	<b>Coordinate for 'monostatic' or 'bistatic' Detection Mode (Dependence on HasRangeRate and HasElevation )</b> <table border="1"> <thead> <tr> <th><b>HasRangeRate</b></th><th><b>HasElevation</b></th><th><b>Coordinates</b></th></tr> </thead> <tbody> <tr> <td>true</td><td>true</td><td>[az; el; rng; rr]</td></tr> <tr> <td>true</td><td>false</td><td>[az; rng; rr]</td></tr> <tr> <td>false</td><td>true</td><td>[az; el; rng]</td></tr> <tr> <td>false</td><td>false</td><td>[az; rng]</td></tr> </tbody> </table>		<b>HasRangeRate</b>	<b>HasElevation</b>	<b>Coordinates</b>	true	true	[az; el; rng; rr]	true	false	[az; rng; rr]	false	true	[az; el; rng]	false	false	[az; rng]
<b>HasRangeRate</b>	<b>HasElevation</b>	<b>Coordinates</b>															
true	true	[az; el; rng; rr]															
true	false	[az; rng; rr]															
false	true	[az; el; rng]															
false	false	[az; rng]															
'Sensor spherical'	<b>Coordinate for 'ESM' Detection Mode (Dependence on HasElevation )</b> <table border="1"> <thead> <tr> <th><b>HasElevation</b></th><th><b>Coordinates</b></th></tr> </thead> <tbody> <tr> <td>true</td><td>[az; el]</td></tr> <tr> <td>false</td><td>[az]</td></tr> </tbody> </table>		<b>HasElevation</b>	<b>Coordinates</b>	true	[az; el]	false	[az]									
<b>HasElevation</b>	<b>Coordinates</b>																
true	[az; el]																
false	[az]																

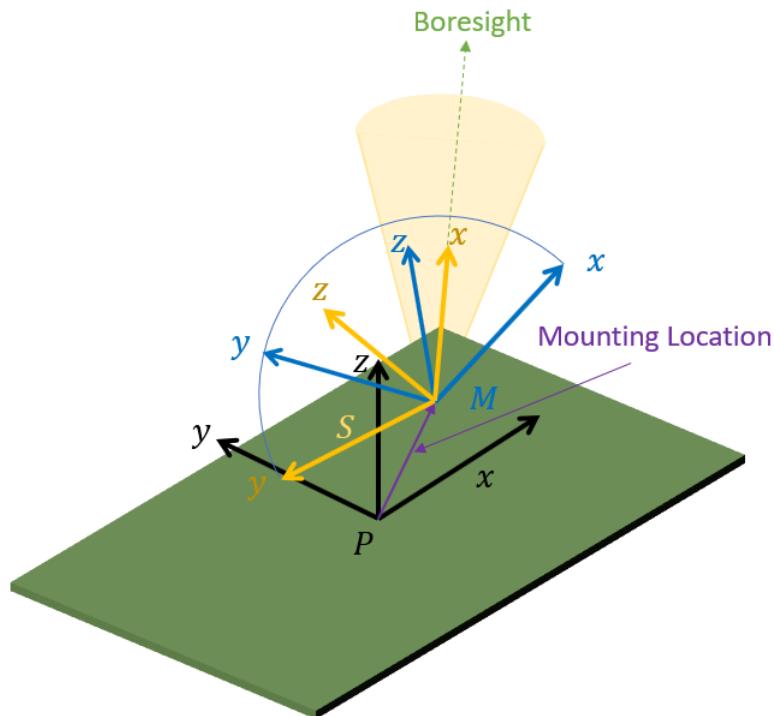
where az, el, rng and rr represent azimuth angle, elevation angle, range and range rate, respectively.

### Measurement Parameters

The **MeasurementParameters** property consists of an array of structures that describe a sequence of coordinate transformations from a child frame to a parent frame or the inverse transformations (see "Frame Rotation"). 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 the 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 form the scenario frame to the platform frame then from 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.

Trivially, if the detections are reported in platform rectangular coordinates and **HasINS** is set to false, the transformation consists only of the identity.

The fields of **MeasurementParameters** are shown here. Not all fields have to be present in the structure. The 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, <b>Frame</b> is set to 'rectangular'. When detections are reported in spherical coordinates, <b>Frame</b> is set 'spherical' for the first struct.
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 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, and 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 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].

### Object Attributes

Object attributes contain additional information about a detection.

Attribute	Description
TargetIndex	Identifier of the platform, PlatformID, that generated the detection. For false alarms, this value is negative.
EmitterIndex	Index of the emitter from which the detected signal was emitted.
SNR	Detection signal-to-noise ratio in dB.
CenterFrequency	<ul style="list-style-type: none"> <li>Measured center frequency of the detected radar signal. Units are in Hz.</li> <li>This attribute is present only when the DetectionMode property is set to 'ESM' or 'Bistatic'.</li> </ul>

Bandwidth	<ul style="list-style-type: none"> <li>Measured bandwidth of the detected radar signal, Units are in Hz.</li> <li>This attribute is present only when the DetectionMode property is set to 'ESM' or 'Bistatic'.</li> </ul>
WaveformType	<ul style="list-style-type: none"> <li>Identifier of the waveform type that was classified by the ESM sensor for the detected signal.</li> <li>This attribute is present only when the DetectionMode property is set to 'ESM' or 'Bistatic'.</li> </ul>

## 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.

## References

- [1] Doerry, A. W. "Earth curvature and atmospheric refraction effects on radar signal propagation." *Sandia Report* . SAND 2012-10690, 2013.
- [2] Doerry, A. W. "Motion Measurement for Synthetic Aperture Radar." *Sandia Report* . SAND 2015-20818, 2015.

## 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

### Objects

[objectDetection](#) | [radarEmission](#)

### Functions

[targetPoses](#)

### System Objects

[monostaticRadarSensor](#) | [trackerGNN](#) | [trackerTOMHT](#)

## Introduced in R2018b

# irSensor

Generate infrared detections for tracking scenario

## Description

The `irSensor` System object creates a statistical model for generating detections using infrared sensors. You can use the `irSensor` object in a scenario that models moving and stationary platforms using `trackingScenario`. The sensor can simulate real detections with added random noise and also generate false alarm detections. In addition, you can use this object to create input to trackers such as `trackerGNN`, `trackerJPDA`, or `trackerTOMHT`.

This object enables you to configure a mechanically scanning sensor. An infrared scanning sensor changes the look angle between updates by stepping the mechanical position of the beam in increments of the angular span specified in the `FieldOfView` property. The infrared sensor scans the total region in azimuth and elevation defined by the `MechanicalScanLimits` property. If the scanning limits for azimuth or elevation are set to `[0 0]`, no scanning is performed along that dimension for that scan mode. Also, if the maximum scan rate for azimuth or elevation is set to zero, no scanning is performed along that dimension.

To generate infrared detections:

- 1** Create the `irSensor` 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

```
sensor = irSensor(SensorIndex)
sensor = irSensor(SensorIndex, 'No scanning')
sensor = irSensor(SensorIndex, 'Raster')
sensor = irSensor(SensorIndex, 'Rotator')
sensor = irSensor(SensorIndex, 'Sector')

sensor = irSensor(____, Name, Value)
```

### Description

`sensor = irSensor(SensorIndex)` creates an infrared detection generator object with a specified sensor index, `SensorIndex`, and default property values.

`sensor = irSensor(SensorIndex, 'No scanning')` is a convenience syntax that creates an `irSensor` that stares along the sensor boresight direction. No mechanical scanning is performed. This syntax sets the `ScanMode` property to '`No scanning`'.

`sensor = irSensor(SensorIndex, 'Raster')` is a convenience syntax that creates an `irSensor` object that mechanically scans a raster pattern. The raster span is 90° in azimuth from -45° to +45° and in elevation from the horizon to 10° above the horizon. See “Convenience Syntaxes” on page 3-129 for the properties set by this syntax.

`sensor = irSensor(SensorIndex, 'Rotator')` is a convenience syntax that creates an `irSensor` object that mechanically scans 360° in azimuth by electronically rotating the sensor at a constant rate. When you set `HasElevation` to `true`, the infrared sensor mechanically points towards the center of the elevation field of view. See “Convenience Syntaxes” on page 3-129 for the properties set by this syntax.

`sensor = irSensor(SensorIndex, 'Sector')` is a convenience syntax to create an `irSensor` object that mechanically scans a 90° azimuth sector from -45° to +45°. Setting `HasElevation` to `true`, points the infrared sensor towards the center of the elevation field of view. Beams are stacked mechanically to process the entire elevation spanned by the scan limits in a single dwell. See “Convenience Syntaxes” on page 3-129 for the properties set by this syntax.

`sensor = irSensor(____, Name, Value)` sets properties using one or more name-value pairs after all other input arguments. Enclose each property name in quotes. For example, `irSensor(1, 'UpdateRate', 1, 'CutoffFrequency', 20e3)` creates an infrared sensor that reports detections at an update rate of 1 Hz and a cut off frequency of 20 kHz. If you specify the sensor index using the `SensorIndex` property, you can omit the `SensorIndex` 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.

### **SensorIndex — Unique sensor identifier**

positive integer

Unique sensor identifier, specified as a positive integer. This property distinguishes detections that come from different sensors in a multi-sensor system. When creating an `irSensor` system object, you must either specify the `SensorIndex` as the first input argument in the creation syntax, or specify it as the value for the `SensorIndex` property in the creation syntax.

Example: 2

Data Types: double

### **UpdateRate — Sensor update rate**

1 (default) | positive scalar

Sensor update rate, specified as a positive scalar. This interval must be an integer multiple of the simulation time interval defined by `trackingScenario`. The `trackingScenario` object calls the infrared sensor at simulation time intervals. The sensor generates new detections at intervals defined by the reciprocal of the `UpdateRate` property. Any update requested to the sensor between update intervals contains no detections. Units are in hertz.

Example: 5

Data Types: double

**ScanMode — Scanning mode of infrared sensor**

[*Mechanical*] (default) | 'No scanning'

Scanning mode of infrared sensor, specified as 'Mechanical' or 'No scanning'. When set to 'Mechanical', the sensor scans mechanically across the azimuth and elevation limits specified by the MechanicalScanLimits property. The scan positions step by the sensor's field of view between dwells. When set to 'No scanning', no scanning is performed by the sensor.

Example: 'No scanning'

Data Types: char

**MountingLocation — Sensor location on platform**

[0 0 0] (default) | 1-by-3 real-valued vector

Sensor location on platform, specified as a 1-by-3 real-valued vector. This property defines the coordinates of the sensor with respect to the platform origin. The default value specifies that the sensor origin is at the origin of its platform. Units are in meters.

Example: [.2 0.1 0]

Data Types: double

**MountingAngles — Orientation of sensor**

[0 0 0] (default) | 3-element real-valued vector

Orientation of the sensor 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 sensor axes. The three elements describes the rotations around the *z*-, *y*-, and *x*-axes sequentially. Units are in degrees.

Example: [10 20 -15]

Data Types: double

**FieldOfView — Fields of view of sensor**

[1;5] | real-valued 2-by-1 vector of positive real-values

This property is read-only.

Fields of view of sensor, specified as a 2-by-1 vector of positive real values, [azfov;elfov]. The field of view defines the total angular extent spanned by the sensor. Each component must lie in the interval (0,180]. Targets outside of the field of view of the sensor will not be detected. Units are in degrees.

Example: [14;70]

Data Types: double

**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 infrared sensor can mechanically scan. The sensor sets its scan rate to step the 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'.

Data Types: double

#### MechanicalScanLimits — Angular limits of mechanical scan directions of sensor

[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 sensor, 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 sensor 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: [10 90;0 85]

#### Dependencies

To enable this property, set the `ScanMode` property to 'Mechanical'.

Data Types: double

#### MechanicalAngle — Current mechanical scan angle

scalar | real-valued 2-by-1 vector

This property is read-only.

Current mechanical scan angle, 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 sensor on the platform. When `HasElevation` is `false`, the scan angle is a scalar representing the azimuth scan angle.

Data Types: double

#### LookAngle — Look angle of sensor

scalar | real-valued 2-by-1 vector

This property is read-only.

Look angle of sensor, specified as a scalar or real-valued 2-by-1 vector. Look angle depends on the mechanical angle set in the `ScanMode` property.

ScanMode	LookAngle
'Mechanical'	MechanicalAngle
'No scanning'	0

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.

#### **LensDiameter — Lens diameter**

8.0e-2 (default) | positive scalar

Lens diameter, specified as a positive scalar. Units are in meters.

Example: 0.1

Data Types: double

#### **FocalLength — Focal length of sensor circular lens**

800 (default) | scalar

Focal length of sensor circular lens, specified as a scalar. The focal length in pixels is  $f = F s$ , where  $F$  is the focal length in millimeters and  $s$  is the number of pixels per millimeter.

Example: 500

Data Types: double

#### **NumDetectors — Number of infrared detectors in sensor imaging plane**

[1000 1000] | positive, real-valued, two-element vector

Number of infrared detectors in the sensor imaging plane, specified as a positive, real-valued, two-element row vector. The first element defines the number of rows in the imaging plane and the second element defines the number of columns in the imaging plane. The number of rows corresponds to the sensor elevation resolution and the number of columns corresponds to the sensor azimuth resolution.

Example: [500 750]

Data Types: double

#### **CutoffFrequency — Cut off frequency of sensor modulation transfer function**

20e3 | positive scalar

Cut off frequency of the sensor modulation transfer function (MTF), specified as a positive scalar. Units are in hertz.

Example: 30.5e3

#### **Dependencies**

To enable this property, set the `ScanMode` property to 'Mechanical'.

Data Types: double

#### **DetectorArea — Area of infrared detector element**

1.44e-6 | positive scalar

Area of an infrared detector element/pixel, specified as a positive scalar. Units are in square-meters.

Example: 3.0e-5

Data Types: double

#### **Detectivity — Specific detectivity of detector material**

1.2e10 | positive scalar

Specific detectivity of the detector material, specified as a positive scalar. Units are cm-sqrt(Hz)/W.

Example: .9e10

Data Types: double

#### **NoiseEquivalentBandwidth — Noise equivalent bandwidth of sensor**

30 (default) | positive scalar

Noise equivalent bandwidth of sensor, specified as a positive scalar. Units are in Hz.

Example: 100

Data Types: double

#### **FalseAlarmRate — False alarm rate**

1e-6 (default) | positive scalar

Rate of false alarm report in each resolution cell, specified as a positive scalar in the range of  $[10^{-7}, 10^{-3}]$ . Units are dimensionless. Resolution cells are determined from the AzimuthResolution property and the optionally enabled ElevationResolution property.

Example: 1e-5

Data Types: double

#### **AzimuthResolution — Azimuth resolution**

1 (default) | positive scalar

This property is read-only.

Azimuth resolution of the sensor, specified as a positive scalar. The azimuth resolution defines the minimum separation in azimuth angle at which the sensor can distinguish two targets. The azimuth resolution is derived from the focal length of the lens and the number of columns in the detector's imaging plane. Units are in degrees.

Data Types: double

#### **ElevationResolution — Elevation resolution of sensor**

1 (default) | positive scalar

This property is read-only.

Elevation resolution of the sensor, specified as a positive scalar. The elevation resolution defines the minimum separation in elevation angle at which the sensor can distinguish two targets. The elevation resolution is derived from the focal length of the lens and the number of rows in the detector's imaging plane. Units are in degrees.

#### **Dependencies**

To enable this property, set the HasElevation property to true.

Data Types: double

**AzimuthBiasFraction — Azimuth bias fraction**

0.1 (default) | nonnegative scalar

Azimuth bias fraction of the sensor, specified as a nonnegative scalar. The azimuth bias is expressed as a fraction of the azimuth resolution specified in **AzimuthResolution**. This value sets a lower bound on the azimuthal accuracy of the sensor. This property only applies for modes where the sensor is scanning. The value is dimensionless.

Data Types: double

**ElevationBiasFraction — Elevation bias fraction**

0.1 (default) | nonnegative scalar

Elevation bias fraction of the sensor, specified as a nonnegative scalar. Elevation bias is expressed as a fraction of the elevation resolution specified by the value of the **ElevationResolution** property. This value sets a lower bound on the elevation accuracy of the sensor. This property only applies for modes where the sensor is scanning. The value is dimensionless.

**Dependencies**

To enable this property, set the **HasElevation** property to **true**.

Data Types: double

**HasElevation — Enable sensor elevation scan and measurements**

false (default) | true

Enable the sensor to measure target elevation angles and to scan in elevation, specified as **false** or **true**. Set this property to **true** to model an infrared sensor that can estimate target elevation and scan in elevation.

Data Types: logical

**HasAngularSize — Enable angular size measurements**

false (default) | true

Enable the sensor to return the azimuth and elevation size or span of the target in the reported detections, specified as **false** or **true**. If this property is set to **false**, then the only azimuth and elevation locations instead of their angular extent are reported in the detections.

Data Types: logical

**HasINS — Enable inertial navigation system (INS) input**

false (default) | true

Enable the optional input argument that passes the current estimate of the sensor platform pose to the sensor, specified as **false** or **true**. When **true**, pose information is added to the **MeasurementParameters** structure of the reported detections. Pose information lets tracking and fusion algorithms estimate the state of the target detections in the north-east-down (NED) frame.

Data Types: logical

**HasNoise — Enable addition of noise to sensor measurements**

true (default) | false

Enable addition of noise to sensor measurements, specified as **true** or **false**. Set this property to **true** to add noise to the measurements. Otherwise, the measurements have no noise. Note that the

reported measurement noise covariance is not dependent on this property and is always representative of the noise that will be added when `HasNoise` is set to true.

Data Types: logical

**HasFalseAlarms — Enable creating false alarm sensor detections**

true (default) | false

Enable creating false alarm sensor measurements, specified as `true` or `false`. Set this property to `true` to report false alarms. Otherwise, only actual detections are reported.

Data Types: logical

**HasOcclusion — Enable occlusion from extended objects**

true (default) | false

Enable occlusion from extended objects, specified as `true` or `false`. Set this property 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 this property 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

**MinClassificationArea — Minimum image size for classification**

100 (default) | positive integer

Minimum image size for classification, specified as a positive integer. `MinClassificationArea` specifies the minimum area (in square pixels) used to decide whether the sensor recognizes the detection as a classified object. The `irSensor` tries to enclose the extent detection using a minimum rectangular bounding box (along the azimuth and elevation directions) in the sensor image plane. If the area of the minimum bounding box is less than the value given by the `MinClassificationArea` property, then the reported `ClassID` is zero in the returned `objectDetection` for that detection. Otherwise, the reported `ClassID` is obtained from the `ClassID` of the corresponding target input.

Data Types: double

**MaxAllowedOcclusion — Maximum allowed occlusion**

0.5 (default) | real scalar in [0,1)

Maximum allowed occlusion, specified as a real scalar on the interval of [0,1). The property specifies the ratio of the occluded area relative to the total area of a target's bounding box. If the occluded area ratio is larger than the value specified by the `MaxAllowedOcclusion` property, the occluded target will not be detected.

Data Types: double

**MaxNumDetectionsSource — Source of maximum number of detections to be reported**

'Auto' (default) | 'Property'

Source of maximum number of detections reported by the sensor, specified as '`Auto`' or '`Property`'. When this property is set to '`Auto`', the sensor reports all detections. When this

property is set to 'Property', the sensor reports detections up to the number specified by the **MaxNumDetections** property.

Data Types: char

#### **MaxNumDetections — Maximum number of reported detections**

50 (default) | positive integer

Maximum number of detections can be reported by the sensor, specified as a positive integer. Detections are reported in order of distance to the sensor until the maximum number is reached.

#### **Dependencies**

To enable this property, set the **MaxNumDetectionsSource** property to 'Property'.

Data Types: double

## **Usage**

### **Syntax**

```
dets = sensor(targets,simTime)
dets = sensor(targets,ins,simTime)
[dets,numDets,config] = sensor(____)
```

#### **Description**

`dets = sensor(targets,simTime)` creates infrared detections, `dets`, from sensor measurements taken of targets at the current simulation time, `simTime`. The sensor can generate detections for multiple targets simultaneously.

`dets = sensor(targets,ins,simTime)` also specifies the INS estimated pose information, `ins`, for the sensor platform. INS information is used by tracking and fusion algorithms to estimate the target positions in the NED frame.

To enable this syntax, set the `HasINS` property to `true`.

`[dets,numDets,config] = sensor(____)` also returns the number of valid detections reported, `numValidDets`, and the configuration of the sensor, `config`, at the current simulation time.

#### **Input Arguments**

##### **targets — Tracking scenario target poses**

structure | structure array

Tracking scenario target poses, specified as a structure or array of structures. Each structure corresponds to a target. You can generate this structure using the `targetPoses` method of a platform. You can also create such a structure manually. The table shows the required fields of the structure:

Field	Description
PlatformID	Unique identifier for the platform, specified as a scalar positive integer. This is a required field with no default value.

Field	Description
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 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 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 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 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 coordinate system.

#### **simTime — Current simulation time**

nonnegative scalar

Current simulation time, specified as a positive scalar. The `trackingScenario` object calls the infrared sensor at regular time intervals. The sensor generates new detections 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 sensor between update intervals contain no detections. Units are in seconds.

Example: 10.5

Data Types: double

#### **ins — Platform pose from INS**

structure

Sensor platform pose obtained from the inertial navigation system (INS), specified as a structure.

Platform pose information from an inertial navigation system (INS) is a structure with these fields:

Field	Definition
Position	Position in the navigation frame, specified as a real-valued 1-by-3 vector. Units are in meters.
Velocity	Velocity in the navigation frame, specified as a real-valued 1-by-3 vector. Units are in meters per second.
Orientation	Orientation with respect to the navigation 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.

### Dependencies

To enable this argument, set the `HasINS` property to `true`.

Data Types: `struct`

### interference — Interfering or jamming signal

structure

Interfering or jamming signal, specified as a structure.

### Dependencies

To enable this argument, set the `HasInterference` property to `true`.

Data Types: `double`

Complex Number Support: Yes

### Output Arguments

#### dets — sensor detections

cell array of `objectDetection` objects

Sensor detections, returned as a cell array of `objectDetection` objects. Each object has 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

`Measurement` and `MeasurementNoise` are reported in the sensor spherical coordinate frame.

**numDets — Number of detections**

nonnegative integer

Number of detections reported, returned as a nonnegative integer.

- When the `MaxNumDetectionsSource` property is set to 'Auto', `numDets` is set to the length of `dets`.
- When the `MaxNumDetectionsSource` property is set to 'Property', `dets` is a cell array with length determined by the `MaxNumDetections` property. No more than `MaxNumDetections` number of detections are returned. If the number of detections is fewer than `MaxNumDetections`, the first `numDets` elements of `dets` hold valid detections. The remaining elements of `dets` are set to the default value.

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 sensor beam during object execution.

Field	Description
<code>SensorIndex</code>	Unique sensor index, returned as a positive integer.
<code>IsValidTime</code>	<code>IsValidTime</code> is <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>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.

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 irSensor

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

### Detection Using Infrared Sensor

Detect a target with an infrared sensor.

First create a target structure.

```
tgt = struct( ...
    'PlatformID',1, ...
    'Position',[10e3 0 0], ...
    'Speed',900*1e3/3600);
```

Then create an IR sensor.

```
sensor = irSensor(1);
```

Generate detection from target.

```
time = 0;
[dets,numDets,config] = sensor(tgt,time)

dets = 1x1 cell array
    {1x1 objectDetection}

numDets = 1

config = struct with fields:
    SensorIndex: 1
    IsValidTime: 1
    IsScanDone: 0
    FieldOfView: [64.0108 64.0108]
    MeasurementParameters: [1x1 struct]
```

## More About

### Convenience Syntaxes

The convenience syntaxes set several properties together to model a specific type of infrared sensor.

### 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]

### 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	$10/\sqrt{12}$

## 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

### Objects

`objectDetection`

### Functions

`targetPoses`

### System Objects

`trackerGNN` | `trackerTOMHT`

**Introduced in R2018b**

# sonarSensor

Generate detections from sonar emissions

## Description

The `sonarSensor` System object creates a statistical model for generating detections from sonar emissions. You can generate detections from active or passive sonar systems. You can use the `sonarSensor` object in a scenario that models moving and stationary platforms using `trackingScenario`. The sonar sensor can simulate real detections with added random noise and also generate false alarm detections. In addition, you can use this object to create input to trackers such as `trackerGNN` or `trackerTOMHT`.

This object enables you to configure an electronically scanning sonar. A scanning sonar changes the look angle between updates by stepping the electronic position of the beam in increments of the angular span specified in the `FieldOfView` property. The sonar scans the total region in azimuth and elevation defined by the sonar electronic scan limits, `ElectronicScanLimits`. If the scanning limits for azimuth or elevation are set to `[0 0]`, no scanning is performed along that dimension for that scan mode. If the maximum electronic scan rate for azimuth or elevation is set to zero, no electronic scanning is performed along that dimension.

To generate sonar detections:

- 1 Create the `sonarSensor` 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

```
sensor = sonarSensor(SensorIndex)  
  
sensor = sonarSensor(SensorIndex, 'No scanning')  
sensor = sonarSensor(SensorIndex, 'Raster')  
sensor = sonarSensor(SensorIndex, 'Rotator')  
sensor = sonarSensor(SensorIndex, 'Sector')  
  
sensor = sonarSensor(____, Name, Value)
```

### Description

`sensor = sonarSensor(SensorIndex)` creates a sonar detection generator object with default property values.

`sensor = sonarSensor(SensorIndex, 'No scanning')` is a convenience syntax that creates a `sonarSensor` that stares along the sonar transducer boresight direction. No electronic scanning is performed. This syntax sets the `ScanMode` property to `'No scanning'`.

`sensor = sonarSensor(SensorIndex, 'Raster')` is a convenience syntax that creates a `sonarSensor` object that electronically scans a raster pattern. The raster span is 90° in azimuth from -45° to +45° and in elevation from the horizon to 10° above the horizon. See “Convenience Syntaxes” on page 3-150 for the properties set by this syntax.

`sensor = sonarSensor(SensorIndex, 'Rotator')` is a convenience syntax that creates a `sonarSensor` object that electronically scans 360° in azimuth by electronically rotating the transducer at a constant rate. When you set `HasElevation` to `true`, the sonar transducer electronically points towards the center of the elevation field of view. See “Convenience Syntaxes” on page 3-150 for the properties set by this syntax.

`sensor = sonarSensor(SensorIndex, 'Sector')` is a convenience syntax to create a `sonarSensor` object that electronically scans a 90° azimuth sector from -45° to +45°. Setting `HasElevation` to `true`, points the sonar transducer towards the center of the elevation field of view. Beams are stacked electronically to process the entire elevation spanned by the scan limits in a single dwell. See “Convenience Syntaxes” on page 3-150 for the properties set by this syntax.

`sensor = sonarSensor(____, Name, Value)` sets properties using one or more name-value pairs after all other input arguments. Enclose each property name in quotes. For example, `sonarSensor('DetectionCoordinates', 'Sensor cartesian', 'MaxRange', 200)` creates a sonar detection generator that reports detections in the sensor Cartesian coordinate system and has a maximum detection range of 200 meters. If you specify the sensor index using the `SensorIndex` property, you can omit the `SensorIndex` 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.

### **SensorIndex — Unique sensor identifier**

positive integer

Unique sensor identifier, specified as a positive integer. This property distinguishes detections that come from different sensors in a multi-sensor system. When creating a `sonarSensor` system object, you must either specify the `SensorIndex` as the first input argument in the creation syntax, or specify it the value for the `SensorIndex` property in the creation syntax.

Example: 2

Data Types: double

### **UpdateRate — Sensor update rate**

1 (default) | positive scalar

Sensor update rate, specified as a positive scalar. This interval must be an integer multiple of the simulation time interval defined by `trackingScenario`. The `trackingScenario` object calls the sonar sensor at simulation time intervals. The sonar generates new detections at intervals defined by the reciprocal of the `UpdateRate` property. Any update requested to the sensor between update intervals contains no detections. Units are in hertz.

Example: 5

Data Types: double

#### **DetectionMode — Detection mode**

'passive' (default) | 'monostatic'

Detection mode, specified as 'passive' or 'monostatic'. When set to 'passive', the sensor operates passively. When set to 'monostatic', the sensor generates detections from reflected signals originating from a collocated sonar emitter.

Example: 'Monostatic'

Data Types: char | string

#### **EmitterIndex — Unique monostatic emitter index**

positive integer

Unique monostatic emitter index, specified as a positive integer. The emitter index identifies the monostatic sonar emitter providing the reference signal to the sensor.

Example: 404

#### **Dependencies**

Set this property when the **DetectionMode** property is set to 'monostatic'.

Data Types: double

#### **MountingLocation — Sensor location on platform**

[0 0 0] (default) | 1-by-3 real-valued vector

Sensor location on platform, specified as a 1-by-3 real-valued vector. This property defines the coordinates of the sensor with respect to the platform origin. The default value specifies that the sensor origin is at the origin of its platform. Units are in meters.

Example: [ .2 0.1 0 ]

Data Types: double

#### **MountingAngles — Orientation of sensor**

[0 0 0] (default) | 3-element real-valued vector

Orientation of the sensor 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 sensor axes. The three elements define the rotations around the z-, y-, and x-axes, 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 the 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 field of view

`azfov` must lie in the interval  $(0, 360]$ . The elevation field of view `elfov` must lie in the interval  $(0, 180]$ .

Example: [14;7]

Data Types: double

#### **ScanMode — Scanning mode of sonar**

'Electronic' (default) | 'No scanning'

Scanning mode of sonar, specified as 'Electronic' or 'No scanning'.

#### **Scan Modes**

ScanMode	Purpose
'Electronic'	The sonar scans electronically across the azimuth and elevation limits specified by the <code>ElectronicScanLimits</code> property. The scan direction increments by the sonar field of view angle between dwells.
'No scanning'	The sonar beam points along the transducer boresight defined by the <code>mountingAngles</code> property.

Example: 'No scanning'

Data Types: char

#### **MechanicalAngle — Current mechanical scan angle**

scalar | real-valued 2-by-1 vector

This property is read-only.

Current mechanical scan angle of sonar, 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 sonar on the platform. When `HasElevation` is `false`, the scan angle is a scalar representing the azimuth scan angle.

Data Types: double

#### **ElectronicScanLimits — Angular limits of electronic scan directions of sonar**

[-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 sonar, 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 sonar 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'.

Data Types: double

#### ElectronicAngle — Current electronic scan angle

electronic scalar | nonnegative scalar

This property is read-only.

Current electronic scan angle of sonar, 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'.

Data Types: double

#### LookAngle — Look angle of sensor

scalar | real-valued 2-by-1 vector

This property is read-only.

Look angle of sensor, specified as a scalar or real-valued 2-by-1 vector. Look angle depends on the electronic angle set in the ScanMode property.

ScanMode	LookAngle
'Electronic'	ElectronicAngle
'No scanning'	0

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.

#### HasElevation — Enable sonar elevation scan and measurements

false (default) | true

Enable the sonar to measure target elevation angles and to scan in elevation, specified as false or true. Set this property to true to model a sonar sensor that can estimate target elevation and scan in elevation.

Data Types: logical

#### CenterFrequency — Center frequency of sonar band

20e3 (default) | positive scalar

Center frequency of sonar band, specified as a positive scalar. Units are in hertz.

Example: 25.5e3

Data Types: double

**Bandwidth — Sonar waveform bandwidth**

2e3 | positive scalar

Sonar waveform bandwidth, specified as a positive scalar. Units are in hertz.

Example: 1.5e3

Data Types: double

**WaveformTypes — Types of detected waveforms**0 (default) | nonnegative integer-valued  $L$ -element vectorTypes of detected waveforms, specified as a nonnegative integer-valued  $L$ -element vector.

Example: [1 4 5]

Data Types: double

**ConfusionMatrix — Probability of correct classification of detected waveform**1 (default) | positive scalar | real-valued nonnegative  $L$ -element vector | real-valued nonnegative  $L$ -by- $L$  matrix

Probability of correct classification of a detected waveform, specified as a positive scalar, a real-valued nonnegative  $L$ -element vector, or a real-valued nonnegative  $L$ -by- $L$  matrix. Matrix values range from 0 through 1 and matrix rows must sum to 1.  $L$  is the number of waveform types that the sensor can detect, as indicated by the value set in the `WaveformTypes` property. The  $(i,j)$  matrix element represents the probability of classifying the  $i^{\text{th}}$  waveform as the  $j^{\text{th}}$  waveform. When specified as a scalar from 0 through 1, the value is expanded along the diagonal of the confusion matrix. When specified as a vector, it must have the same number of elements as the `WaveformTypes` property. When defined as a scalar or a vector, the off diagonal values are set to  $(1-\text{val})/(L-1)$ .

Data Types: double

**AmbientNoiseLevel — Spectrum-level ambient isotropic noise**

70 (default) | scalar

Spectrum-level ambient isotropic noise, specified as a scalar. Units are in dB relative to the intensity of a plane wave with 1  $\mu\text{Pa}$  rms pressure in a 1-hertz frequency band.

Example: 25

Data Types: double

**FalseAlarmRate — False alarm rate**

1e-6 (default) | positive scalar

False alarm report rate within each resolution cell, specified as a positive scalar in the range  $[10^{-7}, 10^{-3}]$ . Units are dimensionless. Resolution cells are determined from the `AzimuthResolution` property and the `ElevationResolution` property when enabled.

Example: 1e-5

Data Types: double

**AzimuthResolution — Azimuth resolution of sonar**

1 (default) | positive scalar

Azimuth resolution of the sonar, specified as a positive scalar. The azimuth resolution defines the minimum separation in azimuth angle at which the sonar can distinguish two targets. The azimuth

resolution is typically the 3-dB downpoint of the azimuth angle beamwidth of the sonar. Units are in degrees.

Data Types: double

**ElevationResolution — Elevation resolution of sonar**

1 (default) | positive scalar

Elevation resolution of the sonar, specified as a positive scalar. The elevation resolution defines the minimum separation in elevation angle at which the sonar can distinguish two targets. The elevation resolution is typically the 3-dB downpoint in the elevation angle beamwidth of the sonar. Units are in degrees.

**Dependencies**

To enable this property, set the `HasElevation` property to `true`.

Data Types: double

**RangeResolution — Range resolution of sonar**

100 (default) | positive scalar

Range resolution of the sonar, specified as a positive scalar. The range resolution defines the minimum separation in range at which the sonar can distinguish between two targets. Units are in meters.

Data Types: double

**RangeRateResolution — Range rate resolution of sonar**

10 (default) | positive scalar

Range rate resolution of the sonar, specified as a positive scalar. The range rate resolution defines the minimum separation in range rate at which the sonar can distinguish between two targets. Units are in meters per second.

**Dependencies**

To enable this property, set the `HasRangeRate` property to `true`.

Data Types: double

**AzimuthBiasFraction — Azimuth bias fraction**

0.1 (default) | nonnegative scalar

Azimuth bias fraction of the sonar, specified as a nonnegative scalar. The azimuth bias is expressed as a fraction of the azimuth resolution specified in `AzimuthResolution`. This value sets a lower bound on the azimuthal accuracy of the sonar. This value is dimensionless.

Data Types: double

**ElevationBiasFraction — Elevation bias fraction**

0.1 (default) | nonnegative scalar

Elevation bias fraction of the sonar, specified as a nonnegative scalar. Elevation bias is expressed as a fraction of the elevation resolution specified by the value of the `ElevationResolution` property. This value sets a lower bound on the elevation accuracy of the sonar. This value 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 sonar, specified as a nonnegative scalar. Range bias is expressed as a fraction of the range resolution specified in `RangeResolution`. This property sets a lower bound on the range accuracy of the sonar. This value is dimensionless.

Data Types: double

**RangeRateBiasFraction — Range rate bias fraction**

`0.05` (default) | nonnegative scalar

Range rate bias fraction of the sonar, specified as a nonnegative scalar. Range rate bias is expressed as a fraction of the range rate resolution specified in `RangeRateResolution`. This property sets a lower bound on the range-rate accuracy of the sonar. This value is dimensionless.

**Dependencies**

To enable this property, set the `HasRangeRate` property to `true`.

Data Types: double

**HasRangeRate — Enable sonar to measure range rate**

`false` (default) | `true`

Enable the sonar to measure target range rates, specified as `false` or `true`. Set this property to `true` to model a sonar sensor that can measure target range rate. Set this property to `false` to model a sonar sensor that cannot measure range rate.

Data Types: logical

**HasRangeAmbiguities — Enable range ambiguities**

`false` (default) | `true`

Enable range ambiguities, specified as `false` or `true`. Set this property to `true` to enable range ambiguities by the sensor. In this case, the sensor cannot resolve range ambiguities for targets at ranges beyond the `MaxUnambiguousRange` are wrapped into the interval  $[0, \text{MaxUnambiguousRange}]$ . When `false`, targets are reported at their unambiguous range.

Data Types: logical

**HasRangeRateAmbiguities — Enable range-rate ambiguities**

`false` (default) | `true`

Enable range-rate ambiguities, specified as `false` or `true`. Set to `true` to enable range-rate ambiguities by the sensor. When `true`, the sensor does not resolve range rate ambiguities and target range rates beyond the `MaxUnambiguousRadialSpeed` are wrapped into the interval  $[0, \text{MaxUnambiguousRadialSpeed}]$ . When `false`, targets are reported at their unambiguous range rate.

### Dependencies

To enable this property, set the `HasRangeRate` property to `true`.

Data Types: `logical`

### **MaxUnambiguousRange — Maximum unambiguous detection range**

`100e3` (default) | positive scalar

Maximum unambiguous range, specified as a positive scalar. Maximum unambiguous range defines the maximum range for which the sonar 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 applies to true target detections when you set the `HasRangeAmbiguities` property to `true`.

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 for false alarms.

Units are in meters.

Example: `5e3`

### Dependencies

To enable this property, set the `HasRangeAmbiguities` property to `true` or set the `HasFalseAlarms` property to `true`.

Data Types: `double`

### **MaxUnambiguousRadialSpeed — Maximum unambiguous radial speed**

`200` (default) | positive scalar

Maximum unambiguous radial speed, specified as a positive scalar. Radial speed is the magnitude of the target range rate. Maximum unambiguous radial speed defines the radial speed for which the sonar 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 applies to true target detections when you set `HasRangeRateAmbiguities` property to `true`.

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 for which false alarms can be generated.

Units are in meters per second.

### Dependencies

To enable this property, set `HasRangeRate` and `HasRangeRateAmbiguities` to `true` and/or set `HasRangeRate` and `HasFalseAlarms` to `true`.

Data Types: `double`

### **HasINS — Enable inertial navigation system (INS) input**

`false` (default) | `true`

Enable the optional input argument that passes the current estimate of the sensor platform pose to the sensor, specified as `false` or `true`. When `true`, pose information is added to the

**MeasurementParameters** structure of the reported detections. Pose information lets tracking and fusion algorithms estimate the state of the target detections in the north-east-down (NED) frame.

Data Types: logical

**HasNoise — Enable addition of noise to sonar sensor measurements**

true (default) | false

Enable addition of noise to sonar sensor measurements, specified as `true` or `false`. Set this property to `true` to add noise to the sonar measurements. Otherwise, the measurements have no noise. Even if you set `HasNoise` to `false`, the object still computes the `MeasurementNoise` property of each detection.

Data Types: logical

**HasFalseAlarms — Enable creating false alarm sonar detections**

true (default) | false

Enable creating false alarm sonar measurements, specified as `true` or `false`. Set this property to `true` to report false alarms. Otherwise, only actual detections are reported.

Data Types: logical

**MaxNumDetectionsSource — Source of maximum number of detections reported**

'Auto' (default) | 'Property'

Source of maximum number of detections reported by the sensor, specified as '`Auto`' or '`Property`'. When this property is set to '`Auto`', the sensor reports all detections. When this property is set to '`Property`', the sensor reports up to the number of detections specified by the `MaxNumDetections` property.

Data Types: char

**MaxNumDetections — Maximum number of reported detections**

50 (default) | positive integer

Maximum number of detections reported by the sensor, specified as a positive integer. Detections are reported in order of distance to the sensor until the maximum number is reached.

**Dependencies**

To enable this property, set the `MaxNumDetectionsSource` property to '`Property`'.

Data Types: double

**DetectionCoordinates — Coordinate system of reported detections**

'Body' (default) | 'Scenario' | 'Sensor rectangular' | 'Sensor spherical'

Coordinate system of reported detections, specified as:

- '`Scenario`' — Detections are reported in the rectangular scenario coordinate frame. The scenario coordinate system is defined as the local NED 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 sonar 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 sonar sensor and aligned with the orientation of the sonar on the platform.

Example: 'Sensor spherical'

Data Types: char

## Usage

### Syntax

```
dets = sensor(sonarsigs,simTime)
dets = sensor(sonarsigs,txconfigs,simTime)
dets = sensor(___,ins,simTime)
[dets,numDets,config] = sensor(___)
```

### Description

`dets = sensor(sonarsigs,simTime)` creates passive detections, `dets`, from sonar emissions, `sonarsigs`, at the current simulation time, `simTime`. The sensor generates detections at the rate defined by the `UpdateRate` property.

`dets = sensor(sonarsigs,txconfigs,simTime)` also specifies emitter configurations, `txconfigs`, at the current simulation time.

`dets = sensor(___,ins,simTime)` also specifies the inertial navigation system (INS) estimated sensor platform pose, `ins`. INS information is used by tracking and fusion algorithms to estimate the target positions in the NED frame.

To enable this syntax, set the `HasINS` property to `true`.

`[dets,numDets,config] = sensor(___)` also returns the number of valid detections reported, `numValidDets`, and the configuration of the sensor, `config`, at the current simulation time.

### Input Arguments

#### **sonarsigs — Sonar emissions**

array of sonar emission objects

Sonar emissions, specified as an array of `sonarEmission` objects.

#### **txconfigs — Emitter configurations**

array of structures

Emitter configurations, specified as an array of structures. Each structure has these fields:

Field	Description
EmitterIndex	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.

Data Types: struct

#### **ins — Platform pose from INS**

structure

Sensor platform pose obtained from the inertial navigation system (INS), specified as a structure.

Platform pose information from an inertial navigation system (INS) is a structure with these fields:

Field	Definition
<code>Position</code>	Position in the navigation frame, specified as a real-valued 1-by-3 vector. Units are in meters.
<code>Velocity</code>	Velocity in the navigation frame, specified as a real-valued 1-by-3 vector. Units are in meters per second.
<code>Orientation</code>	Orientation with respect to the navigation 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.

#### **Dependencies**

To enable this argument, set the `HasINS` property to `true`.

Data Types: struct

#### **simTime — Current simulation time**

nonnegative scalar

Current simulation time, specified as a positive scalar. The `trackingScenario` object calls the sonar sensor at regular time intervals. The sonar sensor generates new detections 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 sensor between update intervals contain no detections. Units are in seconds.

Example: 10.5

Data Types: double

### Output Arguments

#### **dets — sensor detections**

cell array of `objectDetection` objects

Sensor detections, returned as a cell array of `objectDetection` objects. Each object has these properties:

Property	Definition
<code>Time</code>	Measurement time
<code>Measurement</code>	Object measurements
<code>MeasurementNoise</code>	Measurement noise covariance matrix
<code>SensorIndex</code>	Unique ID of the sensor
<code>ObjectClassID</code>	Object classification
<code>ObjectAttributes</code>	Additional information passed to tracker
<code>MeasurementParameters</code>	Parameters used by initialization functions of nonlinear Kalman tracking filters

`Measurement` and `MeasurementNoise` are reported in the coordinate system specified by the `DetectionCoordinates` property.

#### **numDets — Number of detections**

nonnegative integer

Number of detections reported, returned as a nonnegative integer.

- When the `MaxNumDetectionsSource` property is set to 'Auto', `numDets` is set to the length of `dets`.
- When the `MaxNumDetectionsSource` property is set to 'Property', `dets` is a cell array with length determined by the `MaxNumDetections` property. No more than `MaxNumDetections` number of detections are returned. If the number of detections is fewer than `MaxNumDetections`, the first `numDets` elements of `dets` hold valid detections. The remaining elements of `dets` are set to the default value.

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 sonar beam during object execution.

Field	Description
<code>SensorIndex</code>	Unique sensor index, returned as a positive integer.

<b>IsValidTime</b>	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.
<b>IsScanDone</b>	<code>IsScanDone</code> is <code>true</code> when the sensor has completed a scan.
<b>FieldOfView</b>	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.
<b>MeasurementParameters</b>	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 sonarSensor

<code>coverageConfig</code>	Sensor and emitter coverage configuration
<code>perturbations</code>	Perturbation defined on object
<code>perturb</code>	Apply perturbations to object

### 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
<code>reset</code>	Reset internal states of System object

## Examples

### Detect Sonar Emission with Passive Sensor

Create a sonar emission and then detect the emission using a `sonarSensor` object.

First, create a sonar emission.

```
orient = quaternion([180 0 0],'eulerd','zyx','frame');
sonarSig = sonarEmission('PlatformID',1,'EmitterIndex',1, ...
    'OriginPosition',[30 0 0],'Orientation',orient, ...
    'SourceLevel',140,'TargetStrength',100);
```

Then create a passive sonar sensor.

```
sensor = sonarSensor(1,'No scanning');

Detect the sonar emission.

time = 0;
[dets, numDets, config] = sensor(sonarSig,time)

dets = 1x1 cell array
    {1x1 objectDetection}

numDets = 1

config = struct with fields:
    SensorIndex: 1
    IsValidTime: 1
    IsScanDone: 1
    FieldOfView: [1 5]
    MeasurementParameters: [1x1 struct]
```

## More About

### Object Detections

#### Measurements

The sensor measures the coordinates of the target. The `Measurement` and `MeasurementNoise` values are reported in the coordinate system specified by the `DetectionCoordinates` property of the sensor.

When the `DetectionCoordinates` property is '`Scenario`', '`Body`', or '`Sensor rectangular`', the `Measurement` and `MeasurementNoise` values are reported in rectangular coordinates. Velocities are only reported when the range rate property, `HasRangeRate`, is `true`.

When the `DetectionCoordinates` property is '`Sensor spherical`', the `Measurement` and `MeasurementNoise` values are reported in a spherical coordinate system derived from the sensor rectangular coordinate system. Elevation and range rate are only reported when `HasElevation` and `HasRangeRate` are `true`.

Measurements are ordered as [azimuth, elevation, range, range rate]. Reporting of elevation and range rate depends on the corresponding `HasElevation` and `HasRangeRate` property values. Angles are in degrees, range is in meters, and range rate is in meters per second.

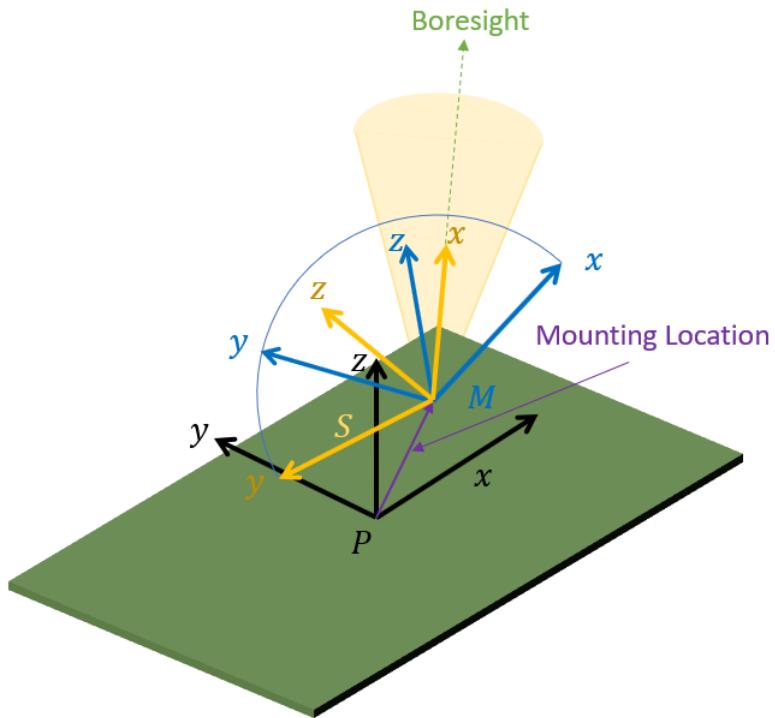
## Measurement Coordinates

DetectionCoordinates	Measurement and Measurement Noise Coordinates	
'Scenario'	Coordinate Dependence on HasRangeRate	
'Body'	HasRangeRate	Coordinates
'Sensor rectangular'	true	[x; y; z; vx; vy; vz]
	false	[x; y; z]
'Sensor spherical'	Coordinate Dependence on HasRangeRate and HasElevation	
HasRangeRate	HasElevation	Coordinates
true	true	[az; el; rng; rr]
true	false	[az; rng; rr]
false	true	[az; el; rng]
false	false	[az; rng]

## Measurement Parameters

The **MeasurementParameters** property consists of an array of structures that describe a sequence of coordinate transformations from a child frame to a parent frame or the inverse transformations (see “Frame Rotation”). 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 the 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 form the scenario frame to the platform frame then from 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.

Trivially, if the detections are reported in platform rectangular coordinates and `HasINS` is set to `false`, the transformation consists only of the identity.

The fields of `MeasurementParameters` are shown here. Not all fields have to be present in the structure. The set of fields and their default values can depend on the type of sensor.

Field	Description
<code>Frame</code>	Enumerated type indicating the frame used to report measurements. When detections are reported using a rectangular coordinate system, <code>Frame</code> is set to ' <code>rectangular</code> '. When detections are reported in spherical coordinates, <code>Frame</code> is set ' <code>spherical</code> ' for the first struct.
<code>OriginPosition</code>	Position offset of the origin of the child frame relative to the parent frame, represented as a 3-by-1 vector.
<code>OriginVelocity</code>	Velocity offset of the origin of the child frame relative to the parent frame, represented as a 3-by-1 vector.

<b>Orientation</b>	3-by-3 real-valued orthonormal frame rotation matrix. The direction of the rotation depends on the <b>IsParentToChild</b> field.
<b>IsParentToChild</b>	A logical scalar indicating if <b>Orientation</b> performs a frame rotation from the parent coordinate frame to the child coordinate frame. If <b>false</b> , <b>Orientation</b> performs a frame rotation from the child coordinate frame to the parent coordinate frame.
<b>HasElevation</b>	A logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if <b>HasElevation</b> is <b>false</b> , the measurements are reported assuming 0 degrees of elevation.
<b>HasAzimuth</b>	A logical scalar indicating if azimuth is included in the measurement.
<b>HasRange</b>	A logical scalar indicating if range is included in the measurement.
<b>HasVelocity</b>	A logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if <b>HasVelocity</b> is <b>false</b> , the measurements are reported as [x y z]. If <b>HasVelocity</b> is <b>true</b> , measurements are reported as [x y z vx vy vz].

### Object Attributes

Object attributes contain additional information about a detection.

<b>Attribute</b>	<b>Description</b>
<b>TargetIndex</b>	Identifier of the platform, <b>PlatformID</b> , that generated the detection. For false alarms, this value is negative.
<b>EmitterIndex</b>	Index of the emitter from which the detected signal was emitted.
<b>SNR</b>	Detection signal-to-noise ratio in dB.
<b>CenterFrequency</b>	<ul style="list-style-type: none"> <li>Measured center frequency of the detected sonar signal. Units are in Hz.</li> <li>This attribute is present only when the <b>DetectionMode</b> property is set to '<b>passive</b>'.</li> </ul>
<b>Bandwidth</b>	<ul style="list-style-type: none"> <li>Measured bandwidth of the detected sonar signal, Units are in Hz.</li> <li>This attribute is present only when the <b>DetectionMode</b> property is set to '<b>passive</b>'.</li> </ul>

WaveformType	<ul style="list-style-type: none"> <li>Identifier of the waveform type that was classified by a passive sensor for the detected signal.</li> <li>This attribute is present only when the <code>DetectionMode</code> property is set to 'passive'.</li> </ul>
--------------	--

## Convenience Syntaxes

The convenience syntaxes set several properties together to model a specific type of sonar.

### No Scanning

Sets `ScanMode` to 'No scanning'.

### Raster Scanning

This syntax sets these properties:

Property	Value
<code>ScanMode</code>	'Electronic'
<code>HasElevation</code>	<code>true</code>
<code>ElectronicScanLimits</code>	<code>[-45 45; -10 0]</code>

### Rotator Scanning

This syntax sets these properties:

Property	Value
<code>ScanMode</code>	'Electronic'
<code>FieldOfView</code>	<code>[1:10]</code>
<code>HasElevation</code>	<code>false</code> or <code>true</code>
<code>ElevationResolution</code>	$10/\sqrt{12}$

### Sector Scanning

This syntax sets these properties:

Property	Value
<code>ScanMode</code>	'Electronic'
<code>FieldOfView</code>	<code>[1;10]</code>
<code>HasElevation</code>	<code>false</code>
<code>ElectronicScanLimits</code>	<code>[-45 45; -10 0]</code>
<code>ElevationResolution</code>	$10/\sqrt{12}$

## 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

### Objects

[objectDetection](#) | [sonarEmission](#)

### Functions

[targetPoses](#)

### System Objects

[trackerGNN](#) | [trackerTOMHT](#)

## Introduced in R2018b

# 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 `trackingScenario`.

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 “Raster Scanning” on page 3-272 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 “Rotator Scanning” on page 3-272 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 “Sector Scanning” on page 3-272 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 `trackingScenario`. 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 field of view **azfov** must lie in the interval (0,360]. The elevation field 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 <b>MechanicalScanLimits</b> 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 <b>ElectronicScanLimits</b> 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 <b>mountingAngles</b> 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° 90°]. 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
'Mechanical'	MechanicalAngle
'Electronic'	ElectronicAngle
'Mechanical and Electronic'	MechanicalAngle + ElectronicAngle
'No scanning'	0

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

**WaveformTypes — Types of detected waveforms**0 (default) | nonnegative integer-valued  $L$ -element vector

Types of detected waveforms, specified as a nonnegative integer-valued  $L$ -element vector.

Example: [1 4 5]

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
<code>PlatformID</code>	Unique identifier for the platform, specified as a scalar positive integer. This is a required field which has 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 vector. This is a required field. There is no default value. Units are in meters.
<code>Velocity</code>	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].
<code>Speed</code>	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.
<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 is [0 0 0].
<code>Orientation</code>	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 array {rcsSignature,irSignature , tsSignature}

**simTime — Current simulation time**

nonnegative scalar

Current simulation time, specified as a positive scalar. The `trackingScenario` 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
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.

<b>MeasurementParameters</b>	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.
------------------------------	---

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

<code>coverageConfig</code>	Sensor and emitter coverage configuration
<code>perturbations</code>	Perturbation defined on object
<code>perturb</code>	Apply perturbations to object

### 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
<code>reset</code>	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 `trackingScenario` object to manage the motion of the platforms.

```
scene = trackingScenario('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]
            Mesh: [1x1 extendedObjectMesh]
        Trajectory: [1x1 kinematicTrajectory]
```

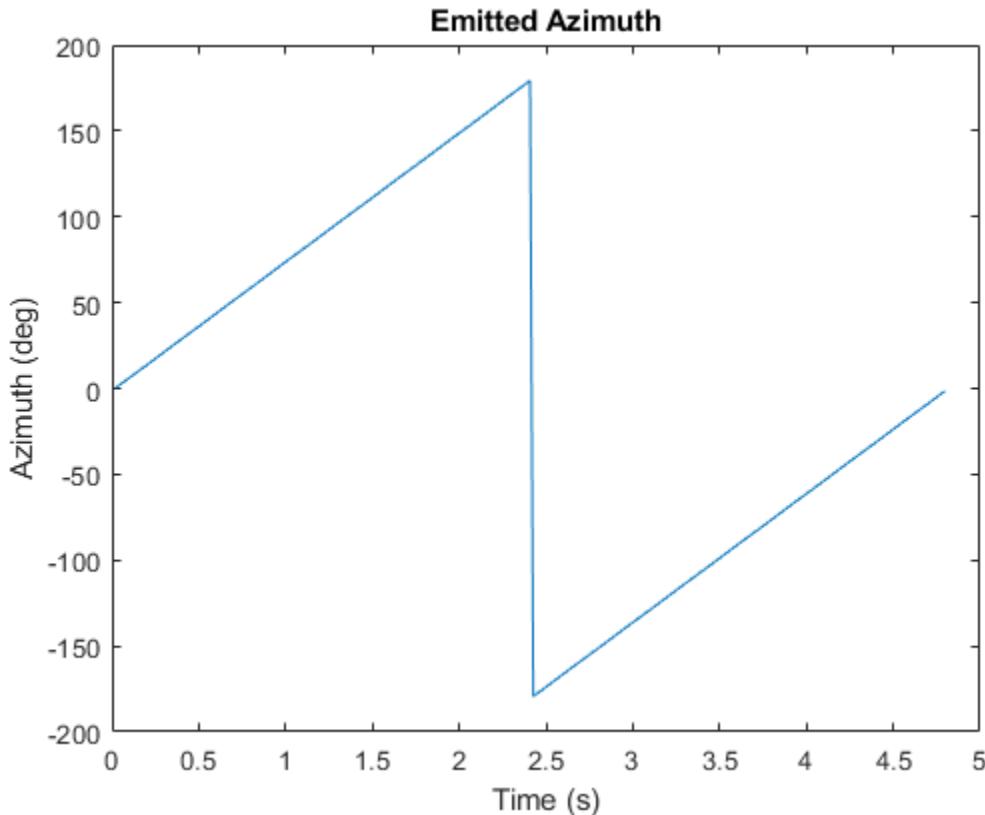
```
PoseEstimator: [1x1 insSensor]
    Emitters: {[1x1 radarEmitter]}
    Sensors: {}
    Signatures: {[1x1 rcsSignature] [1x1 irSignature] [1x1 tsSignature]}
```

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™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

## See Also

### Classes

`platform` | `radarEmission`

### Functions

`emissionsInBody` | `targetPoses`

### System Objects

`monostaticRadarSensor` | `radarSensor`

**Introduced in R2018b**

# sonarEmitter

Acoustic signals and interferences generator

## Description

The `sonarEmitter` System object creates an emitter to simulate sonar emissions. You can use the `sonarEmitter` object in a scenario that detects and tracks moving and stationary platforms. Construct a scenario using `trackingScenario`.

A sonar 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 sonar emitter scans the total region in azimuth and elevation defined by the sonar 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 sonar detections:

- 1 Create the `sonarEmitter` 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 = sonarEmitter(EmitterIndex)  
  
emitter = sonarEmitter(EmitterIndex,'No scanning')  
emitter = sonarEmitter(EmitterIndex,'Raster')  
emitter = sonarEmitter(EmitterIndex,'Rotator')  
emitter = sonarEmitter(EmitterIndex,'Sector')  
  
emitter = sonarEmitter(__,Name,Value)
```

### Description

`emitter = sonarEmitter(EmitterIndex)` creates a sonar emitter object with default property values.

`emitter = sonarEmitter(EmitterIndex,'No scanning')` is a convenience syntax that creates a `sonarEmitter` that stares along the sonar transducer boresight direction. No mechanical or electronic scanning is performed. This syntax sets the `ScanMode` property to `'No scanning'`.

`emitter = sonarEmitter(EmitterIndex,'Raster')` is a convenience syntax that creates a `sonarEmitter` object that mechanically scans a raster pattern. The raster span is  $90^\circ$  in azimuth

from  $-45^\circ$  to  $+45^\circ$  and in elevation from the horizon to  $10^\circ$  above the horizon. See “Raster Scanning” on page 3-272 for the properties set by this syntax.

`emitter = sonarEmitter(EmitterIndex, 'Rotator')` is a convenience syntax that creates a `sonarEmitter` object that mechanically scans  $360^\circ$  in azimuth by mechanically rotating the sonar at a constant rate. When you set `HasElevation` to `true`, the sonar mechanically points towards the center of the elevation field of view. See “Rotator Scanning” on page 3-272 for the properties set by this syntax.

`emitter = sonarEmitter(EmitterIndex, 'Sector')` is a convenience syntax to create a `sonarEmitter` object that mechanically scans a  $90^\circ$  azimuth sector from  $-45^\circ$  to  $+45^\circ$ . Setting `HasElevation` to `true`, points the sonar 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 sonar 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 “Sector Scanning” on page 3-272 for the properties set by this syntax.

`emitter = sonarEmitter(___, Name, Value)` sets properties using one or more name-value pairs after all other input arguments. Enclose each property name in quotes. For example, `sonarEmitter('CenterFrequency', 2e6)` creates a sonar 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 `sonarEmitter` 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 `trackingScenario`. 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 field of view **azfov** must lie in the interval (0,360]. The elevation field of view **elfov** must lie in the interval (0,180].

Example: [14;7]

Data Types: double

**ScanMode — Scanning mode of sonar**

'Mechanical' (default) | 'Electronic' | 'Mechanical and electronic' | 'No scanning'

Scanning mode of sonar, specified as 'Mechanical', 'Electronic', 'Mechanical and electronic', or 'No scanning'.

**Scan Modes**

ScanMode	Purpose
'Electronic'	The sonar scans electronically across the azimuth and elevation limits specified by the <b>ElectronicScanLimits</b> property. The scan direction increments by the sonar field of view angle between dwells.
'No scanning'	The sonar beam points along the antenna boresight defined by the <b>mountingAngles</b> property.

Example: 'No scanning'

Data Types: char

**ElectronicScanLimits — Angular limits of electronic scan directions of sonar**  
 $[-45 \ 45; -45 \ 45]$  (default) | real-valued 1-by-2 row vector | real-valued 2-by-2 matrix

Angular limits of electronic scan directions of sonar, 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 sonar 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 sonar, 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
----------	-----------

'Mechanical'	MechanicalAngle
'Electronic'	ElectronicAngle
'Mechanical and Electronic'	MechanicalAngle + ElectronicAngle
'No scanning'	0

Data Types: double

**HasElevation — Enable sonar elevation scan and measurements**

false (default) | true

Enable the sonar to measure target elevation angles and to scan in elevation, specified as `false` or `true`. Set this property to `true` to model a sonar emitter that can estimate target elevation and scan in elevation.

Data Types: logical

**SourceLevel — Sonar source level**

140 (default) | scalar

Sonar source level, specified as a scalar. Source level is relative to the intensity of a sound wave having an rms pressure of 1  $\mu\text{Pa}$ . Units are in dB/1  $\mu\text{Pa}$ .

Data Types: double

**CenterFrequency — Center frequency of sonar band**

positive scalar

Center frequency of sonar band, specified as a positive scalar. Units are in hertz.

Example: `100e6`

Data Types: double

**Bandwidth — Sonar waveform bandwidth**

positive scalar

Sonar waveform bandwidth, specified as a positive scalar. Units are in hertz.

Example: `100e3`

Data Types: double

**WaveformTypes — Types of detected waveforms**

0 (default) | nonnegative integer-valued  $L$ -element vector

Types of detected waveforms, specified as a nonnegative integer-valued  $L$ -element vector.

Example: `[1 4 5]`

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

```
sonarsigs = emitter(platform,simTime)
[sonarsigs,config] = emitter(platform,simTime)
```

### Description

`sonarsigs = emitter(platform,simTime)` creates sonar signals, `sonarsigs`, from emitter on the platform at the current simulation time, `simTime`. The emitter object can simultaneously generate signals from multiple emitters on the platform.

`[sonarsigs,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 [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.

Field	Description
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 quaternion(1,0,0,0).
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 array {rcsSignature,irSignature , tsSignature}

**simTime — Current simulation time**

nonnegative scalar

Current simulation time, specified as a positive scalar. The `trackingScenario` object calls the sonar emitter at regular time intervals. The sonar 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****sonarsigs — Sonar emissions**

array of sonar emission objects

Sonar emissions, returned as an array of `sonarEmission` objects.**config — Current emitter configuration**

structure array

Current emitter configurations, returned as an array of structures.

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.

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 sonarEmitter

<code>coverageConfig</code>	Sensor and emitter coverage configuration
<code>perturbations</code>	Perturbation defined on object
<code>perturb</code>	Apply perturbations to object

### 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
<code>reset</code>	Reset internal states of System object

## Examples

### Reflect Sonar Emission from Platform within Tracking Scenario

Reflect a sonar emission from a platform defined within a trackingScenario.

Create a tracking scenario object.

```
scenario = trackingScenario;
```

Create an sonarEmitter.

```
emitter = sonarEmitter(1);
```

Mount the emitter on a platform within the scenario.

```
plat = platform(scenario,'Emitters',emitter);
```

Add another platform to reflect the emitted signal.

```
tgt = platform(scenario);
tgt.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 sonarEmission}
```

Reflect the signal from the platforms in the scenario.

```
sigs = underwaterChannel(txSigs, scenario.Platforms)
```

```
sigs = 1x1 cell array
    {1x1 sonarEmission}
```

## More About

### Convenience Syntaxes

The convenience syntaxes set several properties together to model a specific type of sonar emitter.

#### No Scanning

Sets `ScanMode` to 'No scanning'.

#### Raster Scanning

This syntax sets these properties:

Property	Value
<code>ScanMode</code>	'Electronic'
<code>HasElevation</code>	<code>true</code>
<code>ElectronicScanLimits</code>	<code>[-45 45; -10 0]</code>

#### Rotator Scanning

This syntax sets these properties:

Property	Value
<code>ScanMode</code>	'Electronic'
<code>FieldOfView</code>	<code>[1:10]</code>
<code>HasElevation</code>	<code>false</code> or <code>true</code>
<code>ElevationResolution</code>	$10/\sqrt{12}$

## Sector Scanning

This syntax sets these properties:

Property	Value
ScanMode	'Electronic'
FieldOfView	[1;10]
HasElevation	false
ElectronicScanLimits	[ -45 45; -10 0 ]
ElevationResolution	10/sqrt(12)

## 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

### Classes

[platform](#) | [sonarEmission](#)

### Functions

[emissionsInBody](#) | [targetPoses](#)

### System Objects

[sonarSensor](#)

## Introduced in R2018b

# 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 3-183.

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 3-183.

## 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

<code>lookupPose</code>	Obtain pose of geodetic trajectory for a certain time
<code>perturbations</code>	Perturbation defined on object
<code>perturb</code>	Apply perturbations to object

### Common to All System Objects

<code>clone</code>	Create duplicate <code>System</code> object
<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
<code>isDone</code>	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 = 37x3
```

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
:
```

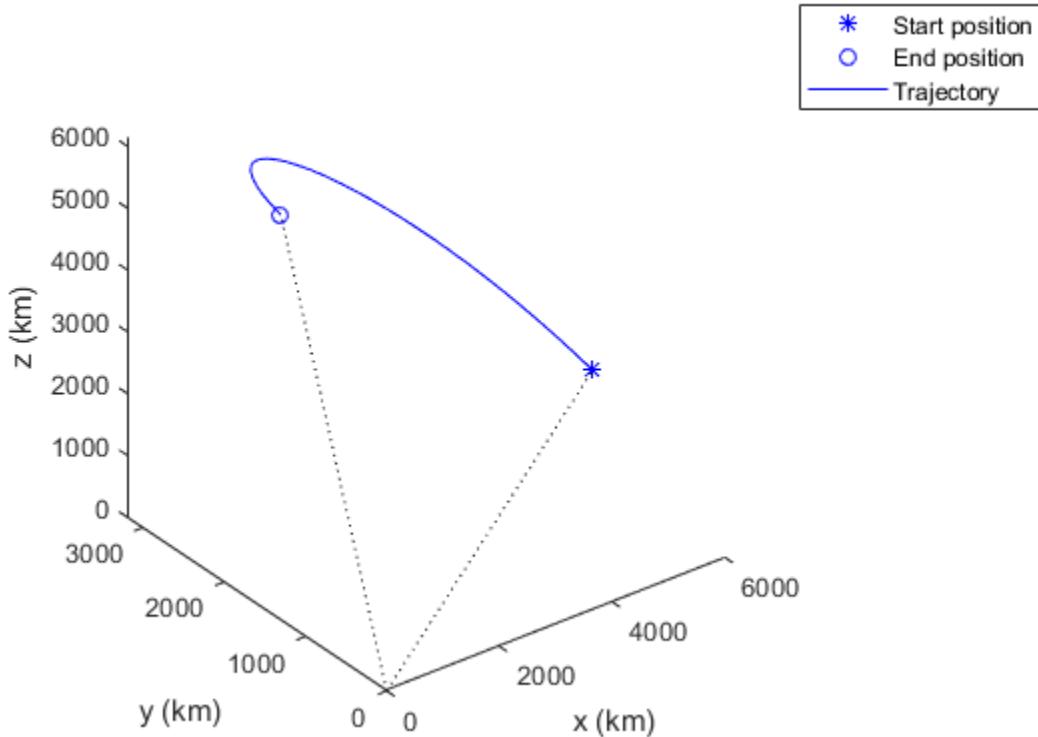
Look up the Cartesian waypoints of the trajectory in the ECEF frame using the `lookupPose` function.

```
sampleTimes = 0:1000:3600*10;
n = length(sampleTimes);
positionsCart = lookupPose(trajectory,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

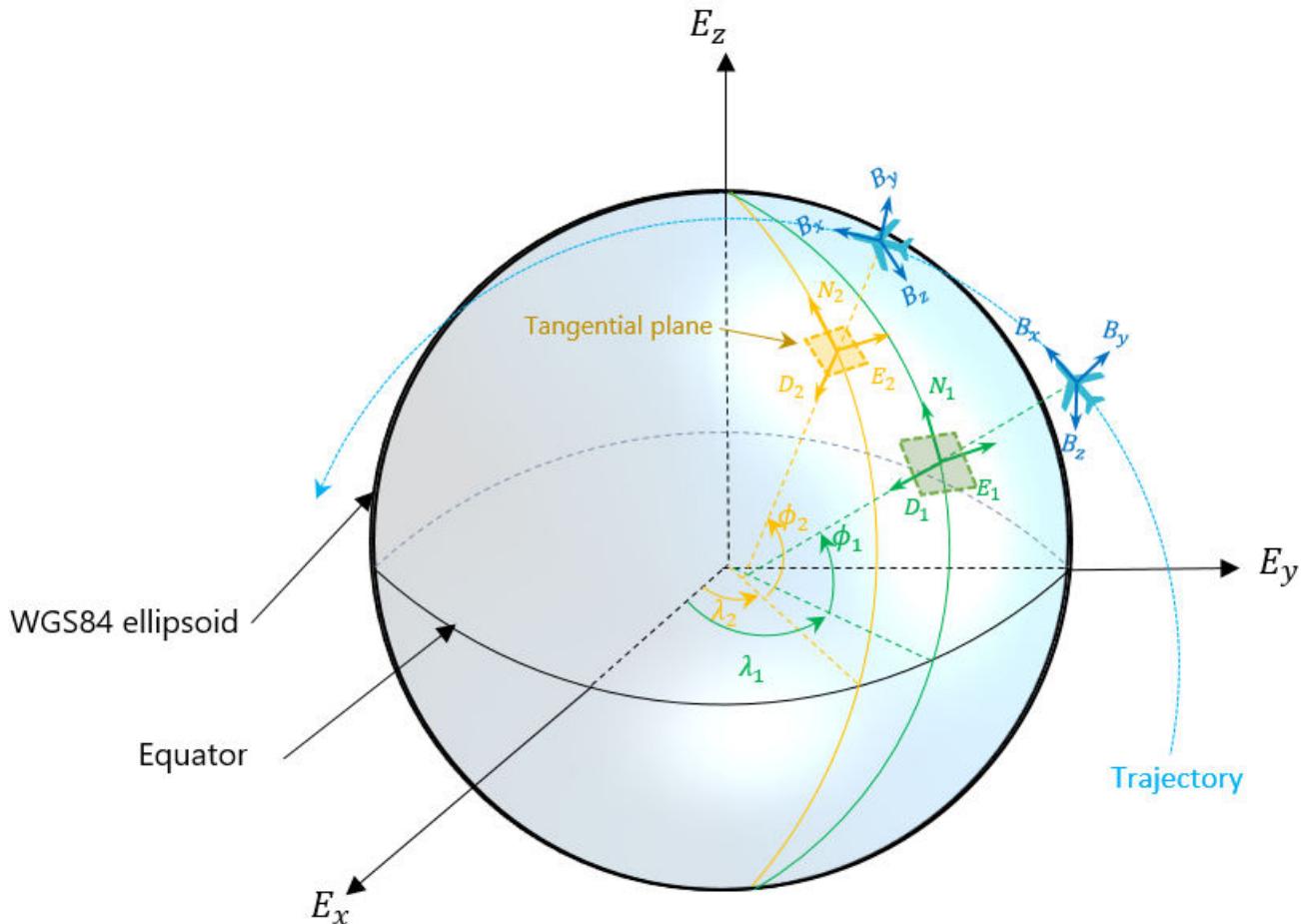
The geoTrajectory System object involves three coordinate frames:

- 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 figures 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™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

### See Also

`kinematicTrajectory` | `waypointTrajectory`

Introduced in R2020b

# 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 = 37x3

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 using the `lookupPose` function.

```

sampleTimes = 0:1000:3600*10;
n = length(sampleTimes);
positionsCart = lookupPose(traj, sampleTimes, 'ECEF');

```

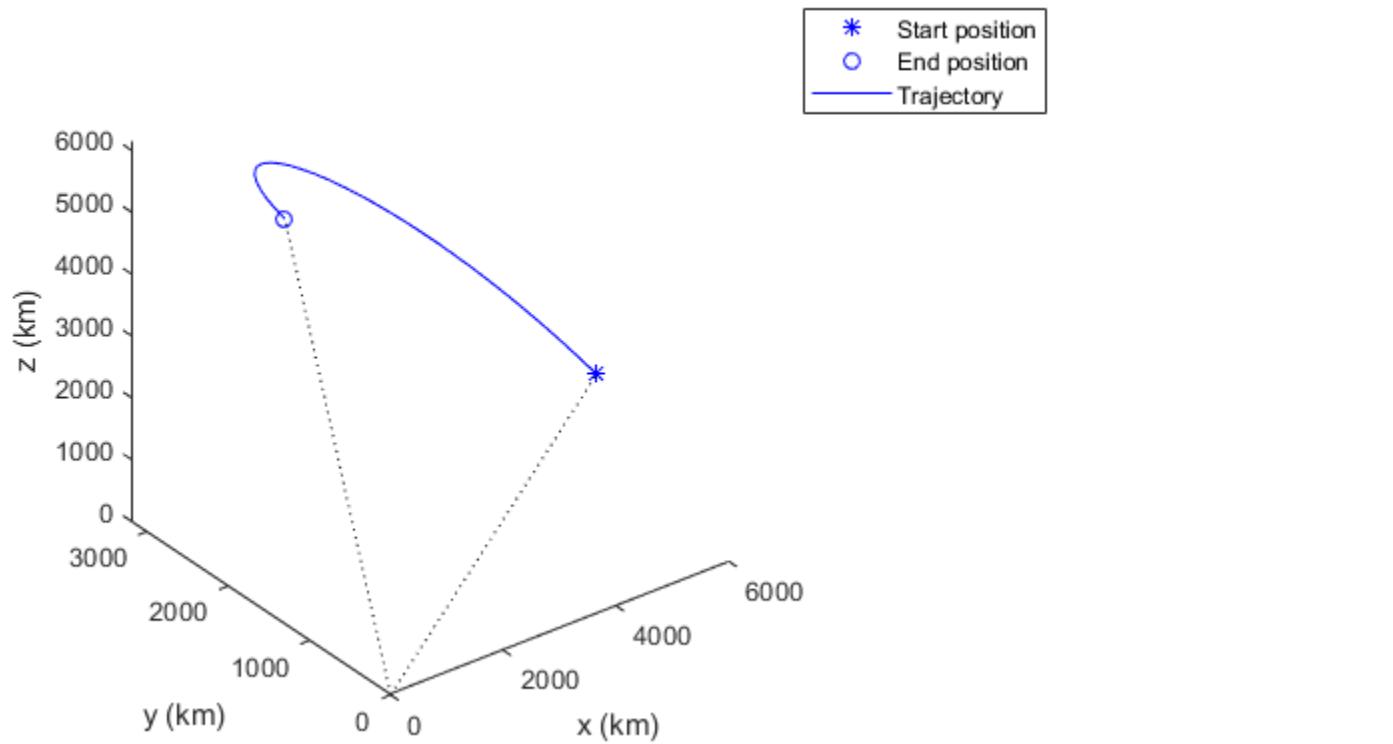
Visualize the results in the ECEF frame.

```

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 R2020b

# 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 = 1x3
```

```
5      5      0
```

```
rotatedAngularVelocity = rotatepoint(orientation,angularVelocity)
```

```
rotatedAngularVelocity = 1x3
```

```
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(traj);
traj.Position = [0,0,0];
```

```
position = traj(bodyAcceleration,bodyAngularVelocity)
```

```
position = 1x3
```

```
0      0      0
```

```
clonePosition = trajClone(bodyAcceleration,bodyAngularVelocity)
```

```
clonePosition = 1x3
10^-3 x
```

```
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
```

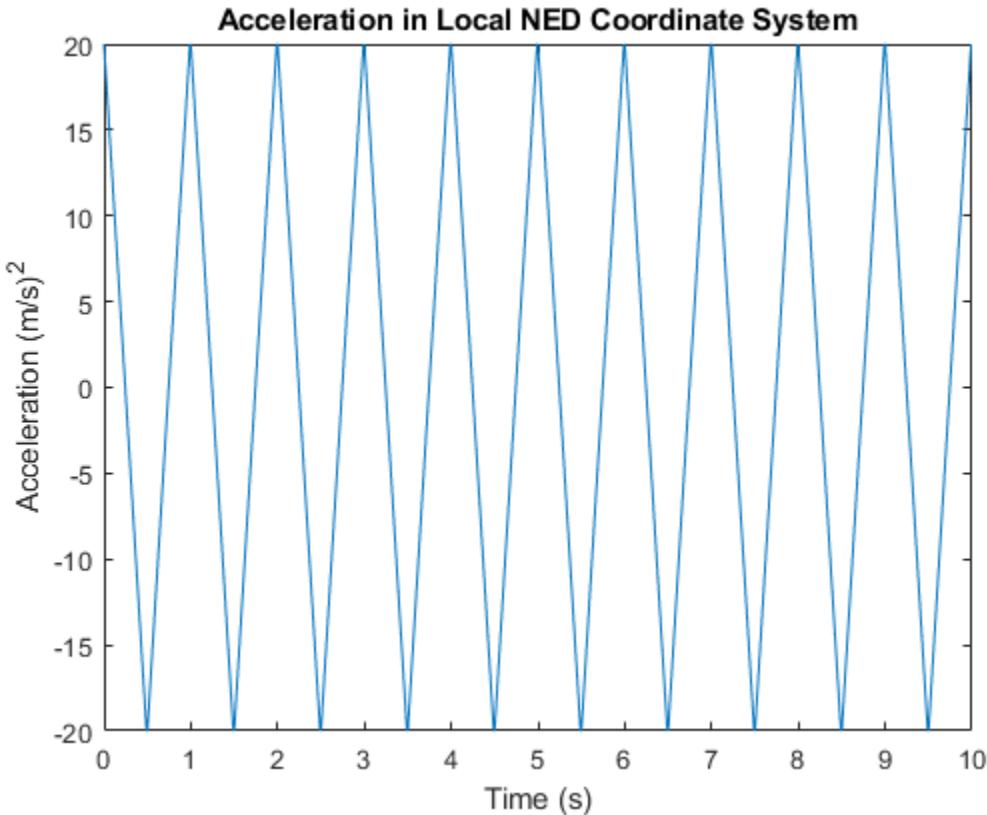
```
traj =  
kinematicTrajectory with properties:  
  
    SampleRate: 100  
    Position: [0 0 0]  
    Orientation: [1x1 quaternion]  
    Velocity: [0 0 0]  
    AccelerationSource: 'Input'  
    AngularVelocitySource: 'Input'
```

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);

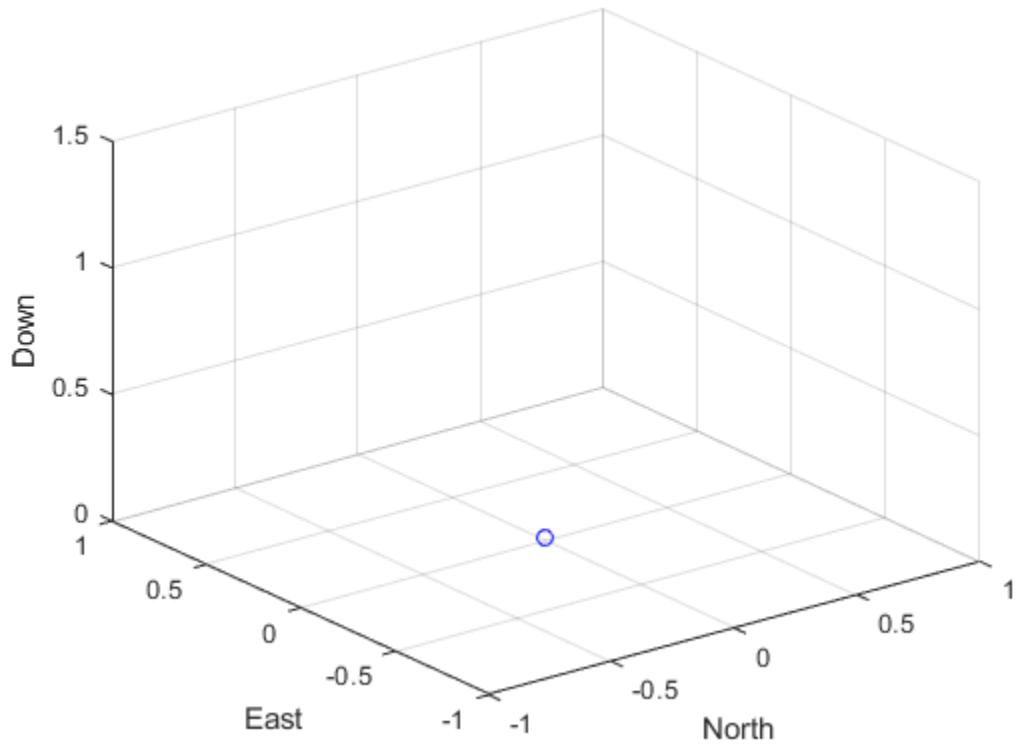
    set(plotHandle,'XData',pos(ii,1),'YData',pos(ii,2),'ZData',pos(ii,3))

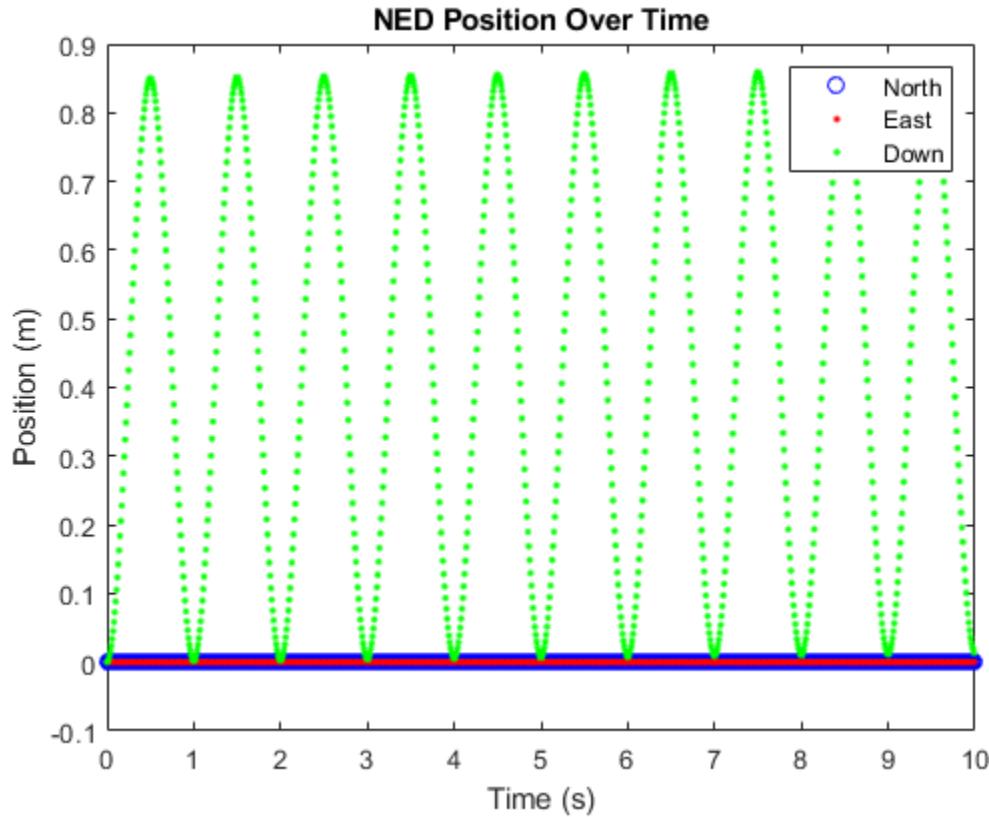
    pause(1/fs)
end

```

```
end

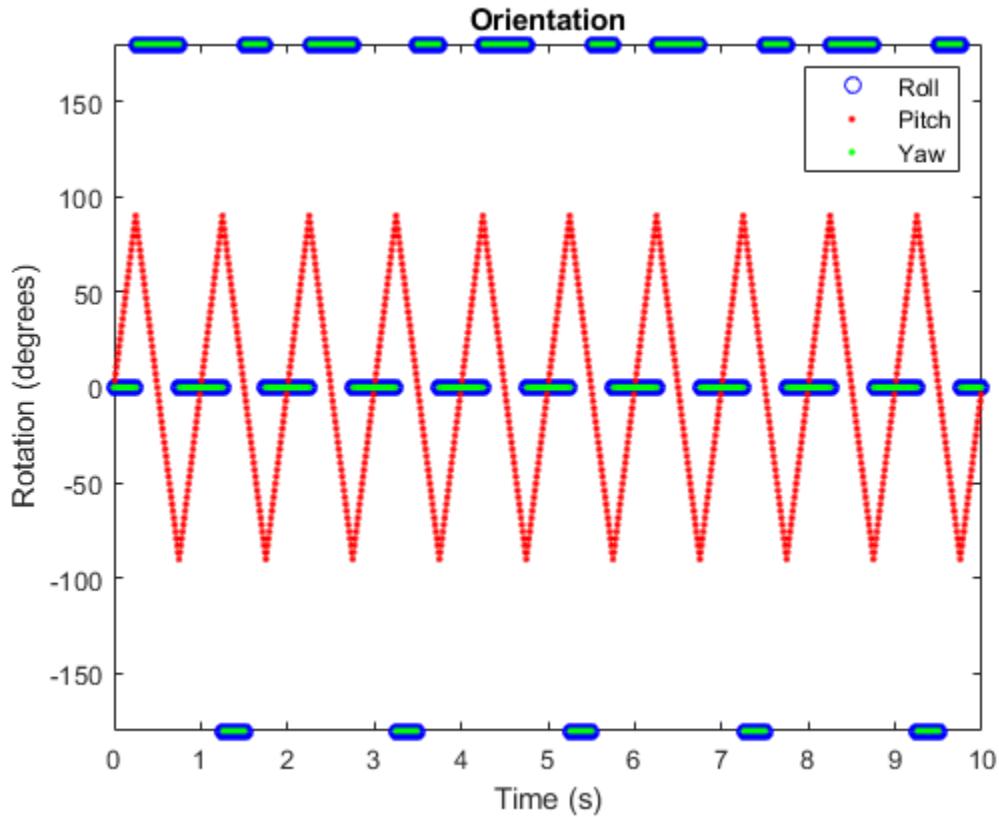
figure(3)
plot(timeVector,pos(:,1),'bo',...
      timeVector,pos(:,2),'r.',...
      timeVector,pos(:,3),'g.')
xlabel('Time (s)')
ylabel('Position (m)')
title('NED Position Over Time')
legend('North','East','Down')
```





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('Roll','Pitch','Yaw')
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], 'eulerrd', '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);
```

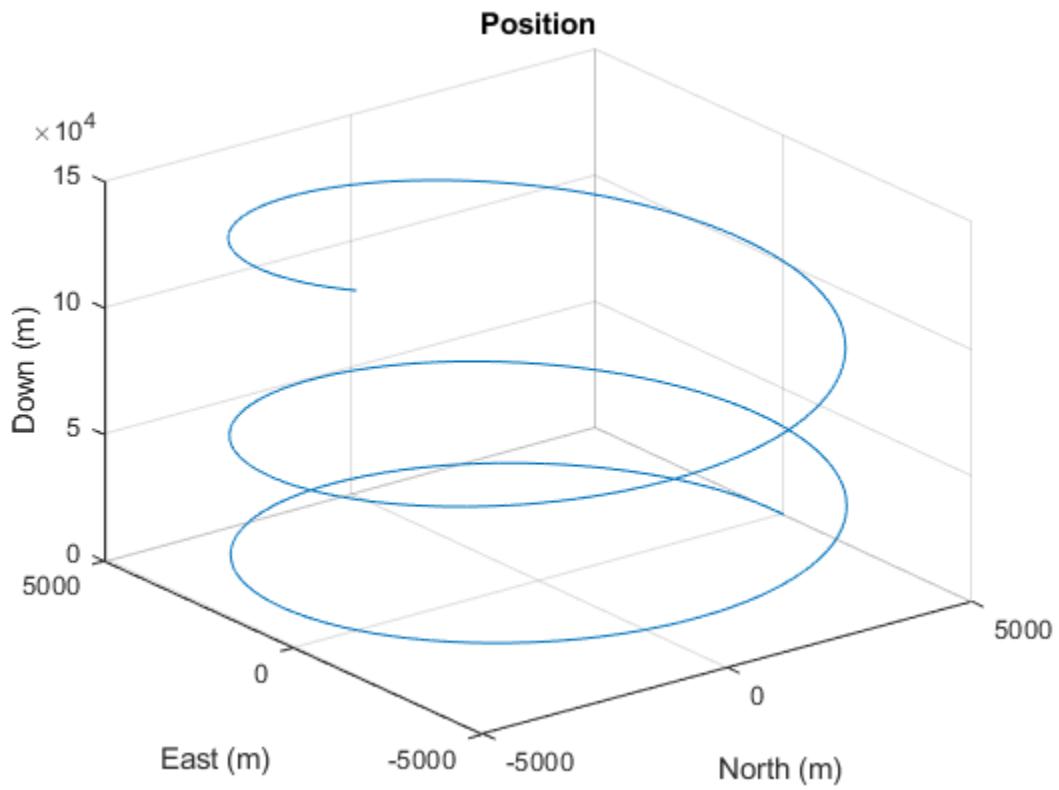
Call `trajectory` with the specified acceleration and angular velocity in the body coordinate system. Plot the position, orientation, and speed over time.

```
[position, orientation, velocity] = trajectory(accBody,angVelBody);

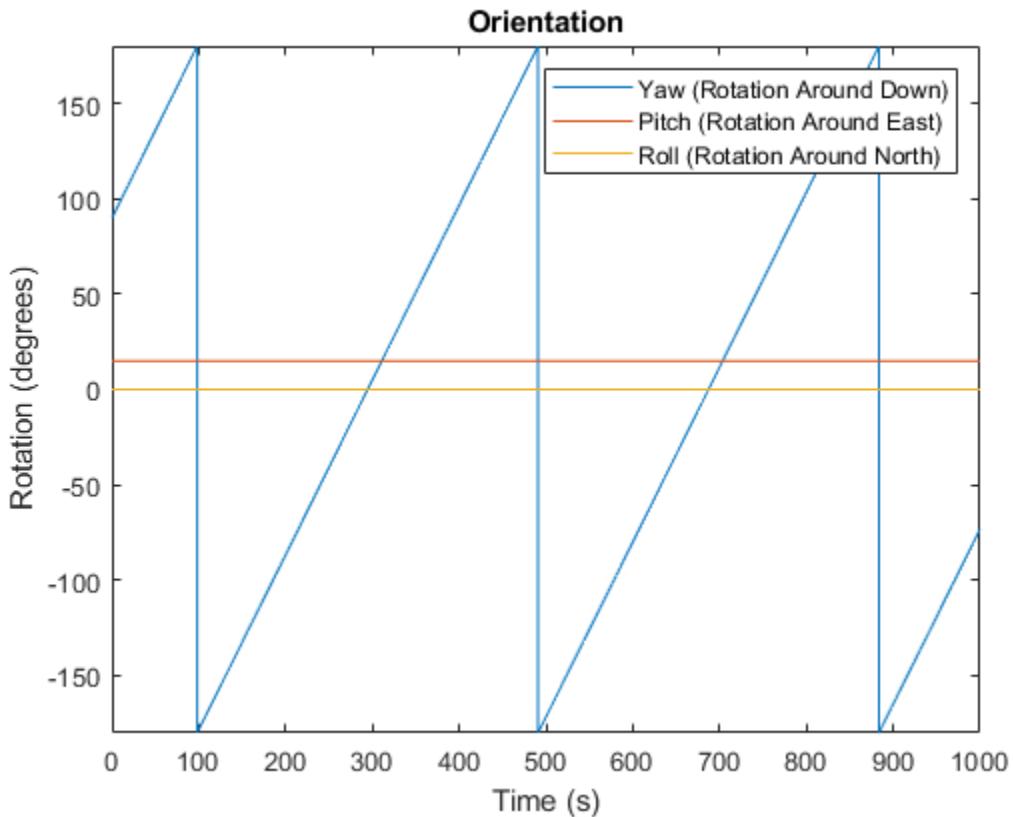
eulerAngles = eulerd(orientation, 'ZYX', 'frame');
speed = sqrt(sum(velocity.^2,2));

timeVector = (0:(N-1))/fs;

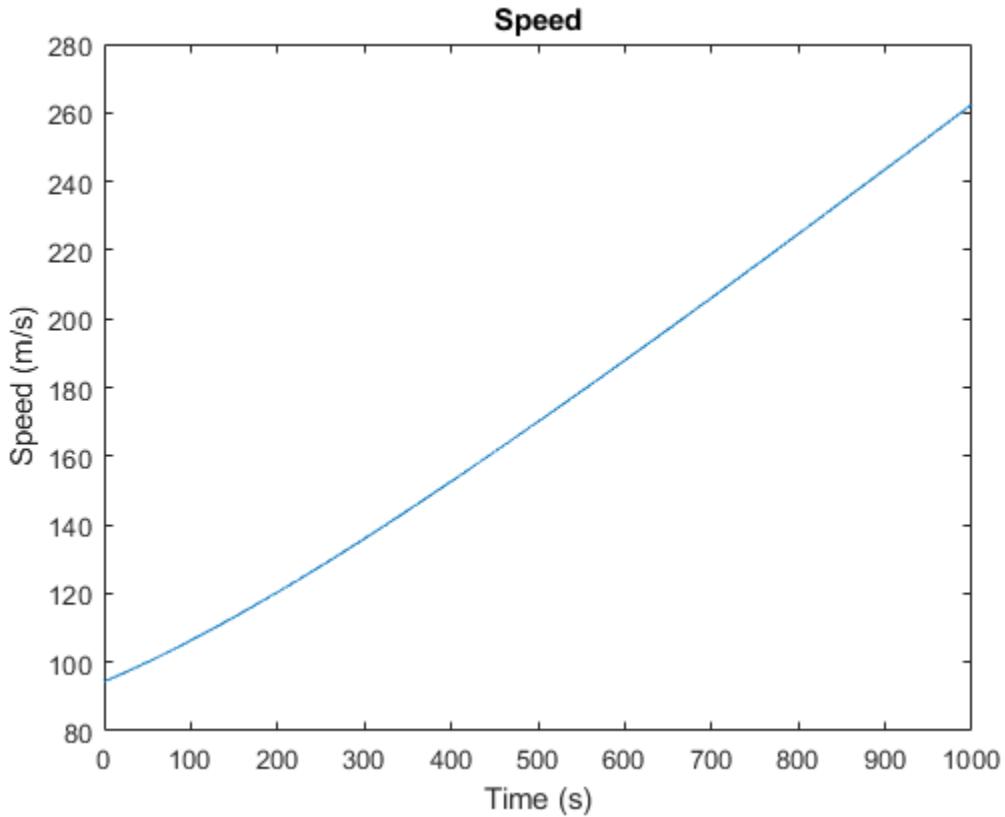
figure(1)
plot3(position(:,1),position(:,2),position(:,3))
xlabel('North (m)')
ylabel('East (m)')
zlabel('Down (m)')
title('Position')
grid on
```



```
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], 'eulerrd', 'ZYX', 'frame');

accBody = [0 speed^2/r 0.01];
angVelBody = [0 0 speed/r];
```

Create a kinematic trajectory object.

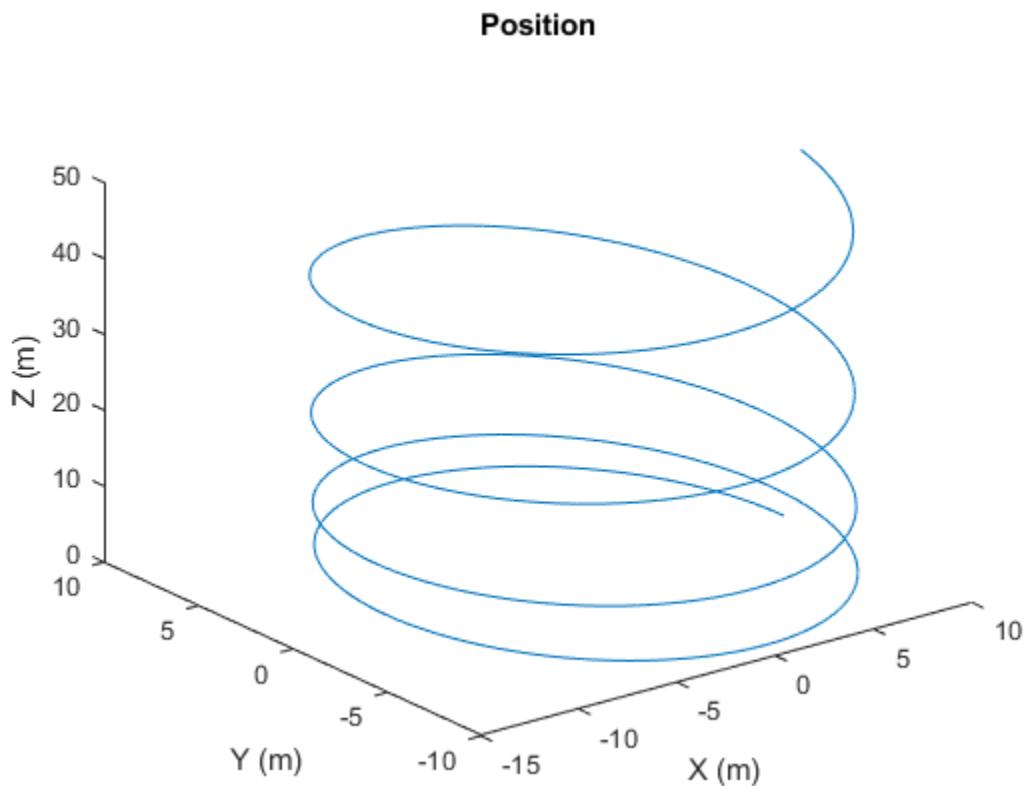
```
traj = kinematicTrajectory('SampleRate',Fs, ...
    'Position',initPos, ...
    'Velocity',initVel, ...
    'Orientation',initOrient, ...
    'AccelerationSource','Property', ...
    'Acceleration',accBody, ...
```

```
'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™.

Usage notes and limitations:

"System Objects in MATLAB Code Generation" (MATLAB Coder)

**See Also**

[platform](#) | [trackingScenario](#) | [waypointTrajectory](#)

**Introduced in R2018b**

# 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.

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 way point 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 — Climbrate at each waypoint (m/s)**

$N$ -element real vector

Climbrate 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, climbrate 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. 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: `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 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 counteract 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

waypointInfo	Get waypoint information table
lookupPose	Obtain pose information for 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 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

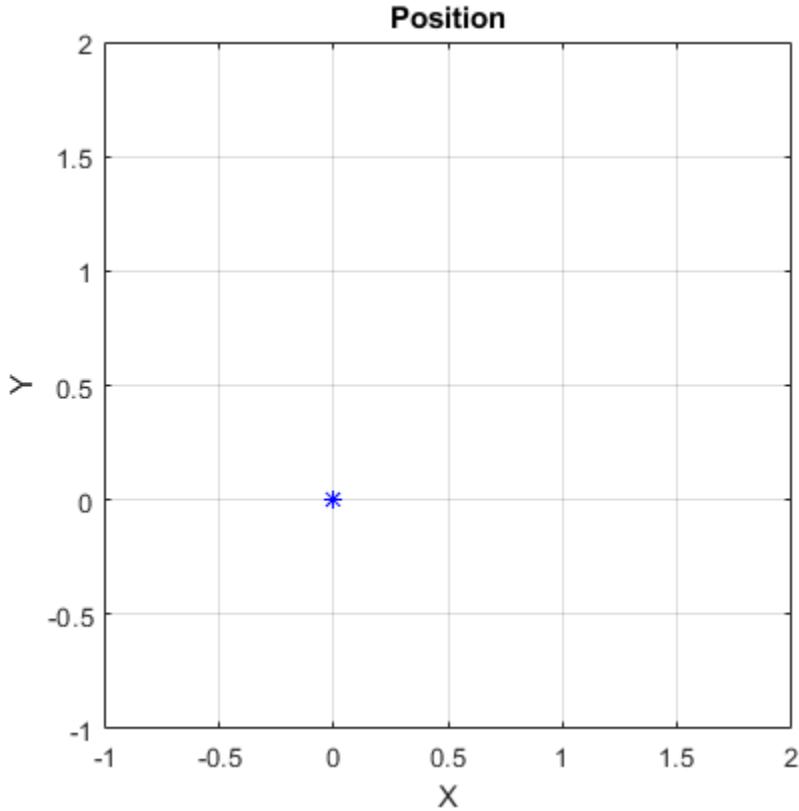
toa = 0:4; % time of arrival

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.

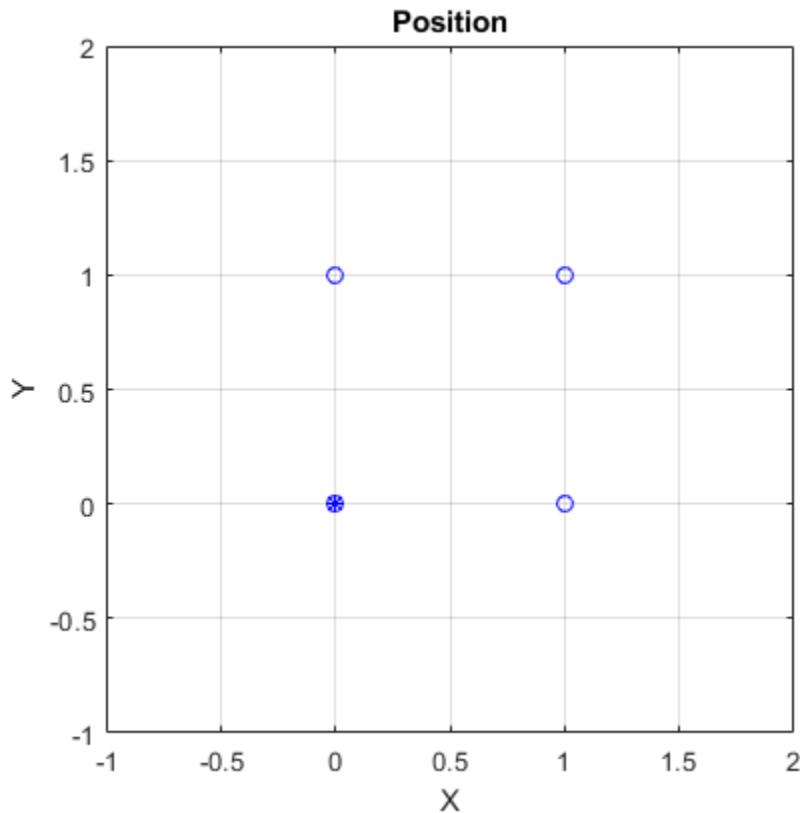
```
figure(1)
plot(waypoints(1,1),waypoints(1,2), 'b*')
title('Position')
axis([-1,2,-1,2])
axis square
xlabel('X')
ylabel('Y')
grid on
hold on
```



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.

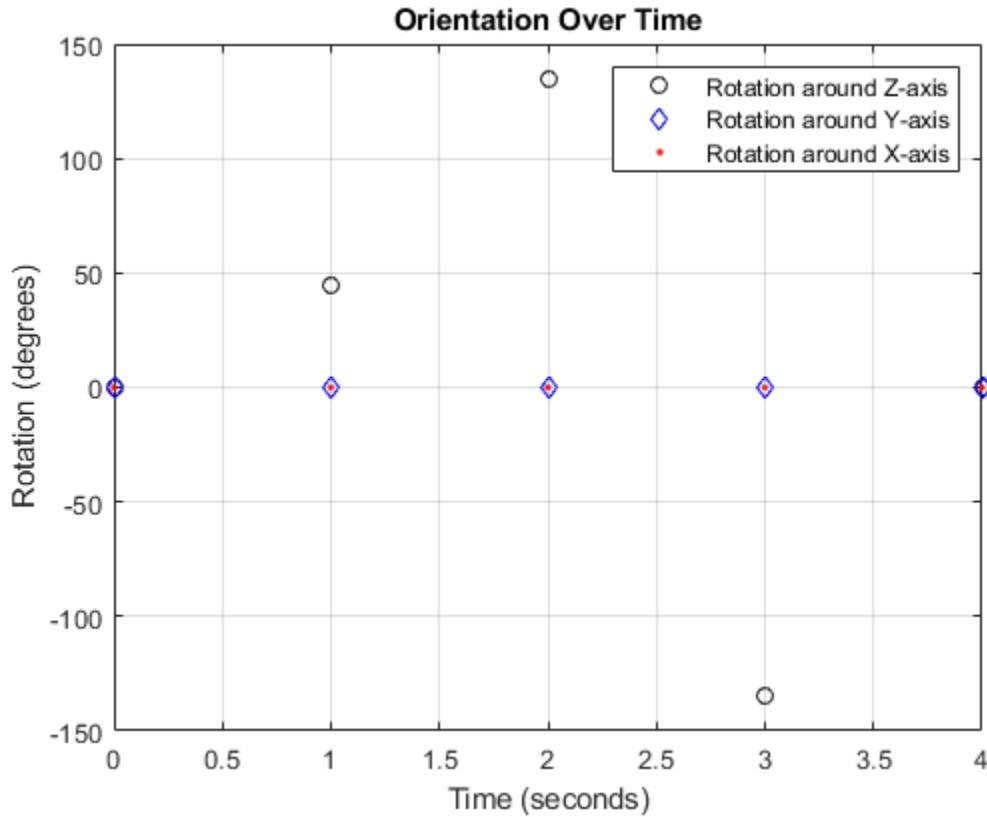
```
orientationLog = zeros(toa(end)*trajectory.SampleRate,1,'quaternion');
count = 1;
while ~isDone(trajectory)
    [currentPosition,orientationLog(count)] = trajectory();
    plot(currentPosition(1),currentPosition(2),'bo')

    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.

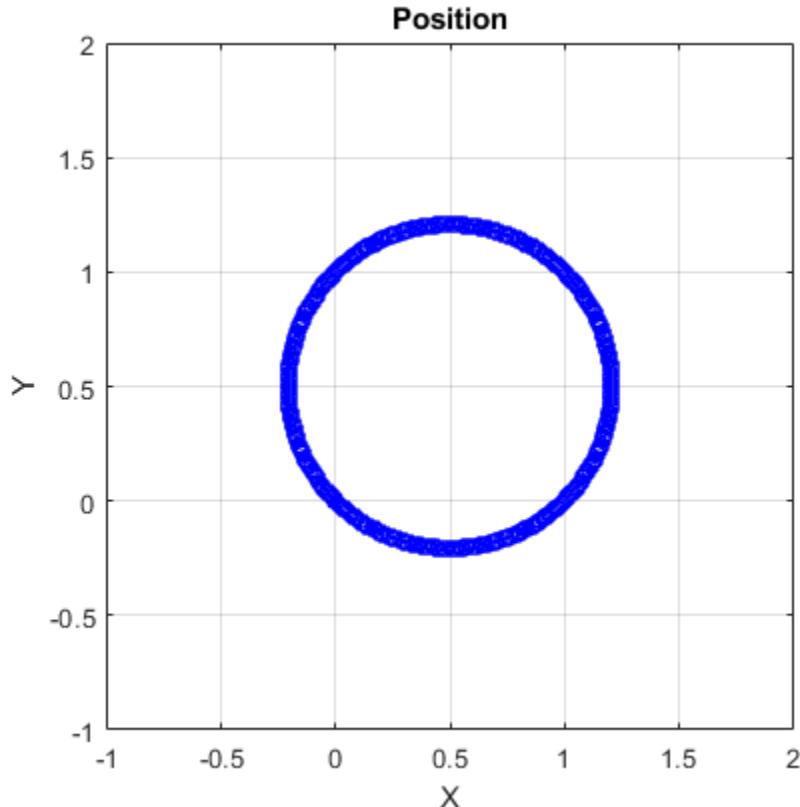
```
figure(1)
plot(waypoints(1,1),waypoints(1,2), 'b*')
title('Position')
axis([-1,2,-1,2])
axis square
xlabel('X')
ylabel('Y')
grid on
hold on

orientationLog = zeros(toa(end)*trajectory.SampleRate,1, 'quaternion');
count = 1;
while ~isDone(trajectory)
    [currentPosition,orientationLog(count)] = trajectory();
    plot(currentPosition(1),currentPosition(2), 'bo')
    pause(0.01);
end
```

```

    pause(traj.SamplesPerFrame/traj.SampleRate)
    count = count + 1;
end
hold off

```



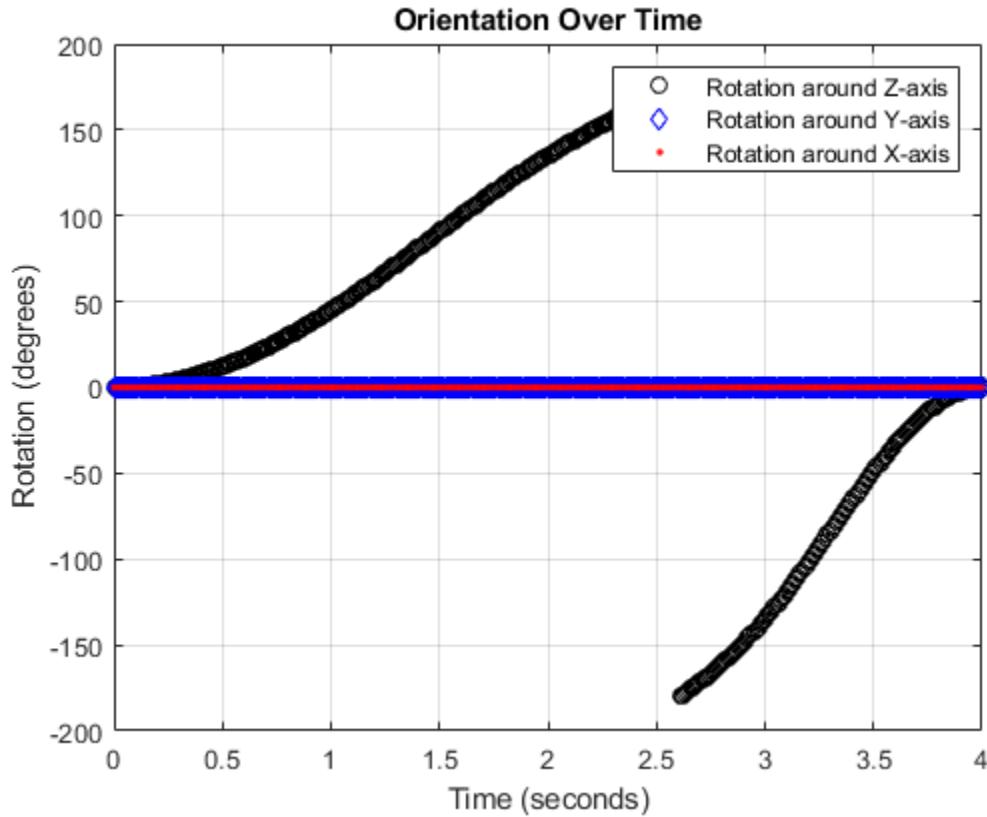
The trajectory output now appears circular. This is because the `waypointTrajectory` System object™ minimizes the acceleration and angular velocity when interpolating, which results in smoother, more realistic motions in most scenarios.

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:traj.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, 0,0.9,0, 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.

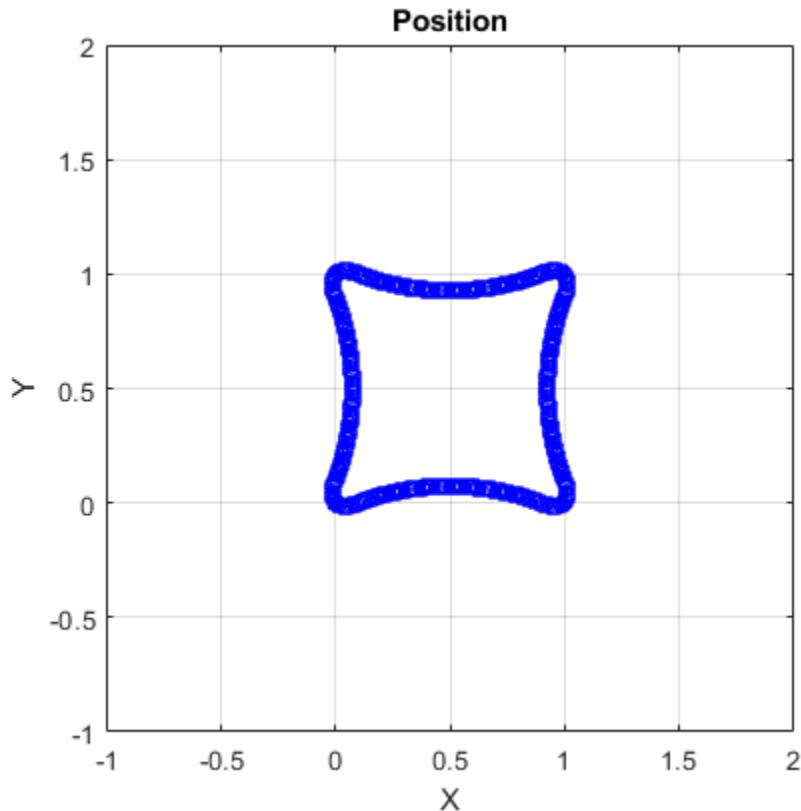
```

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(trajecotry.SamplesPerFrame/trajectory.SampleRate)
    count = count+1;
end
hold off

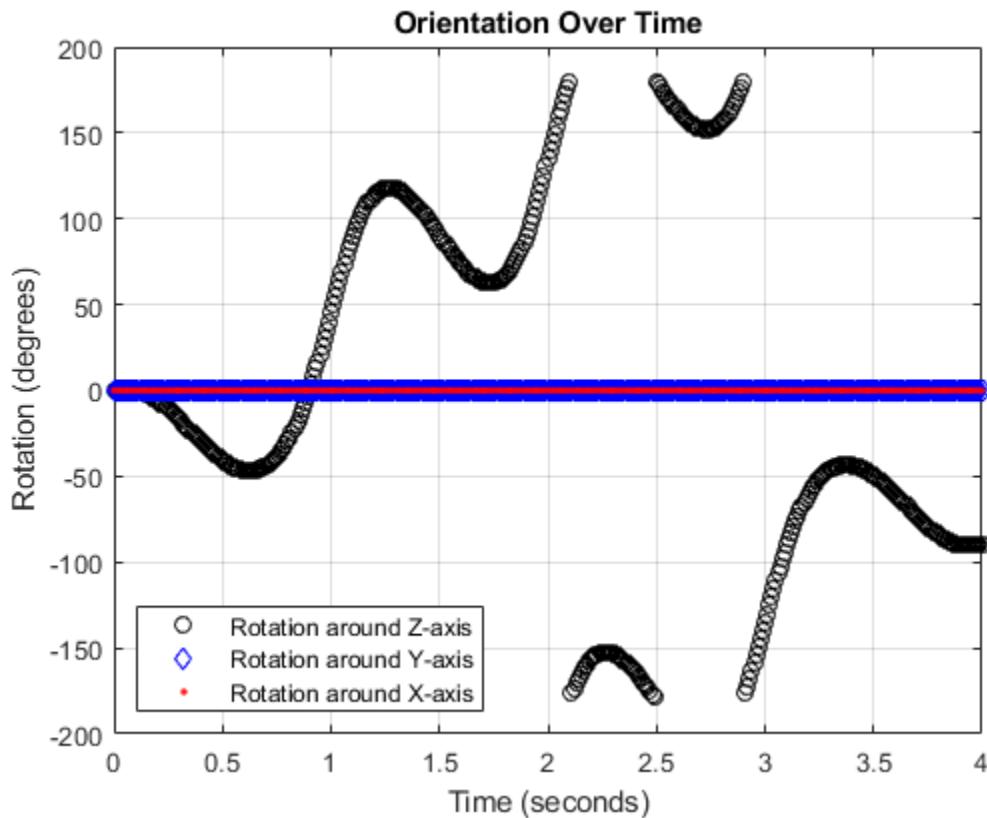
```



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 =
```

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}

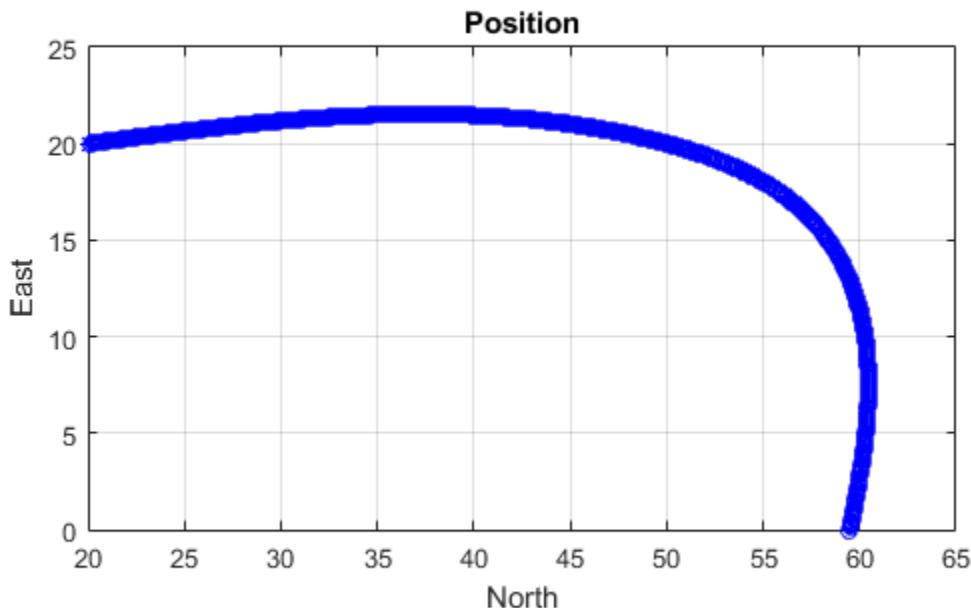
The trajectory object outputs the current position, velocity, acceleration, and angular velocity at each call. Call `trajectory` in a loop and plot the position over time. Cache the other outputs.

```
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(traj.SamplesPerFrame/traj.SampleRate)  
count = count + 1;  
end
```



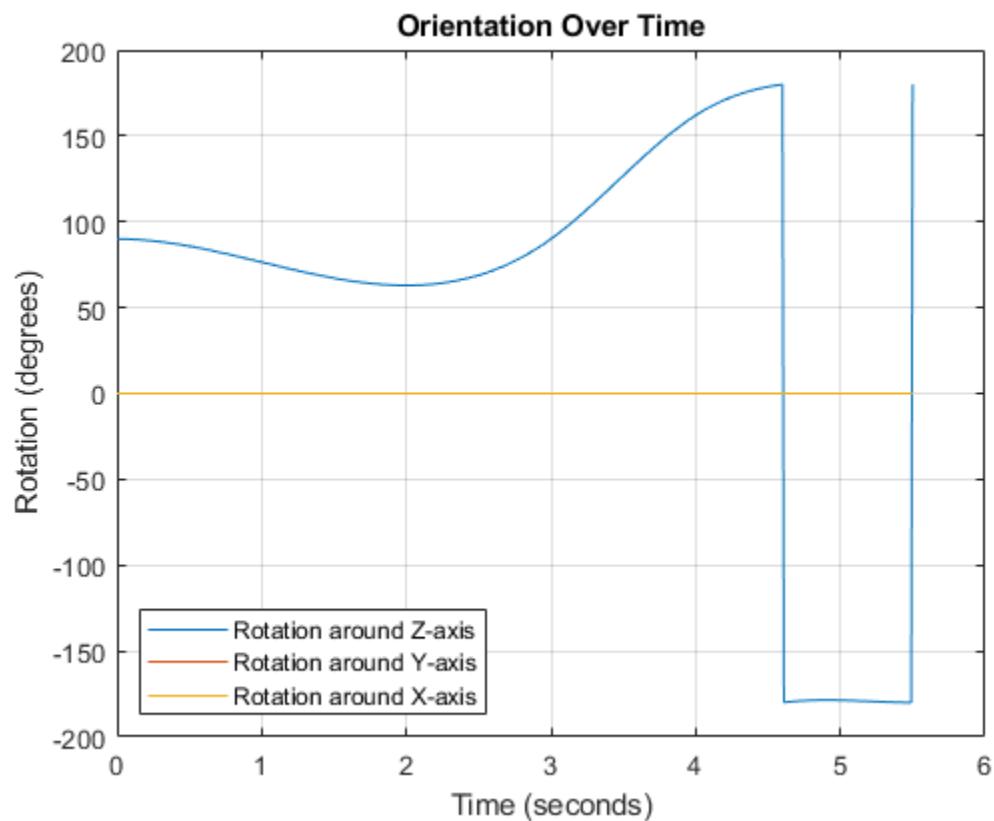
Inspect the orientation, velocity, acceleration, and angular velocity over time. The **waypointTrajectory** System object™ creates a path through the specified constraints that minimized acceleration and angular velocity.

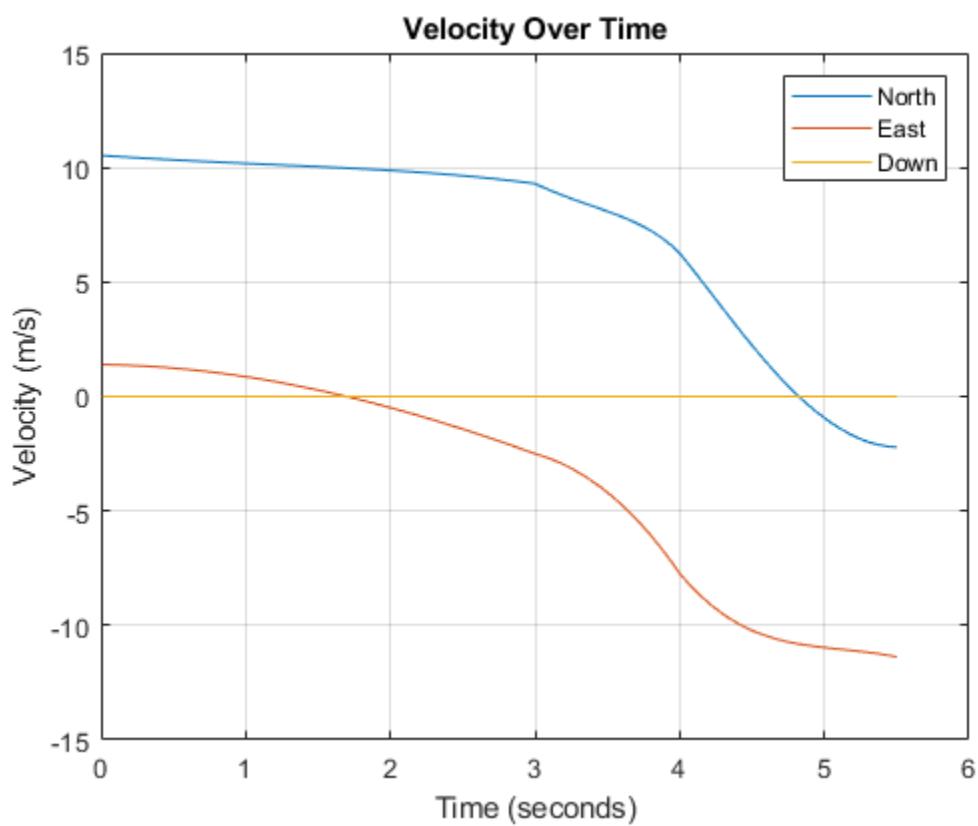
```
figure(2)  
timeVector = 0:(1/traj.SampleRate):tInfo.TimeOfArrival(end);  
eulerAngles = eulerd([tInfo.Orientation{1};orient], 'ZYX','frame');  
plot(timeVector,eulerAngles(:,1), ...  
     timeVector,eulerAngles(:,2), ...  
     timeVector,eulerAngles(:,3));  
title('Orientation Over Time')  
legend('Rotation around Z-axis', ...  
      'Rotation around Y-axis', ...  
      'Rotation around X-axis', ...  
      'Location','southwest')  
xlabel('Time (seconds)')  
ylabel('Rotation (degrees)')  
grid on
```

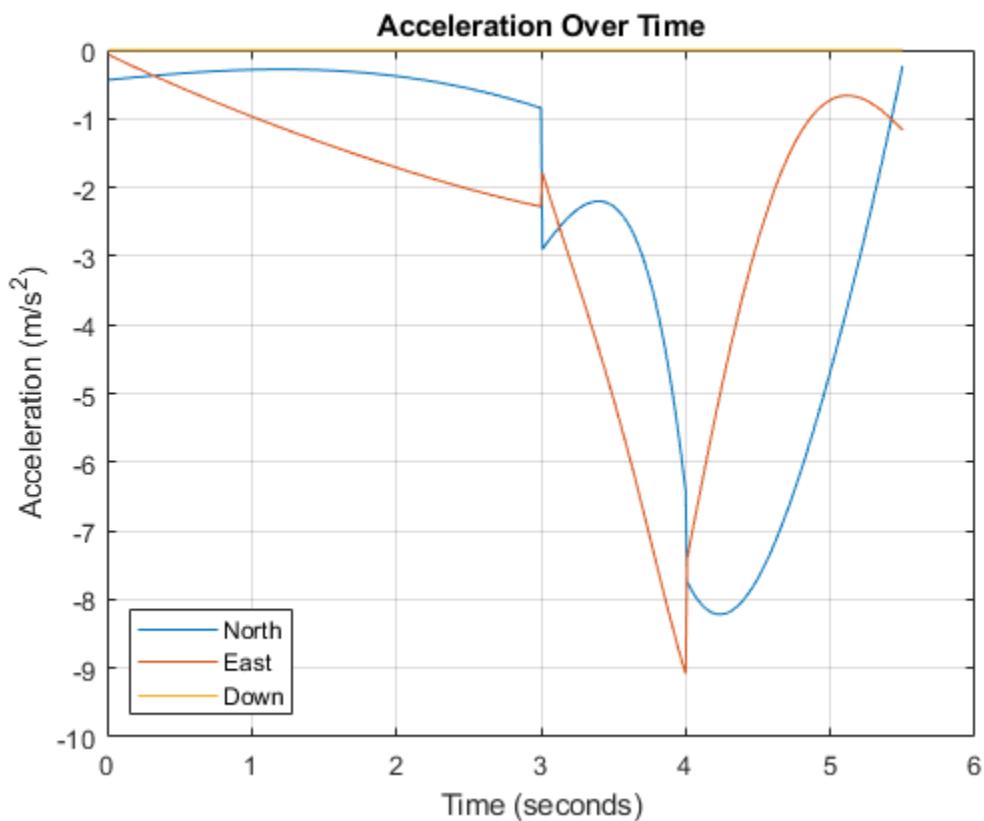
```
figure(3)
plot(timeVector(2:end),vel(:,1), ...
      timeVector(2:end),vel(:,2), ...
      timeVector(2:end),vel(:,3));
title('Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Velocity (m/s)')
grid on

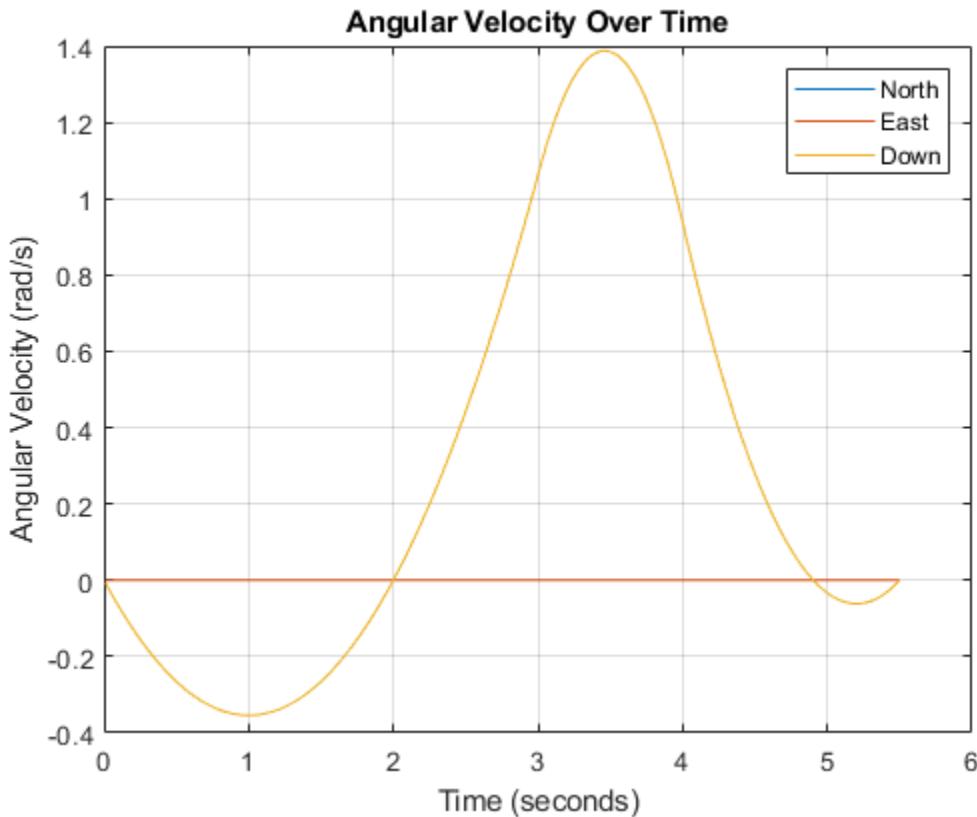
figure(4)
plot(timeVector(2:end),acc(:,1), ...
      timeVector(2:end),acc(:,2), ...
      timeVector(2:end),acc(:,3));
title('Acceleration Over Time')
legend('North','East','Down','Location','southwest')
xlabel('Time (seconds)')
ylabel('Acceleration (m/s^2)')
grid on

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
```









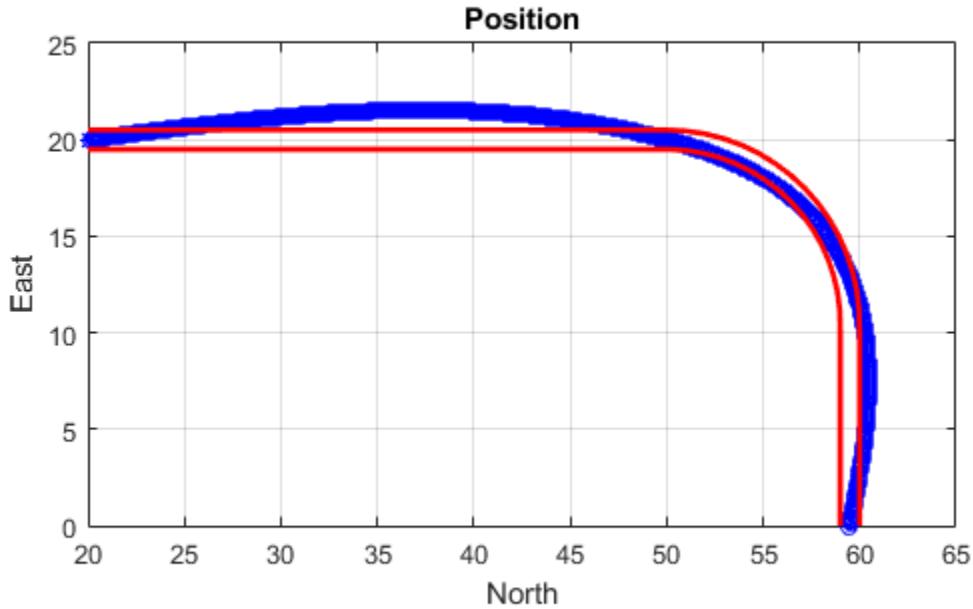
### Restrict Arc Trajectory Within Preset Bounds

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);

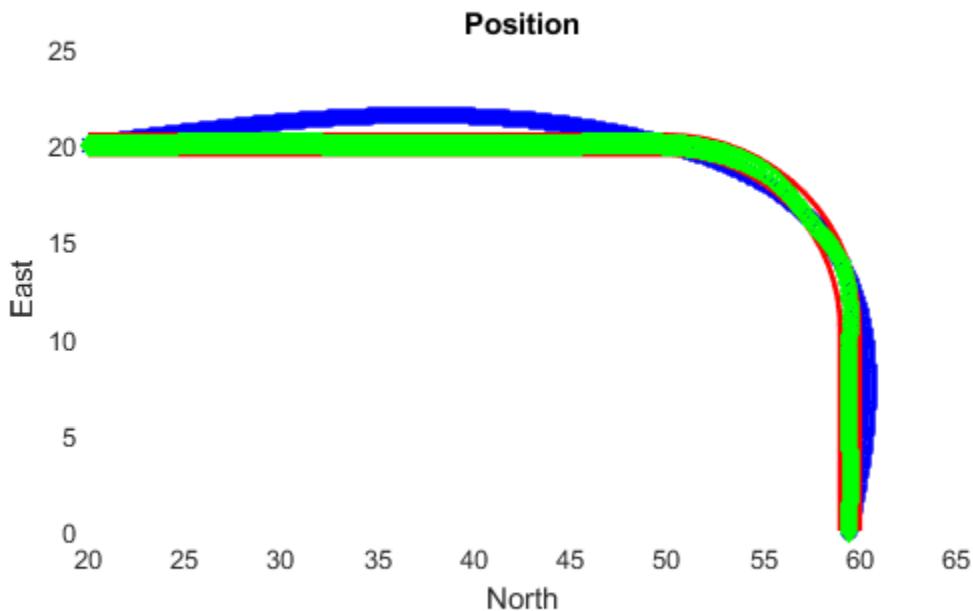
figure(1)
plot(tInfo.Waypoints(1,1),tInfo.Waypoints(1,2), 'b*')

count = 1;
```

```

while ~isDone(trajectory)
    [pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajectory();
    plot(pos(1),pos(2),'gd')
    pause(trajectory.SamplesPerFrame/trajectory.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/trajectory.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

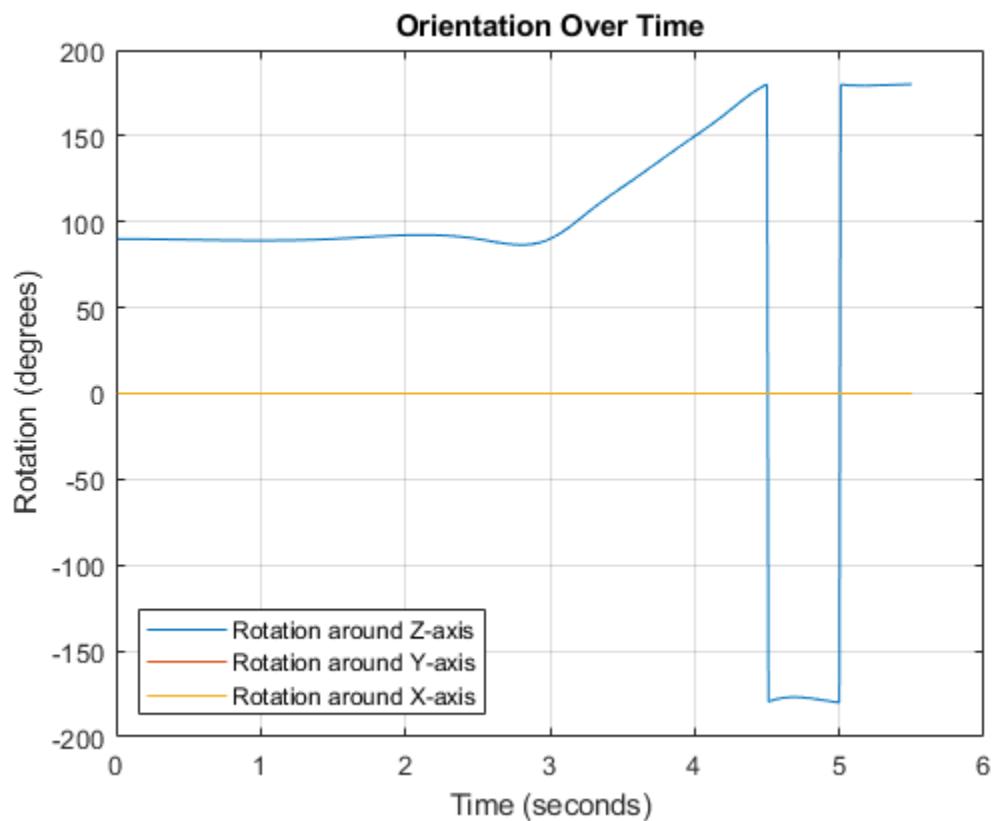
figure(3)

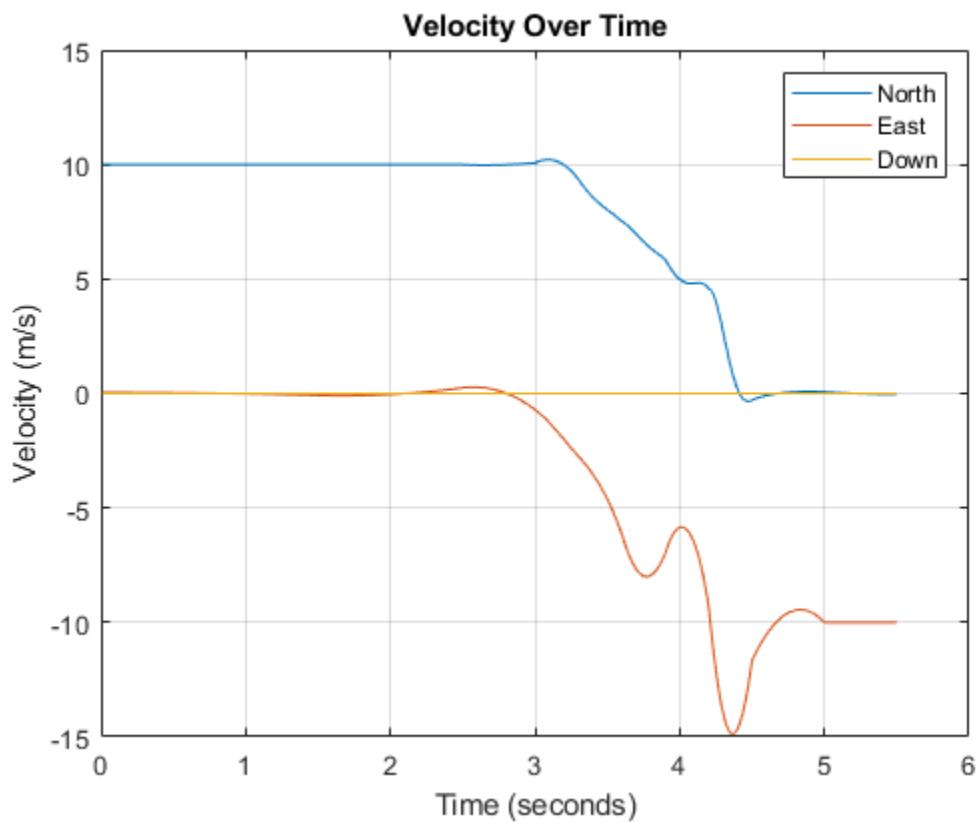
```

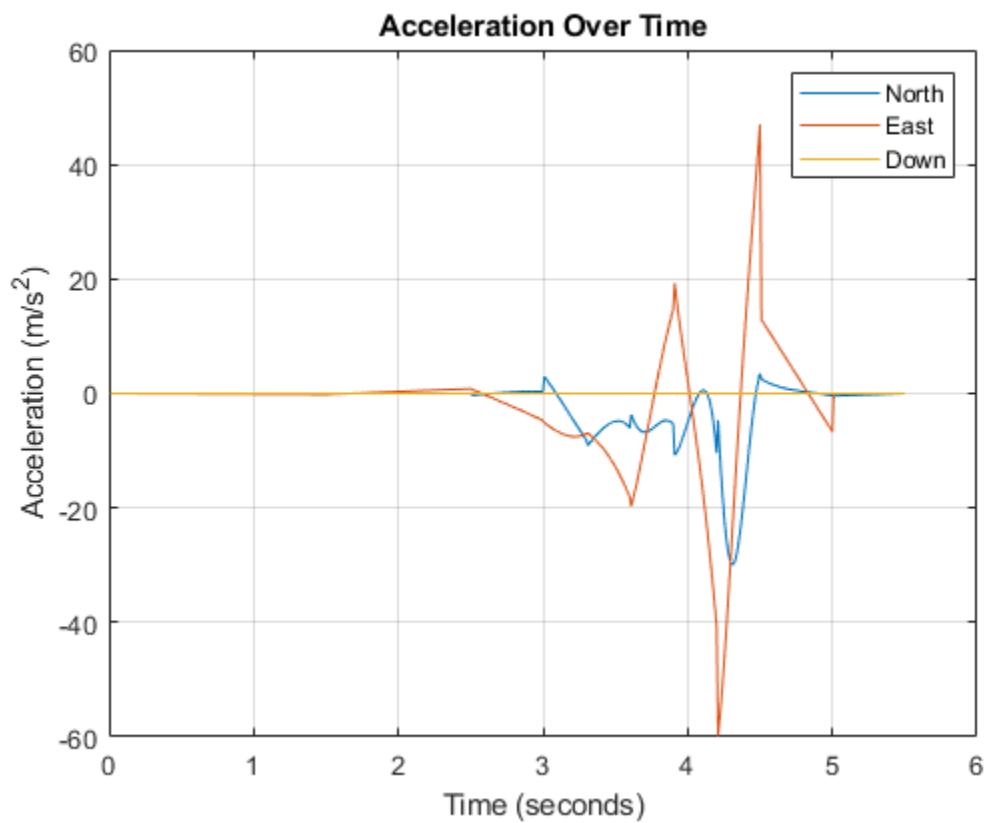
```
plot(timeVector(2:end),vel(:,1), ...
      timeVector(2:end),vel(:,2), ...
      timeVector(2:end),vel(:,3));
title('Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Velocity (m/s)')
grid on

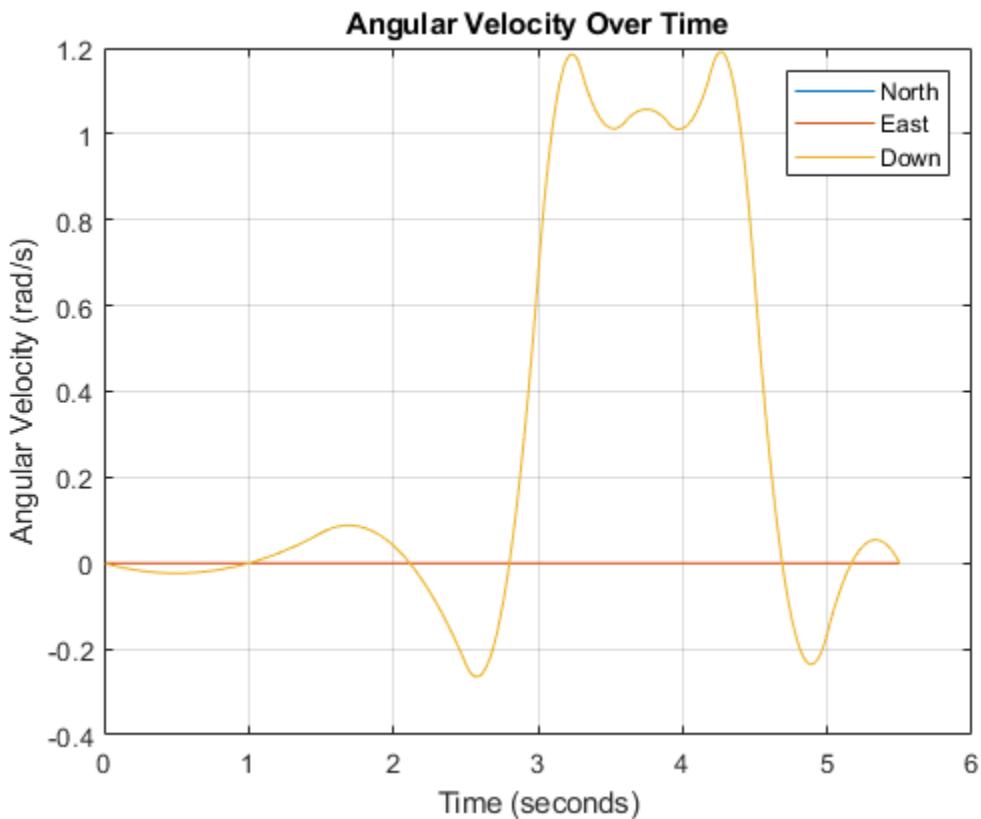
figure(4)
plot(timeVector(2:end),acc(:,1), ...
      timeVector(2:end),acc(:,2), ...
      timeVector(2:end),acc(:,3));
title('Acceleration Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Acceleration (m/s^2)')
grid on

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.

## 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

[Platform](#) | [kinematicTrajectory](#) | [trackingScenario](#)

**Introduced in R2018b**

# perturb

Apply perturbations to object

## Syntax

```
offsets = perturb(obj)
```

## Description

`offsets = perturb(obj)` applies the perturbations defined on the object, `obj`. 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)
```

Property	Type	Value
"Waypoints"	"Normal"	{[ 1]} {[ 1]}
"TimeOfArrival"	"None"	{[NaN]} {[NaN]}

```
perturbs2 = perturbations(traj,'TimeOfArrival','Selection',[0;1],[0;2])
```

Property	Type	Value
"Waypoints"	"Normal"	{[ 1 ]}
"TimeOfArrival"	"Selection"	{1x2 cell} {[ 1 ]}

Perturb the trajectory.

```
offsets = perturb(traj)

offsets=2×1 struct array with fields:
    Property
    Offset
    PerturbedValue
```

The `Waypoints` property and the `TimeOfArrival` property have changed.

```
traj.Waypoints
```

```
ans = 2×3
```

```
1.8674    1.0203    0.7032
2.3154   -0.3207    0.0999
```

```
traj.TimeOfArrival
```

```
ans = 2×1
```

```
0
2
```

### Perturb Accuracy of insSensor

Create an `insSensor` object.

```
sensor = insSensor

sensor =
    insSensor with properties:

        RollAccuracy: 0.2          deg
        PitchAccuracy: 0.2          deg
        YawAccuracy: 1              deg
        PositionAccuracy: 1          m
        VelocityAccuracy: 0.05      m/s
        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=5×3 table
  Property          Type            Value
  _____
  "RollAccuracy"   "Selection"    {1x3 cell}    {1x3 double}
  "PitchAccuracy"  "None"        {[  NaN]}     {[  NaN]}
  "YawAccuracy"    "None"        {[  NaN]}     {[  NaN]}
  "PositionAccuracy"  "None"      {[  NaN]}     {[  NaN]}
  "VelocityAccuracy"  "None"      {[  NaN]}     {[  NaN]}

```

Perturb the `sensor` object using the `perturb` function.

```

rng(2020)
perturb(sensor);
sensor

sensor =
  insSensor with properties:

    RollAccuracy: 0.5           deg
    PitchAccuracy: 0.2          deg
    YawAccuracy: 1              deg
    PositionAccuracy: 1         m
    VelocityAccuracy: 0.05      m/s
    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 includes:

- `waypointTrajectory`
- `geoTrajectory`
- `kinematicTrajectory`
- `insSensor`

- radarEmitter
- radarSensor
- monostaticRadarSensor
- sonarEmitter
- sonarSensor
- irSensor
- monostaticLidarSensor

## 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 R2020b**

# 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,'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,'Uniform',minVal,maxVal)` defines the `property` perturbation offset drawn from a uniform distribution on an interval `[minVal, maxValue]`.

`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=2x3 table		Type	Value
Property			
"Waypoints"	"None"	{[NaN]}	{[NaN]}
"TimeOfArrival"	"None"	{[NaN]}	{[NaN]}

### Perturb Accuracy of insSensor

Create an `insSensor` object.

```
sensor = insSensor

sensor =
  insSensor with properties:

    RollAccuracy: 0.2          deg
    PitchAccuracy: 0.2          deg
    YawAccuracy: 1              deg
    PositionAccuracy: 1          m
    VelocityAccuracy: 0.05      m/s
    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=5x3 table		Type	Value
Property			
"RollAccuracy"	"Selection"	{1x3 cell}	{1x3 double}
"PitchAccuracy"	"None"	{[ NaN]}	{[ NaN]}
"YawAccuracy"	"None"	{[ NaN]}	{[ NaN]}
"PositionAccuracy"	"None"	{[ NaN]}	{[ NaN]}
"VelocityAccuracy"	"None"	{[ NaN]}	{[ NaN]}

Perturb the `sensor` object using the `perturb` function.

```
rng(2020)
perturb(sensor);
sensor

sensor =
insSensor with properties:

    RollAccuracy: 0.5          deg
    PitchAccuracy: 0.2         deg
    YawAccuracy: 1             deg
    PositionAccuracy: 1        m
    VelocityAccuracy: 0.05     m/s
    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)
```

Property	Type	Value
"Waypoints"	"Normal"	{[ 1]} {[ 1]}
"TimeOfArrival"	"None"	{[NaN]} {[NaN]}

```
perturbs2 = perturbations(traj,'TimeOfArrival','Selection',[{0;1}, {0;2}])
```

Property	Type	Value
"Waypoints"	"Normal"	{[ 1 ]} {1x2 cell}
"TimeOfArrival"	"Selection"	{[ 1 ]} {1x2 double}

Perturb the trajectory.

```
offsets = perturb(traj)

offsets=2×1 struct array with fields:
    Property
    Offset
    PerturbedValue
```

The **Waypoints** property and the **TimeOfArrival** property have changed.

```
traj.Waypoints

ans = 2×3

    1.8674    1.0203    0.7032
    2.3154   -0.3207    0.0999

traj.TimeOfArrival

ans = 2×1

    0
    2
```

## Input Arguments

**obj — Object to be perturbed**  
objects

Object to be perturbed, specified as an object. The objects that you can perturb includes:

- `waypointTrajectory`
- `kinematicTrajectory`
- `geoTrajectory`
- `insSensor`
- `radarEmitter`
- `radarSensor`
- `monostaticRadarSensor`
- `rcsSignature`
- `sonarEmitter`

- `sonarSensor`
- `monostaticLidarSensor`

**property — Perturbable property**

property name

Perturbable property, specified as a property name. For each object listed in `obj`, the corresponding perturbable properties are listed in “More About” on page 3-244.

**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 distribution**

scalar | vector | matrix

Mean of 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 distribution**

nonnegative scalar | vector of nonnegative scalar | matrix of nonnegative scalar

Standard deviation of 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.

**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", "Uniform", or "Custom".
- **Value** — Perturbation values, returned as a cell array.

## More About

### **waypointTrajectory Perturbable Properties**

The perturbable properties of `waypointTrajectory` are:

- `Waypoints`
- `TimeOfArrival`

Other properties of `waypointTrajectory` cannot be perturbed.

### **geoTrajectory Perturbable Properties**

The perturbable properties of `waypointTrajectory` are:

- `Waypoints`
- `TimeOfArrival`

Other properties of `waypointTrajectory` cannot be perturbed.

### **kinematicTrajectory Perturbable Properties**

The perturbable properties of `kinematicTrajectory` are:

- `Position`
- `Velocity`

Other properties of `kinematicTrajectory` cannot be perturbed.

### **insSensor Perturbable Properties**

The perturbable properties of `insSensor` are:

- `RollAccuracy`
- `PitchAccuracy`

- YawAccuracy
- PositionAccuracy
- VelocityAccuracy

Other properties of `insSensor` cannot be perturbed.

### **radarEmitter Perturbable Properties**

The perturbable properties of `radarEmitter` are:

- EIRP
- CeneterFrequency
- Bandwidth
- WaveformTypes
- ProcessingGain

Other properties of `radarEmitter` cannot be perturbed.

### **radarSensor Perturbable Properties**

The perturbable properties of `radarSensor` are:

- Sensitivity
- FalseAlarmRate
- DetectionThreshold
- CeneterFrequency
- Bandwidth
- WaveformTypes
- CofusionMatrix
- AzimuthResolution
- ElevationResolution
- RangeResolution
- RangeRateResolution
- AzimuthBiasFraction
- ElevationBiasFraction
- RangeBiasFraction
- RangeRateBiasFraction

Other properties of `radarSensor` cannot be perturbed.

### **monostaticRadarSensor Perturbable Properties**

The perturbable properties of `monostaticRadarSensor` are:

- DetectionProbability
- FalseAlarmRate
- ReferenceRange

- ReferenceRCS
- HasInterference
- AzimuthResolution
- ElevationResolution
- RangeResolution
- RangeRateResolution
- AzimuthBiasFraction
- ElevationBiasFraction
- RangeBiasFraction
- RangeRateBiasFraction

Other properties of monostaticRadarSensor cannot be perturbed.

#### **sonarEmitter Perturbable Properties**

The perturbable properties of sonarEmitter are:

- SourceLevel
- CenterFrequency
- Bandwidth
- WaveformType
- ProcessingGain

Other properties of sonarEmitter cannot be perturbed.

#### **sonarSensor Perturbable Properties**

The perturbable properties of sonarSensor are:

- FalseAlarmRate
- AmbientNoiseLevel
- CeneterFrequency
- Bandwidth
- WaveformTypes
- CofusionMatrix
- AzimuthResolution
- ElevationResolution
- RangeResolution
- RangeRateResolution
- AzimuthBiasFraction
- ElevationBiasFraction
- RangeBiasFraction
- RangeRateBiasFraction

Other properties of sonarSensor cannot be perturbed.

## **irSensor Perturbable Properties**

The perturbable properties of `irSensor` are:

- `FocalLength`
- `MinClassificationArea`
- `CutoffFrequency`
- `DetectorArea`
- `Detectivity`
- `NoiseEquivalentBandwidth`
- `FalseAlarmRate`
- `AzimuthResolution`
- `ElevationResolution`
- `AzimuthBiasFraction`
- `ElevationBiasFraction`

Other properties of `irSensor` cannot be perturbed.

## **monostaticLidarSensor Perturbable Properties**

The perturbable properties of `irSensor` are:

- `RangeAccuracy`

Other properties of `monostaticLidarSensor` cannot be perturbed.

## **See Also**

`perturb`

**Introduced in R2020b**

# 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

`kinematicTrajectory` | `waypointTrajectory`

**Introduced in R2018b**

# 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**

`waypointTrajectory`

**Introduced in R2020a**

# monostaticRadarSensor

Generate radar detections for tracking scenario

## Description

The `monostaticRadarSensor` System object generates detections of targets by a monostatic surveillance scanning radar. You can use the `monostaticRadarSensor` object in a scenario containing moving and stationary platforms such as one created using `trackingScenario`. The `monostaticRadarSensor` object can simulate real detections with added random noise and also generate false alarm detections. In addition, you can use the detections generated by this object as input to trackers such as `trackerGNN` or `trackerTOMHT`.

This object enable you to configure a scanning radar. A scanning radar changes its look angle 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`. If the scanning 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.

Using a single-exponential mode, 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.

To generate radar detections:

- 1 Create the `monostaticRadarSensor` 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

```
sensor = monostaticRadarSensor(SensorIndex)
sensor = monostaticRadarSensor(SensorIndex,Name,Value)

sensor = monostaticRadarSensor(SensorIndex,'No scanning')
sensor = monostaticRadarSensor(SensorIndex,'Raster')
sensor = monostaticRadarSensor(SensorIndex,'Rotator')
sensor = monostaticRadarSensor(SensorIndex,'Sector')
```

## Description

`sensor = monostaticRadarSensor(SensorIndex)` creates a radar detection generator object with a specified sensor index, `SensorIndex`, and default property values.

`sensor = monostaticRadarSensor(SensorIndex, Name, Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes. For example, `monostaticRadarSensor(1, 'DetectionCoordinates', 'Sensor rectangular')` creates a radar detection generator that reports detections in the sensor Cartesian coordinate system with sensor index equal to 1.

`sensor = monostaticRadarSensor(SensorIndex, 'No scanning')` is a convenience syntax that creates a `monostaticRadarSensor` that only points along the radar antenna boresight direction. No mechanical or electronic scanning is performed. This syntax sets the `ScanMode` property to 'No scanning'.

`sensor = monostaticRadarSensor(SensorIndex, 'Raster')` is a convenience syntax that creates a `monostaticRadarSensor` object that mechanically scans a raster pattern. The raster span is 90° in azimuth from -45° to +45° and in elevation from the horizon to 10° above the horizon. See "Convenience Syntaxes" on page 3-271 for the properties set by this syntax.

`sensor = monostaticRadarSensor(SensorIndex, 'Rotator')` is a convenience syntax that creates a `monostaticRadarSensor` object that mechanically scans 360° 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 3-271 for the properties set by this syntax.

`sensor = monostaticRadarSensor(SensorIndex, 'Sector')` is a convenience syntax to create a `monostaticRadarSensor` object that mechanically scans a 90° azimuth sector from -45° to +45°. 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 3-271 for the properties set by this syntax.

## 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.

### **SensorIndex — Unique sensor identifier**

positive integer

Unique sensor identifier, specified as a positive integer. This property distinguishes detections that come from different sensors in a multi-sensor system. When creating a `monostaticRadarSensor` system object, you must either specify the `SensorIndex` as the first input argument in the creation syntax, or specify it as the value for the `SensorIndex` property in the creation syntax.

Data Types: `double`

**UpdateRate — Sensor update rate**

1 (default) | positive scalar

Sensor update rate, specified as a positive scalar. This interval must be an integer multiple of the simulation time interval defined by `trackingScenario`. The `trackingScenario` object calls the radar scanning sensor at simulation time intervals. The radar generates new detections at intervals defined by the reciprocal of the `UpdateRate` property. Any update requested to the sensor between update intervals contains no detections. Units are in hertz.

Example: 5

Data Types: double

**MountingLocation — Sensor location on platform**

[0 0 0] (default) | 1-by-3 real-valued vector

Sensor location on platform, specified as a 1-by-3 real-valued vector. This property defines the coordinates of the sensor with respect to the platform origin. The default value specifies that the sensor origin is at the origin of its platform. Units are in meters.

Example: [ .2 0.1 0]

Data Types: double

**MountingAngles — Orientation of sensor**

[0 0 0] (default) | 3-element real-valued vector

Orientation of the sensor 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 sensor axes. The three elements define the rotations around the z-, y-, and x-axes, 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 the 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 field of view `azfov` must lie in the interval (0,360]. The elevation field of view `elfov` must lie in the interval (0,180].

Example: [14;7]

Data Types: double

**HasRangeAmbiguities — Enable range ambiguities**

false (default) | true

Enable range ambiguities, specified as `false` or `true`. Set this property to `true` to enable range ambiguities by the sensor. In this case, the sensor cannot resolve range ambiguities and target ranges beyond the `MaxUnambiguousRange` are wrapped into the interval `[0 MaxUnambiguousRange]`. When `false`, targets are reported at their unambiguous range.

Data Types: logical

**MaxUnambiguousRange — Maximum unambiguous detection range**

100e3 (default) | positive scalar

Maximum unambiguous range, specified as a positive scalar. 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 applies to true target detections when you set the `HasRangeAmbiguities` property to `true`.

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 for false alarms.

Units are in meters.

Example: 5e3

**Dependencies**

To enable this property, set the `HasRangeAmbiguities` property to `true` or set the `HasFalseAlarms` property to `true`.

Data Types: double

**HasRangeRateAmbiguities — Enable range-rate ambiguities**

false (default) | true

Enable range-rate ambiguities, specified as `false` or `true`. Set to `true` to enable range-rate ambiguities by the sensor. When `true`, the sensor does not resolve range rate ambiguities and target range rates beyond the `MaxUnambiguousRadialSpeed` are wrapped into the interval  $[-\text{MaxUnambiguousRadialSpeed}, \text{MaxUnambiguousRadialSpeed}]$ . When `false`, targets are reported at their unambiguous range rate.

**Dependencies**

To enable this property, set the `HasRangeRate` property to `true`.

Data Types: logical

**MaxUnambiguousRadialSpeed — Maximum unambiguous radial speed**

200 (default) | positive scalar

Maximum unambiguous radial speed, specified as a positive scalar. 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 applies to true target detections when you set `HasRangeRateAmbiguities` property to `true`.

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 for which false alarms can be generated.

Units are in meters per second.

## Dependencies

To enable this property, set `HasRangeRate` and `HasRangeRateAmbiguities` to `true` and/or set `HasRangeRate` and `HasFalseAlarms` to `true`.

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
<code>'Mechanical'</code>	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.
<code>'Electronic'</code>	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.
<code>'Mechanical and electronic'</code>	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.
<code>'No scanning'</code>	The radar beam points along the antenna boresight defined by the <code>MountingAngles</code> property.

Example: `'No scanning'`

## 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 sensor**

`scalar` | `real-valued 2-by-1 vector`

This property is read-only.

Look angle of sensor, 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.

<b>ScanMode</b>	<b>LookAngle</b>
<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>

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.

**DetectionProbability — Probability of detecting a target**

0.9 | positive scalar less than or equal to 1

Probability of detecting a target, specified as a positive scalar less than or equal to one. This quantity defines the probability of detecting a target with a radar cross-section, **ReferenceRCS**, at the reference detection range, **ReferenceRange**.

Example: 0.95

Data Types: double

**FalseAlarmRate — False alarm rate**

1e-6 (default) | positive scalar

False alarm report rate within each radar resolution cell, specified as a positive scalar in the range  $[10^{-7}, 10^{-3}]$ . Units are dimensionless. Resolution cells are determined from the **AzimuthResolution** and **RangeResolution** properties, and the **ElevationResolution** and **RangeRateResolution** properties when they are enabled.

Example: 1e-5

Data Types: double

**ReferenceRange — Reference range for given probability of detection**

100e3 (default) | positive scalar

Reference range for the given probability of detection and the given reference radar cross-section (RCS), specified as a positive scalar. The reference range is the range at which a target having a radar cross-section specified by **ReferenceRCS** is detected with a probability of detection specified by **DetectionProbability**. Units are in meters.

Example: 25e3

Data Types: double

**ReferenceRCS — Reference radar cross-section for given probability of detection**

0 (default) | scalar

Reference radar cross-section (RCS) for given a probability of detection and reference range, specified as a scalar. The reference RCS is the RCS value at which a target is detected with probability specified by **DetectionProbability** at **ReferenceRange**. Units are in dBsm.

Example: -10

Data Types: double

**RadarLoopGain — Radar loop gain**

scalar

This property is read-only.

Radar loop gain, returned as a 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*. The function is

$$SNR = \text{RadarLoopGain} + RCS - 40\log_{10}(R) \quad (3-1)$$

where  $SNR$  and  $RCS$  are in dB and dBsm, respectively, and range is in meters. Radar loop gain is in dB.

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 sensor that can estimate target elevation and scan in elevation.

Data Types: logical

**HasRangeRate — Enable radar to measure range rate**

false (default) | true

Enable the radar to measure target range rates, specified as `false` or `true`. Set this property to `true` to model a radar sensor that can measure target range rate. Set this property to `false` to model a radar sensor that cannot measure range rate.

Data Types: logical

**AzimuthResolution — Azimuth resolution of radar**

1 (default) | positive 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 two targets. The azimuth resolution is typically the 3dB downpoint of the azimuth angle beamwidth of the radar. Units are in degrees.

Data Types: double

**ElevationResolution — Elevation resolution of radar**

1 (default) | positive 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 two targets. The elevation resolution is typically the 3dB-downpoint in elevation angle beamwidth of the radar. Units are in degrees.

**Dependencies**

To enable this property, set the `HasElevation` property to `true`.

Data Types: double

**RangeResolution — Range resolution of radar**

100 (default) | positive scalar

Range resolution of the radar, specified as a positive scalar. The range resolution defines the minimum separation in range at which the radar can distinguish between two targets. Units are in meters.

Data Types: double

**RangeRateResolution — Range rate resolution of radar**

10 (default) | positive scalar

Range rate resolution of the radar, specified as a positive 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 property, set the `HasRangeRate` property to `true`.

Data Types: double

**AzimuthBiasFraction — Azimuth bias fraction**

0.1 (default) | nonnegative scalar

Azimuth bias fraction of the radar, specified as a nonnegative scalar. The azimuth bias is expressed as a fraction of the azimuth resolution specified in `AzimuthResolution`. This value sets a lower bound on the azimuthal accuracy of the radar. This value is dimensionless.

Data Types: double

**ElevationBiasFraction — Elevation bias fraction**

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 value of the `ElevationResolution` property. This value sets a lower bound on the elevation accuracy of the radar. This value 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 in `RangeResolution`. This property sets a lower bound on the range accuracy of the radar. This value 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 in `RangeRateResolution`. This property sets a lower bound on the range-rate accuracy of the radar. This value is dimensionless.

**Dependencies**

To enable this property, set the `HasRangeRate` property to `true`.

Data Types: double

**HasINS — Enable inertial navigation system (INS) input**

false (default) | true

Enable the optional input argument that passes the current estimate of the sensor platform pose to the sensor, specified as `false` or `true`. When `true`, pose information is added to the `MeasurementParameters` structure of the reported detections. Pose information lets tracking and fusion algorithms estimate the state of the target detections in the north-east-down (NED) frame.

Data Types: logical

**HasNoise — Enable addition of noise to radar sensor measurements**  
`true` (default) | `false`

Enable addition of noise to radar sensor measurements, specified as `true` or `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 object still computes the `MeasurementNoise` property of each detection.

Data Types: logical

**HasFalseAlarms — Enable creating false alarm radar detections**  
`true` (default) | `false`

Enable creating false alarm radar measurements, specified as `true` or `false`. Set this property to `true` to report false alarms. Otherwise, only actual detections are reported.

Data Types: logical

**HasOcclusion — Enable occlusion from extended objects**  
`true` (default) | `false`

Enable occlusion from extended objects, specified as `true` or `false`. Set this property 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 this property 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

**MaxNumDetectionsSource — Source of maximum number of detections reported**  
`'Auto'` (default) | `'Property'`

Source of maximum number of detections reported by the sensor, specified as `'Auto'` or `'Property'`. When this property is set to `'Auto'`, the sensor reports all detections. When this property is set to `'Property'`, the sensor reports up to the number of detections specified by the `MaxNumDetections` property.

Data Types: char

**MaxNumDetections — Maximum number of reported detections**  
`50` (default) | positive integer

Maximum number of detections reported by the sensor, specified as a positive integer. Detections are reported in order of distance to the sensor until the maximum number is reached.

**Dependencies**

To enable this property, set the `MaxNumDetectionsSource` property to 'Property'.

Data Types: double

**DetectionCoordinates — Coordinate system of reported detections**

'Body' (default) | 'Scenario' | 'Sensor rectangular' | 'Sensor spherical'

Coordinate system of reported detections, specified as:

- 'Scenario' — Detections are reported in the rectangular scenario coordinate frame. The scenario coordinate system is defined as the local NED 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 radar 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 radar sensor and aligned with the orientation of the radar on the platform.

Example: 'Sensor spherical'

Data Types: char

**HasInterference — Enable RF interference input**

false (default) | true

Enable RF interference input, specified as `false` or `true`. When `true`, you can add RF interference using an input argument of the object.

Data Types: logical

**Bandwidth — Radar waveform bandwidth**

positive scalar

Radar waveform bandwidth, specified as a positive scalar. Units are in hertz.

Example: 100e3

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

**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 dBm.

Example: -10

Data Types: double

## Usage

### Syntax

```
dets = sensor(targets,simTime)
dets = sensor(targets,ins,simTime)
dets = sensor(targets,interference,simTime)
[dets,numDets,config] = sensor(____)
```

#### Description

`dets = sensor(targets,simTime)` creates radar detections, `dets`, from sensor measurements taken of `targets` at the current simulation time, `simTime`. The sensor can generate detections for multiple targets simultaneously.

`dets = sensor(targets,ins,simTime)` also specifies the INS-estimated pose information, `ins`, for the sensor platform. INS information is used by tracking and fusion algorithms to estimate the target positions in the NED frame.

To enable this syntax, set the `HasINS` property to `true`.

`dets = sensor(targets,interference,simTime)` also specifies an interference signal, `interference`.

To enable this syntax, set the `HasInterference` property to `true`.

`[dets,numDets,config] = sensor(____)` also returns the number of valid detections reported, `numDets`, and the configuration of the sensor, `config`, at the current simulation time. You can use these output arguments with any of the previous input syntaxes.

### Input Arguments

#### targets — Tracking scenario target poses

structure | structure array

Tracking scenario target poses, specified as a structure or array of structures. Each structure corresponds to a target. You can generate this structure using the `targetPoses` method of a platform. You can also create such a structure manually. The table shows the required fields of the structure:

Field	Description
PlatformID	Unique identifier for the platform, specified as a scalar 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	Position of 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 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 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 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 coordinate system.

#### **simTime — Current simulation time**

nonnegative scalar

Current simulation time, specified as a positive scalar. The `trackingScenario` object calls the scan radar sensor at regular time intervals. The radar sensor generates new detections 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 sensor between update intervals contain no detections. Units are in seconds.

Data Types: `double`

#### **ins — Platform pose from INS**

structure

Platform pose information from an inertial navigation system (INS) is a structure with these fields:

Field	Definition
Position	Position in the navigation frame, specified as a real-valued 1-by-3 vector. Units are in meters.

<b>Velocity</b>	Velocity in the navigation frame, specified as a real-valued 1-by-3 vector. Units are in meters per second.
<b>Orientation</b>	Orientation with respect to the navigation 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.

**Dependencies**

To enable this argument, set the `HasINS` property to `true`.

Data Types: `struct`

**interference — Interfering or jamming signal**

array of `radarEmission` objects

Interfering or jamming signal, specified as an array of `radarEmission` objects.

**Dependencies**

To enable this argument, set the `HasInterference` property to `true`.

Data Types: `double`

Complex Number Support: Yes

**Output Arguments****dets — sensor detections**

cell array of `objectDetection` objects

Sensor detections, returned as a cell array of `objectDetection` objects. For a high level view of object detections, see `objectDetection` objects. Each object has these properties but the contents of the properties depend on the specific sensor. For the `monostaticRadarSensor`, see "Object Detections" on page 3-269.

Property	Definition
<code>Time</code>	Measurement time
<code>Measurement</code>	Object measurements
<code>MeasurementNoise</code>	Measurement noise covariance matrix
<code>SensorIndex</code>	Unique ID of the sensor
<code>ObjectClassID</code>	Object classification
<code>ObjectAttributes</code>	Additional information passed to tracker
<code>MeasurementParameters</code>	Parameters used by initialization functions of nonlinear Kalman tracking filters

For the `Measurement` and `MeasurementNoise` are reported in the coordinate system specified by the `DetectionCoordinates` property.

**numDets — Number of detections**

nonnegative integer

Number of detections reported, returned as a nonnegative integer.

- When the `MaxNumDetectionsSource` property is set to 'Auto', `numDets` is set to the length of `dets`.
- When the `MaxNumDetectionsSource` property is set to 'Property', `dets` is a cell array with length determined by the `MaxNumDetections` property. The maximum number of detections returned is `MaxNumDetections`. If the number of detections is fewer than `MaxNumDetections`, the first `numDets` elements of `dets` hold valid detections. The remaining elements of `dets` are set to the default value.

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
<code>SensorIndex</code>	Unique sensor index, returned as a positive integer.
<code>IsValidTime</code>	<code>IsValidTime</code> is <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>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.

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 monostaticRadarSensor

`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

### Air-Traffic Control Tower Radar

Simulate a radar scenario.

```
sc = trackingScenario('UpdateRate',1);
```

Create an airport control tower with a surveillance radar located 15 meters above the ground. The radar rotates at 12.5 rpm and its field of view in azimuth is 5 degrees and its field of view in elevation is 10 degrees.

```

rpm = 12.5;
fov = [5;10]; % [azimuth; elevation]
scanrate = rpm*360/60;
updaterate = scanrate/fov(1) % Hz

updaterate = 15

radar = monostaticRadarSensor(1,'Rotator', ...
    'UpdateRate',updaterate, ...
    'MountingLocation',[0 0 -15], ...
    'MaxMechanicalScanRate',scanrate, ...
    'FieldOfView',fov, ...
    'AzimuthResolution',fov(1));
towermotion = kinematicTrajectory('SampleRate',1,'Position',[0 0 0],'Velocity',[0 0 0]);
tower = platform(sc,'ClassID',1,'Trajectory',towermotion);
aircraft1motion = kinematicTrajectory('SampleRate',1,'Position',[10000 0 1000],'Velocity',[-100 0 0]);
aircraft1 = platform(sc,'ClassID',2,'Trajectory',aircraft1motion);
aircraft2motion = kinematicTrajectory('SampleRate',1,'Position',[5000 5000 200],'Velocity',[100 0 0]);
aircraft2 = platform(sc,'ClassID',2,'Trajectory',aircraft2motion);
```

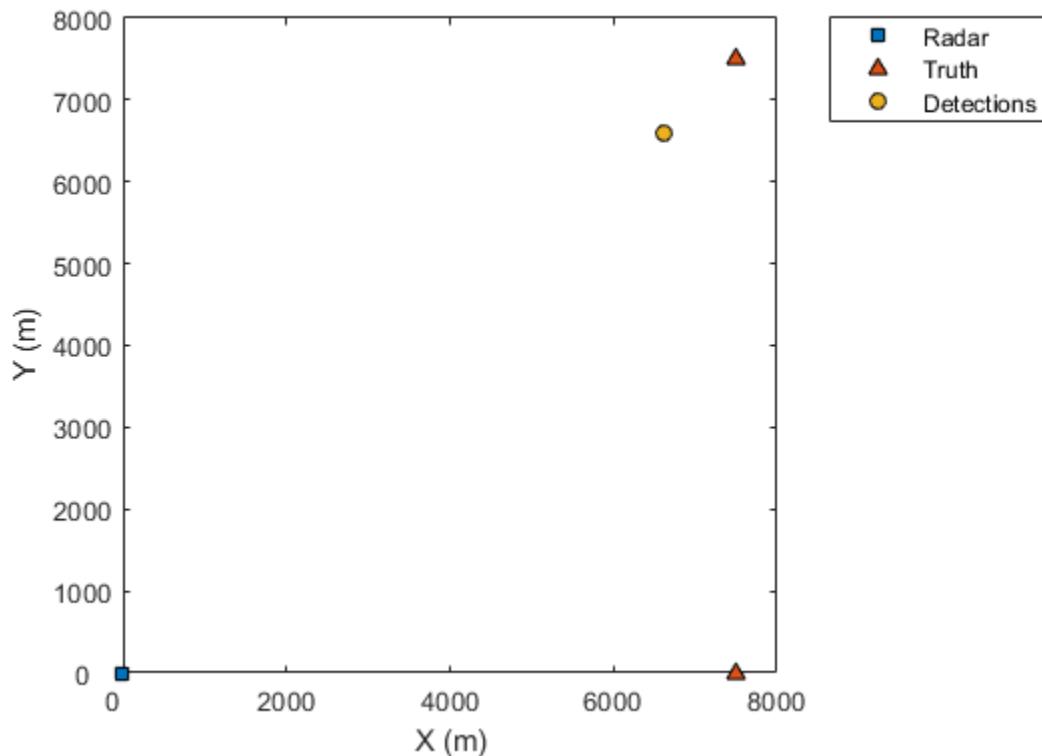
Perform 5 scans.

```

detBuffer = {};
scanCount = 0;
while advance(sc)
    simTime = sc.SimulationTime;
    targets = targetPoses(tower);
    [dets,numDets,config] = radar(targets,simTime);
    detBuffer = [detBuffer;dets];
    if config.IsScanDone
        scanCount = scanCount + 1;
        if scanCount == 5;
            break;
        end
    end
end
```

Plot detections

```
tp = theaterPlot;
clrs = lines(3);
rp = platformPlotter(tp,'DisplayName','Radar','Marker','s',...
    'MarkerFaceColor',clrs(1,:));
pp = platformPlotter(tp,'DisplayName','Truth',...
    'MarkerFaceColor',clrs(2,:));
dp = detectionPlotter(tp,'DisplayName','Detections',...
    'MarkerFaceColor',clrs(3,:));
plotPlatform(rp,[0 0 0])
plotPlatform(pp,[targets(1).Position; targets(2).Position])
if ~isempty(detBuffer)
    detPos = cellfun(@(d)d.Measurement(1:3),detBuffer,...
        'UniformOutput',false);
    detPos = cell2mat(detPos)';
    plotDetection(dp,detPos)
end
```



## More About

### Object Detections

#### Measurements

The sensor measures the coordinates of the target. The **Measurement** and **MeasurementNoise** values are reported in the coordinate system specified by the **DetectionCoordinates** property of the sensor.

When the **DetectionCoordinates** property is '**Scenario**', '**Body**', or '**Sensor rectangular**', the **Measurement** and **MeasurementNoise** values are reported in rectangular coordinates. Velocities are only reported when the range rate property, **HasRangeRate**, is **true**.

When the **DetectionCoordinates** property is '**Sensor spherical**', the **Measurement** and **MeasurementNoise** values are reported in a spherical coordinate system derived from the sensor rectangular coordinate system. Elevation and range rate are only reported when **HasElevation** and **HasRangeRate** are **true**.

Measurements are ordered as [azimuth, elevation, range, range rate]. Reporting of elevation and range rate depends on the corresponding **HasElevation** and **HasRangeRate** property values. Angles are in degrees, range is in meters, and range rate is in meters per second.

#### Measurement Coordinates

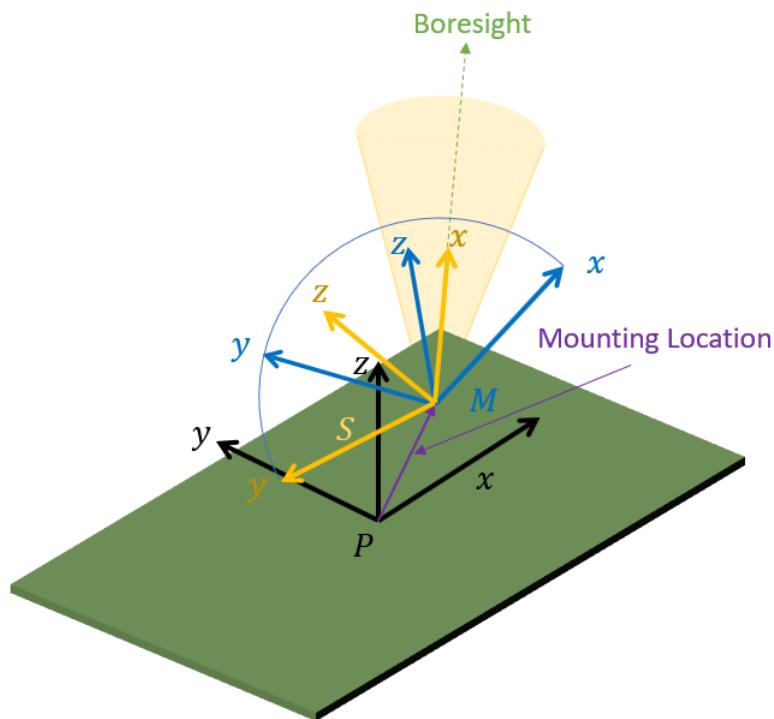
<b>DetectionCoordinates</b>	<b>Measurement and Measurement Noise Coordinates</b>					
'Scenario'	<b>Coordinate Dependence on HasRangeRate</b>					
'Body'	<b>HasRangeRate</b>					
'Sensor rectangular'	<table border="1"> <tr> <td>true</td><td>[x; y; z; vx; vy; vz]</td></tr> <tr> <td>false</td><td>[x; y; z]</td></tr> </table>		true	[x; y; z; vx; vy; vz]	false	[x; y; z]
true	[x; y; z; vx; vy; vz]					
false	[x; y; z]					
'Sensor spherical'	<b>Coordinate Dependence on HasRangeRate and HasElevation</b>					
	<b>HasRangeRate</b>	<b>HasElevation</b>				
	true	true				
	true	false				
	false	true				
	false	false				
	<b>Coordinates</b>					
	[az; el; rng; rr]					
	[az; rng; rr]					
	[az; el; rng]					
	[az; rng]					

#### Measurement Parameters

The **MeasurementParameters** property consists of an array of structures that describe a sequence of coordinate transformations from a child frame to a parent frame or the inverse transformations (see "Frame Rotation"). 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 the 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 form the scenario frame to the platform frame then from 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.

Trivially, if the detections are reported in platform rectangular coordinates and **HasINS** is set to **false**, the transformation consists only of the identity.

The fields of **MeasurementParameters** are shown here. Not all fields have to be present in the structure. The 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 ' <b>'rectangular'</b> '. When detections are reported in spherical coordinates, Frame is set ' <b>'spherical'</b> ' for the first struct.

<b>OriginPosition</b>	Position offset of the origin of the child frame relative to the parent frame, represented as a 3-by-1 vector.
<b>OriginVelocity</b>	Velocity offset of the origin of the child frame relative to the parent frame, represented as a 3-by-1 vector.
<b>Orientation</b>	3-by-3 real-valued orthonormal frame rotation matrix. The direction of the rotation depends on the <b>IsParentToChild</b> field.
<b>IsParentToChild</b>	A logical scalar indicating if <b>Orientation</b> performs a frame rotation from the parent coordinate frame to the child coordinate frame. If <b>false</b> , <b>Orientation</b> performs a frame rotation from the child coordinate frame to the parent coordinate frame.
<b>HasElevation</b>	A logical scalar indicating if elevation is included in the measurement. For measurements reported in a rectangular frame, and if <b>HasElevation</b> is <b>false</b> , the measurements are reported assuming 0 degrees of elevation.
<b>HasAzimuth</b>	A logical scalar indicating if azimuth is included in the measurement.
<b>HasRange</b>	A logical scalar indicating if range is included in the measurement.
<b>HasVelocity</b>	A logical scalar indicating if the reported detections include velocity measurements. For measurements reported in the rectangular frame, if <b>HasVelocity</b> is <b>false</b> , the measurements are reported as [x y z]. If <b>HasVelocity</b> is <b>true</b> , measurements are reported as [x y z vx vy vz].

### Object Attributes

Object attributes contain additional information about a detection:

<b>Attribute</b>	<b>Description</b>
<b>TargetIndex</b>	Identifier of the platform, <b>PlatformID</b> , that generated the detection. For false alarms, this value is negative.
<b>SNR</b>	Detection signal-to-noise ratio in dB.

### 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.

## 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

### Objects

[objectDetection](#) | [radarEmission](#)

### Functions

[targetPoses](#)

### System Objects

[trackerGNN](#) | [trackerTOMHT](#)

**Introduced in R2018b**

# monostaticLidarSensor

Simulate and model lidar point cloud generator

## Description

The `monostaticLidarSensor` System object generates point cloud detections of targets by a monostatic lidar sensor. You can use the `monostaticLidarSensor` object in a scenario containing moving and stationary platforms such as one created using `trackingScenario`. The `monostaticLidarSensor` object generates point clouds from platforms with defined meshes (using the `Mesh` property). The `monostaticLidarSensor` System object models an ideal point cloud generator and does not account for the effects of false alarms and missed detections.

To generate point cloud detections using a simulated lidar sensor:

- 1 Create the `monostaticLidarSensor` 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

```
sensor = monostaticLidarSensor(SensorIndex)
sensor = monostaticLidarSensor(SensorIndex,Name,Value)
```

### Description

`sensor = monostaticLidarSensor(SensorIndex)` creates a simulated lidar sensor with a specified sensor index, `SensorIndex`. Default property values are used.

`sensor = monostaticLidarSensor(SensorIndex,Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes. For example, `monostaticLidarSensor(1,'DetectionCoordinates','Sensor')` creates a simulated lidar sensor that reports detections in the sensor Cartesian coordinate system with sensor index equal to 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](#).

#### **SensorIndex — Unique sensor identifier**

positive integer

Unique sensor identifier, specified as a positive integer. This property distinguishes point clouds generated from different sensors in a multi-sensor system. When creating a `monostaticLidarSensor` system object, you must either specify the `SensorIndex` as the first input argument in the creation syntax, or specify it as the value for the `SensorIndex` property in the creation syntax.

#### **UpdateRate — Sensor update rate**

15 (default) | positive scalar

Sensor update rate, specified as a positive scalar in Hz. The update interval (reciprocal of the `UpdateRate`) must be an integer multiple of the simulation time interval defined in `trackingScenario`. Any update requested to the sensor between update intervals contains no point clouds.

Example: 5

Data Types: double

#### **MountingLocation — Sensor location on platform**

[1.5 0 0] (default) | 1-by-3 real-valued vector

Sensor location on platform, specified as a 1-by-3 real-valued vector. This property defines the coordinates of the sensor with respect to the platform origin. The default value specifies that the sensor origin is 1.5 meters forward of the platform origin. Units are in meters.

Example: [ .2 0.1 0 ]

Data Types: double

#### **MountingAngles — Sensor mounting orientation**

[0 0 0] (default) | 3-element vector of scalar

Sensor mounting orientation on the platform, specified as a 3-element vector of scalars in degrees. Each element of the vector corresponds to an intrinsic Euler angle rotation that carries the body axes of the platform to the sensor axes. The three elements define the rotations around the z-, y-, and x-axes, in that order. The first rotation rotates the platform axes around the z-axis. The second rotation rotates the frame around the rotated y-axis. The final rotation rotates the frame around the carried x-axis.

Example: [10 20 -15]

Data Types: double

#### **MaxRange — Maximum detection range**

120 (default) | positive scalar

Maximum detection range, specified as a positive scalar in meters.

Example: 500

Data Types: double

#### **RangeAccuracy — Accuracy of range measurements**

0.002 (default) | positive scalar

Accuracy of range measurements, specified as a positive scalar in meters. The property value represents the standard deviation of the range measurements.

Example: 0.1

Data Types: double

**AzimuthResolution — Azimuth resolution**

0.16 (default) | positive scalar

Azimuth resolution of the lidar sensor, specified as a positive scalar in degrees. The number of points per elevation channel is equal to the azimuth limits divided by the azimuth resolution.

Data Types: double

**ElevationResolution — Elevation resolution**

1.25 (default) | positive scalar

Elevation resolution of the lidar sensor, specified as a positive scalar in degrees. The number of points per azimuth channel is equal to the elevation limits divided by the elevation resolution.

Data Types: double

**AzimuthLimits — Azimuth limits**

[-180 180] (default) | 1-by-2 row vector of scalar

Azimuth limits of the lidar sensor, specified as a 1-by-2 row vector of scalars in degrees.

Example: [-90 90]

Data Types: double

**ElevationLimits — Elevation limits**

[-20 20] (default) | 1-by-2 row vector of scalar

Elevation limits of the lidar sensor, specified as a 1-by-2 row vector of scalars in degrees.

Example: [-90 90]

Data Types: double

**HasNoise — Enable addition of noise to point cloud locations**

true (default) | false

Enable addition of noise to point cloud locations, specified as true or false. Set this property to true to add noise to point cloud locations. Otherwise, the point cloud locations contain no noise. The sensor adds random Gaussian noise to each point with mean equal to zero and standard deviation specified by the RangeAccuracy property.

Data Types: logical

**HasOrganizedOutput — Enable organized point cloud locations**

false (default) | true

Enable organized point cloud locations, specified as true or false.

- When this property is set as true, the point cloud output is an  $N$ -by- $M$ -by-3 array of scalars, where  $N$  is the number of elevation channels, and  $M$  is the number of azimuth channels.
- When this property is set as false, the point cloud output is an  $P$ -by-3 matrix of scalars, where  $P$  is the product of the numbers of elevation and azimuth channels.

Data Types: logical

**HasINS — Enable inertial navigation system (INS) input**  
`false` (default) | `true`

Enable the optional input argument that passes the current INS estimate of the sensor platform pose to the sensor, specified as `false` or `true`. When `true`, the pose information is added to the `MeasurementParameters` property of the configuration, enabling tracking and fusion algorithms to estimate the state of the targets in the scenario frame. It also enables to report the point cloud locations in the scenario frame.

Data Types: logical

**DetectionCoordinates — Coordinate system of reported detections**  
`'Sensor'` (default) | `'Body'` | `'Scenario'`

Coordinate system in which the detections are reported, specified as:

- `'Sensor'` — Detections are reported in the sensor's rectangular coordinate system.
- `'Body'` — Detections are reported in the rectangular body system of the platform.
- `'Scenario'` — Detections are reported in the rectangular scenario coordinate frame. To enable this value, set the `HasINS` property to `true`.

Data Types: char

## Usage

### Syntax

```
pointCloud = sensor(targetMeshes,time)
pointCloud = sensor(targetMeshes,insPose,time)
[pointCloud,config] = sensor(__)
[pointCloud,config,clusters] = sensor(__)
```

### Description

`pointCloud = sensor(targetMeshes,time)` returns point cloud measurements from the 3-D geometric meshes of targets, `tgtMeshes`, at the simulation time.

`pointCloud = sensor(targetMeshes,insPose,time)` also specifies the INS-estimated pose, `insPose`, for the sensor platform. INS information is used by tracking and fusion algorithms to estimate the target positions in the scenario frame.

To enable this syntax, set the `HasINS` property to `true`.

`[pointCloud,config] = sensor(__)` also returns the configuration of the sensor, `config`, at the current simulation time. You can use these output arguments with any of the previous input syntaxes.

`[pointCloud,config,clusters] = sensor(__)` also returns `clusters`, the true cluster labels for each point in the point cloud.

### Input Arguments

**targetMeshes — Meshes of targets**

array of structure

Meshes of targets, specified as an array of structures. Each structure must contain the following fields.

Field Name	Description
PlatformID	Unique identifier of the target, specified as a nonnegative integer.
ClassID	Unique identifier of the class of the target, specified as a nonnegative integer.
Position	Position of the target with respect to the sensor mounting platform's body frame, specified as a 3-element vector of scalars.
Orientation	Orientation of the target with respect to the sensor mounting platform's body frame, specified as a <code>quaternion</code> object or a rotation matrix.
Mesh	Geometric mesh of the target, specified as an <code>extendedObjectMesh</code> object with respect to the target's body frame.

#### **insPose — Platform pose from INS**

structure

Platform pose from INS estimation, specified as a structure. The INS information can be used by tracking and fusion algorithms to estimate the platform's pose and velocity in the scenario frame.

Platform pose information from an inertial navigation system (INS) is a structure with these fields:

Field	Definition
Position	Position in the navigation frame, specified as a real-valued 1-by-3 vector. Units are in meters.
Velocity	Velocity in the navigation frame, specified as a real-valued 1-by-3 vector. Units are in meters per second.
Orientation	Orientation with respect to the navigation frame, specified as a <code>quaternion</code> 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.

#### **Dependencies**

To enable this argument, set the `HasINS` property to `true`.

Data Types: `struct`

#### **time — Current simulation time**

nonnegative scalar

Current simulation time, specified as a positive scalar in seconds.

Data Types: `double`

## Output Arguments

### **pointCloud — Point cloud detections**

*N*-by-*M*-by-3 array of scalars | *P*-by-3 matrix of scalars

Point cloud detections generated by the sensor, return as an array of scalars. The dimension of the array is determined by the `HasOrganizedOutput` property.

- When the property is set as `true`, `pointClouds` is returned an *N*-by-*M*-by-3 array of scalars, where *N* is the number of elevation channels, and *M* is the number of azimuth channels.
- When the property is set as `false`, `pointClouds` is returned as an *P*-by-3 matrix of scalars, where *P* is the product of the numbers of elevation and azimuth channels.

The coordinate frame in which the point cloud locations are reported is determined by the `DetectionCoordinates` property.

### **config — Current sensor configuration**

structure

Current sensor configuration, returned as a structure. The structure has these fields:

Field	Description
<code>SensorIndex</code>	Unique sensor index, returned as a positive integer.
<code>IsValidTime</code>	<code>IsValidTime</code> is <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-2 matrix of positive real values. The first row elements are the lower and upper azimuth limits; the second row elements are the lower and upper elevation limits.
<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.

Data Types: struct

### **clusters — Cluster labels of points**

*N*-by-*M*-by-2 array of nonnegative integer | *P*-by-2 matrix of nonnegative integer

Cluster labels of points in the `PointCloud` output, returned as an array of nonnegative integers. The dimension of the array is determined by the `HasOrganizedOutput` property.

- When this property is set as `true`, `cluster` is returned as an *N*-by-*M*-by-2 array of scalars, where *N* is the number of elevation channels, and *M* is the number of azimuth channels. On the third

dimension, the first element represents the `PlatformID` of the target generating the point, and the second element represents the `ClassID` of the target.

- When this property is set as `false`, `pointClouds` is returned as a  $P$ -by-2 matrix of scalars, where  $P$  is the product of the numbers of elevation and azimuth channels. For each column of the matrix, the first element represents the `PlatformID` of the target generating the point whereas the second element represents the `ClassID` of the target.

## 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 monostaticLidarSensor

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

### Generate Point Cloud Using monostaticLidarSensor

Create a tracking scenario. Add an ego platform and a target platform.

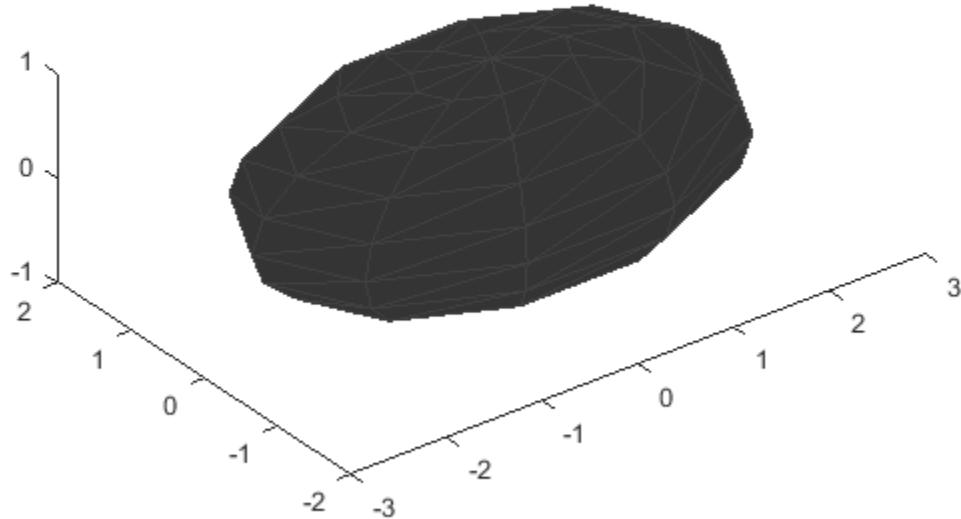
```
scenario = trackingScenario;  
  
ego = platform(scenario);  
target = platform(scenario, 'Trajectory', kinematicTrajectory('Position',[10 -3 0], 'Velocity',[5 0 0]), 'Mesh', extendedObjectMesh('sphere'));
```

Define the geometric mesh of the target. The size of the mesh is adjusted after specifying the target dimensions.

```
target.Mesh = extendedObjectMesh('sphere');  
target.Dimensions.Length = 5;  
target.Dimensions.Width = 3;  
target.Dimensions.Height = 2;
```

Visualize the mesh of the target.

```
show(target.Mesh)
```



```
ans =
Axes with properties:

    XLim: [-3 3]
    YLim: [-2 2]
    XScale: 'linear'
    YScale: 'linear'
    GridLineStyle: '-'
    Position: [0.1300 0.1100 0.7750 0.8150]
    Units: 'normalized'
```

Show all properties

Create a `monostaticLidarSensor` with specified `UpdateRate` and `DetectionCoordinates`.

```
sensor = monostaticLidarSensor(1, 'UpdateRate', 10, 'DetectionCoordinates', 'Body');
```

Obtain the mesh of the target viewed from the ego platform after advancing the scenario one step forward.

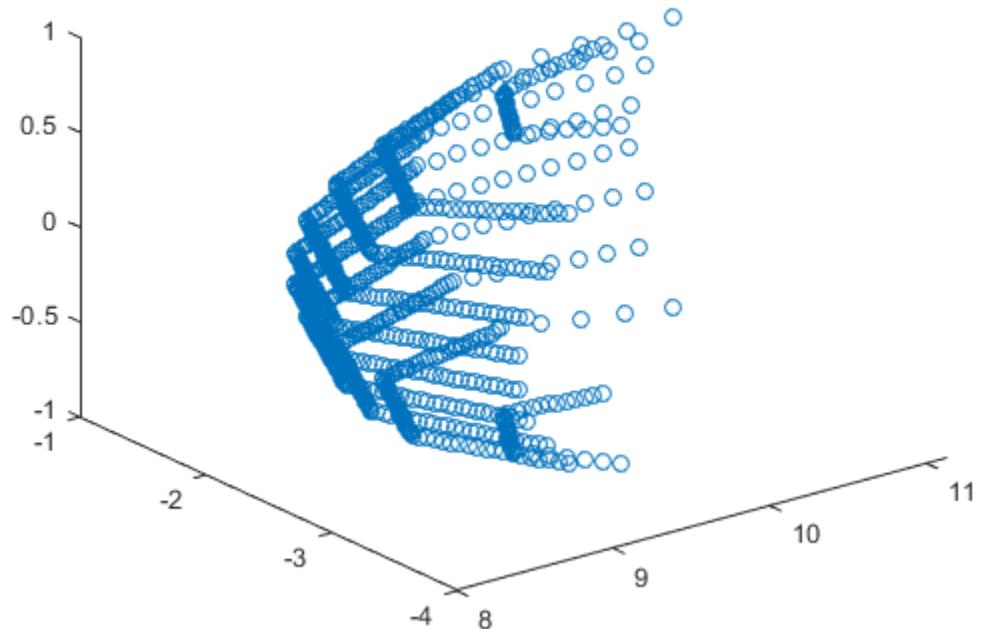
```
advance(scenario);
tgtmeshes = targetMeshes(ego);
```

Use the created sensor to generate point clouds from the obtained target mesh.

```
time = scenario.SimulationTime;
[ptCloud, config, clusters] = sensor(tgtmeshes, time);
```

Visualize the point cloud detections.

```
figure()
plot3(ptCloud(:,1),ptCloud(:,2),ptCloud(:,3), 'o')
```



#### See Also

[extendedObjectMesh](#) | [lidarDetect](#) | [targetMeshes](#)

**Introduced in R2020b**

# trackAssignmentMetrics

Track establishment, maintenance, and deletion metrics

## Description

The `trackAssignmentMetrics` System object compares tracks from a multi-object tracking system against known truth by automatic assignment of tracks to the known truths at each track update. An assignment distance metric determines the maximum distance for which a track can be assigned to the truth object. A divergence distance metric determines when a previously assigned track can be reassigned to a different truth object when the distance exceeds another set threshold.

To generate track assignment metrics:

- 1 Create the `trackAssignmentMetrics` 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

```
assignmentMetrics = trackAssignmentMetrics  
assignmentMetrics = trackAssignmentMetrics(Name,Value)
```

#### Description

`assignmentMetrics = trackAssignmentMetrics` creates a `trackAssignmentMetrics` System object, `assignmentMetrics`, with default property values.

`assignmentMetrics = trackAssignmentMetrics(Name,Value)` sets properties for the `trackAssignmentMetrics` object using one or more name-value pairs. For example, `assignmentMetrics = trackAssignmentMetrics('AssignmentThreshold',5)` creates a `trackAssignmentMetrics` object with an assignment threshold of 5. Enclose property names 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](#).

#### **AssignmentThreshold — Maximum permitted assignment distance**

1 (default) | nonnegative scalar

Maximum permitted assignment distance between a newly encountered or divergent track and a truth object, specified as a nonnegative scalar. For distances beyond this value, assignments between the track and the truth cannot take place. Units are in normalized estimation error squared (NEES).

Data Types: `single` | `double`

**DivergenceThreshold — Maximum permitted divergence distance**

`2` (default) | nonnegative scalar

Maximum permitted divergence distance between a track state and the state of an assigned truth object, specified as a nonnegative scalar. For distances beyond this value, tracks are eligible for reassignment to a different truth object. Units are in NEES.

Data Types: `single` | `double`

**DistanceFunctionFormat — Distance function format**

`'built-in'` (default) | `'custom'`

Distance function format specified as `'built-in'` or `'custom'`.

- `'built-in'` - Enable the `MotionModel`, `AssignmentDistance`, and `DivergenceDistance` properties. These properties are convenient interfaces when tracks are reported by any built-in multi-object tracker (such as `trackerGNN`), and truths reported by the `platformPoses` object function of a `trackingScenario` object.
- `'custom'` - Enable custom properties: `AssignmentDistanceFcn`, `DivergenceDistanceFcn`, `IsInsideCoverageAreaFcn`, `TruthIdentifierFcn`, and `TrackIdentifierFcns`. You can use these properties to construct acceptance or divergence distances, coverage areas, and identifiers for arbitrary `'tracks'` and `'truths'` input arrays.

**MotionModel — Desired platform motion model**

`'constvel'` (default) | `'constacc'` | `'constturn'` | `'singer'`

Desired platform motion model, specified as `'constvel'`, `'constacc'`, `'constturn'`, or `'singer'`. This property selects the motion model used by the `tracks` input.

The motion models expect the `'State'` field of the `tracks` to have a column vector containing these values:

- `'constvel'` — Position is in elements [1 3 5], and velocity is in elements [2 4 6].
- `'constacc'` — Position is in elements [1 4 7], velocity is in elements [2 5 8], and acceleration is in elements [3 6 9].
- `'constturn'` — Position is in elements [1 3 6], velocity is in elements [2 4 7], and yaw rate is in element 5.
- `'singer'` — Position is in elements [1 4 7], velocity is in elements [2 5 8], and acceleration is in elements [3 6 9].

The `'StateCovariance'` field of the `tracks` input must have position, velocity, and turn-rate covariances in the rows and columns corresponding to the position, velocity, and turn-rate of the `'State'` field of the `tracks` input.

**AssignmentDistance — Type of assignment distance**

`'posnees'` (default) | `'velnees'` | `'posabserr'` | `'velabserr'`

Type of assignment distance, specified as `'posnees'`, `'velnees'`, `'posabserr'`, or `'velabserr'`. The type specifies the physical quantity used for assignment. When a new track is detected or a track

becomes divergent, the track is compared against truth using this quantity. The assignment seeks the closest truth within the threshold defined by the `AssignmentThreshold` property.

- '`posnees`' - NEES error of track position
- '`velnees`' - NEES error in track velocity
- '`posabserr`' - Absolute error of track position
- '`velabserr`' - Absolute error of track velocity

#### **Dependencies**

To enable this property, set the `DistanceFunctionFormat` property to '`built-in`'.

#### **DivergenceDistance — Type of assignment distance**

`'posnees'` (default) | `'velnees'` | `'posabserr'` | `'velabserr'`

Type of divergence distance, specified as '`posnees`', '`velnees`', '`posabserr`', or '`velabserr`'. The type specifies the physical quantity used for assessing divergence. When a track was previously assigned to truth, the distance between them is compared to this quantity on subsequent update steps. Any track whose divergence distance to its truth assignment exceeds the value of `DivergenceThreshold` is considered divergent and can be reassigned to a new truth.

- '`posnees`' - NEES error of track position
- '`velnees`' - NEES error in track velocity
- '`posabserr`' - Absolute error of track position
- '`velabserr`' - Absolute error of track velocity

#### **Dependencies**

To enable this property, set the `DistanceFunctionFormat` property to '`built-in`'.

#### **AssignmentDistanceFcn — Assignment distance function**

function handle

Assignment distance function, specified as a function handle. This function determines the assignment distance between truths and tracks. Whenever a new track is detected or an existing track becomes divergent, the track needs to be compared against all truths at the current step. This function help to find the closest truth relative to the track within the threshold defined by the `AssignmentThreshold` property.

The function must have the following syntax:

```
dist = assignmentdistance(onetrack, onetruth)
```

The function must return a nonnegative assignment distance, `dist`, typically expressed in units of NEES. `onetrack` is an element of the `tracks` array input argument. `onetruth` is an element of the `truths` array input argument.

#### **Dependencies**

To enable this property, set the `DistanceFunctionFormat` property to '`custom`'.

Data Types: `function_handle`

#### **DivergenceDistanceFcn — Divergence distance function**

function handle

Divergence distance function, specified as a function handle. This function determines the divergence distance between truths and tracks. If the divergence distance from a track to its truth assignment exceeds the *DivergenceThreshold*, the track is considered divergent and can be reassigned to a new truth.

The function must have the following syntax:

```
dist = divergencedistance(onetrack, onetruth)
```

The function must return a non-negative divergence distance, *dist*, typically expressed in units of NEES. *onetrack* is an element of the *tracks* array input argument. *onetruth* is an element of the *truths* array input argument.

#### **Dependencies**

To enable this property, set the *DistanceFunctionFormat* property to 'custom'.

Data Types: *function\_handle*

#### **IsInsideCoverageAreaFcn — Determine the time that a truth object is detectable**

function handle

Function to determine the time that a truth object is detectable, specified as a function handle. This function determines the time that a truth object is inside the coverage area of the sensors and is therefore detectable.

The function must have the following syntax:

```
status = isinsidecoveragearea(truths)
```

and return a logical array, *status*. *truths* is an array of truth objects expected to be passed in on each step. *status* is a logical array with the same size as the *truths* input. An entry of *status* is true when the corresponding truth object specified by *truths* is within the coverage area of the sensors.

#### **Dependencies**

To enable this property, set the *DistanceFunctionFormat* property to 'custom'.

Data Types: *function\_handle*

#### **TrackIdentifierFcn — Track identifier function**

function handle

Track identifier function for the *tracks* input, specified as a function handle. The track identifiers are unique strings or numeric values.

The function must have the following syntax

```
trackids = trackidentifier(tracks)
```

and return a numeric array, *trackids*. *trackids* must have the same size as *tracks* input argument. The default track identification function assumes *Tracks* is an array of struct or class with a *TrackID* field or property.

#### **Dependencies**

To enable this property, set the *DistanceFunctionFormat* property to 'custom'.

Data Types: function\_handle

### **TruthIdentifierFcn — Truth identifier function**

function handle

Truth identifier function for the `truths` input, specified as a function handle. The truth identifiers are unique strings or numeric values.

The function must have the following syntax

```
truthids = truthidentifier(truths)
```

and return a numeric array, `truthids`. `truthids` must have the same size as the `truths` input argument. The default truth identification function assumes `truths` is an array of struct or class with a `PlatformID` field or property.

### **Dependencies**

To enable this property, set the `DistanceFunctionFormat` property to 'custom'.

Data Types: function\_handle

### **InvalidTrackIdentifier — Track identifier for invalid assignment**

NaN (default) | scalar | string

Track identifier for invalid assignment, specified as a scalar or string. This value is returned when the track assignment is invalid. The value must be of the same class as returned by the function handle specified in `TrackIdentifierFcn`.

Example: -1

Data Types: single | double | string

### **InvalidTruthIdentifier — Truth identifier for invalid assignment**

NaN (default) | scalar | string

Truth identifier for invalid assignment, specified as a scalar or string. This value is returned when the truth assignment is invalid. The value must be of the same class as returned by the function handle specified in `TruthIdentifierFcn`.

Example: -1

Data Types: single | double | string

## **Usage**

To compute metrics, call the track assignment metrics with arguments, as if it were a function (described here).

## **Syntax**

```
[tracksummary,truthsummary] = assignmentMetrics(tracks,truths)
```

## Description

[tracksummary, truthsummary] = assignmentMetrics(tracks, truths) returns structures, `tracksummary` and `truthsummary`, containing cumulative metrics across all `tracks` and `truths`, obtained from the previous object update.

## Input Arguments

### **tracks — Track information**

array of objects | array of structures

Track information, specified as an array of objects or an array of structures. If the `DistanceFunctionFormat` property is specified as 'built-in', then `tracks` must contain `State`, `StateCovariance`, and `TrackID` as property names or field names. The track outputs from built-in trackers, such as `trackerGNN`, are compatible with the `tracks` input.

Data Types: struct

### **truths — Truth information**

structure | array of structures

Truth information, specified as a structure or array of structures. When using a `trackingScenario`, truth information can be obtained from the `platformPoses` object function.

Data Types: struct

## Output Arguments

### **tracksummary — Cumulative track assignment metrics**

structure

Cumulative metrics over all tracks, returned as a structure. The metrics are computed over all tracks since the last call to the `reset` object function. The structure has these fields:

Field	Description
TotalNumTracks	The total number of unique track identifiers encountered
NumFalseTracks	The number of tracks never assigned to any truth
MaxSwapCount	Maximum number of track swaps of each track. A track swap occurs whenever a track is assigned to a different truth.
TotalSwapCount	Total number of track swaps of each track. A track swap occurs whenever a track is assigned to a different truth.
MaxDivergenceCount	Maximum number of divergences. A track is divergent when the result of the <code>DivergenceDistanceFcn</code> is greater than the divergence threshold.
TotalDivergenceCount	Total number of divergences. A track is divergent when the result of the divergence distance function is greater than the divergence threshold.

MaxDivergenceLength	Maximum number of updates during which each track was in a divergent state
TotalDivergenceLength	Total number of updates during which each track was in a divergent state
MaxRedundancyCount	The maximum number of additional tracks assigned to the same truth
TotalRedundancyCount	The total number of additional tracks assigned to the same truth
MaxRedundancyLength	Maximum number of updates during which each track was in a redundant state
TotalRedundancyLength	Total number of updates during which each track was in a redundant state

Data Types: struct

### **truthsummary — Cumulative truth assignment metrics**

structure

Cumulative assignment metrics over all truths, returned as a structure. The metrics are computed over all truths since the last call to the `reset` object function. The structure has these fields:

Field	Description
TotalNumTruths	The total number of unique truth identifiers encountered
NumMissingTruths	The number of truths never established with any track
MaxEstablishmentLength	Maximum number of updates before a truth was associated with any track while inside the coverage area. The lengths of missing truths do not count toward this summary metric.
TotalEstablishmentLength	Total number of updates before a truth was associated with any track while inside the coverage area. The lengths of missing truths do not count toward this summary metric.
MaxBreakCount	Maximum number of times each truth was unassociated by any track after being established.
TotalBreakCount	Total number of times each truth was unassociated by any track after being established.
MaxBreakLength	Maximum number of updates during which each truth was in a broken state
TotalBreakLength	Total number of updates during which each truth was in a broken state

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 trackAssignmentMetrics

currentAssignment	Mapping of tracks to truth
trackMetricsTable	Compare tracks to truth
truthMetricsTable	Compare truth to tracks

### Common to All System Objects

release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isLocked	Determine if System object is in use
clone	Create duplicate System object

## Examples

### Assignment and Error Metrics for Two Tracked Targets

Examine the assignments and errors for a system tracking two targets.

First, load the stored track data.

```
load trackmetricex tracklog truthlog
```

Create objects to analyze assignment and error metrics.

```
tam = trackAssignmentMetrics;  
tem = trackErrorMetrics;
```

Create the output variables.

```
posRMSE = zeros(numel(tracklog),1);  
velRMSE = zeros(numel(tracklog),1);  
posANEEs = zeros(numel(tracklog),1);  
velANEEs = zeros(numel(tracklog),1);
```

Loop over all tracks to:

- Extract the tracks and ground truth at the  $i$  th tracker update.
- Analyze and retrieve the current track-to-truth assignment.
- Analyze instantaneous error metrics over all tracks and truths.

```
for i=1:numel(tracklog)  
    tracks = tracklog{i};  
    truths = truthlog{i};  
    [trackAM,truthAM] = tam(tracks, truths);  
    [trackIDs,truthIDs] = currentAssignment(tam);
```

```
[posRMSE(i),velRMSE(i),posANEES(i),velANEES(i)] = ...
    tem(tracks,trackIDs,truths,truthIDs);
end
```

Show the track metrics table.

```
trackMetricsTable(tam)
```

TrackID	AssignedTruthID	Surviving	TotalLength	DeletionStatus	DeletionLength
1	NaN	false	1120	false	0
2	NaN	false	1736	false	0
6	3	true	1138	false	0
8	2	true	662	false	0

Show the truth metrics table.

```
truthMetricsTable(tam)
```

TruthID	AssociatedTrackID	DeletionStatus	TotalLength	BreakStatus	BreakCount
2	8	false	2678	false	4
3	6	false	2678	false	3

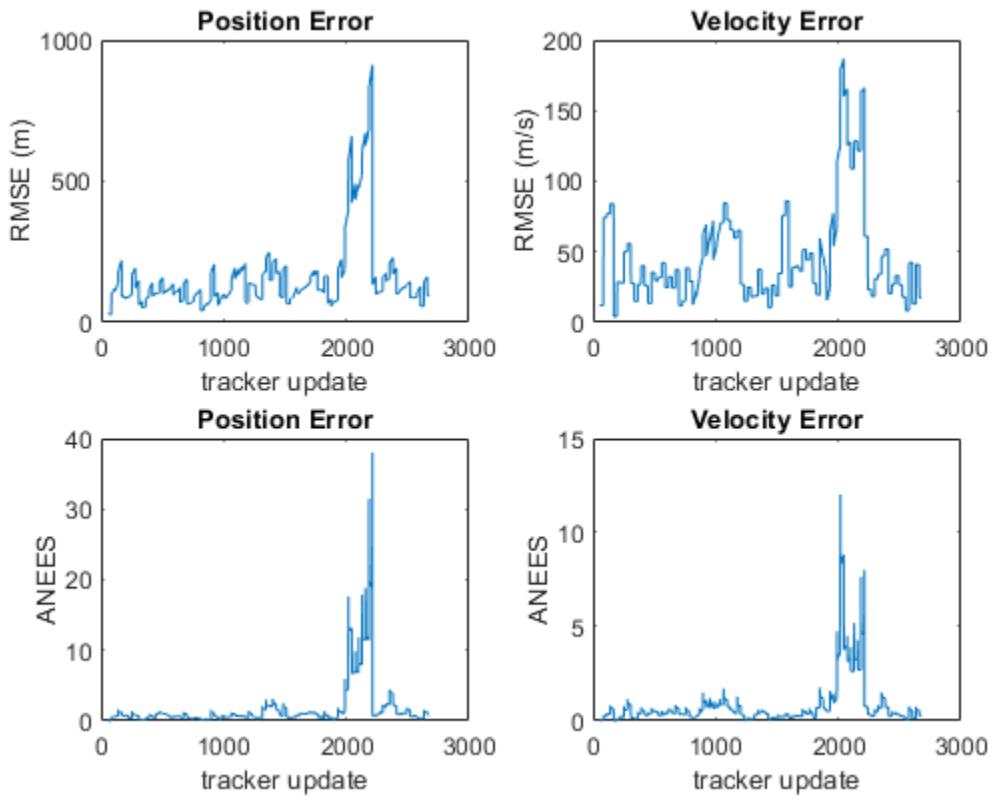
Plot the RMSE and ANEES error metrics.

```
subplot(2,2,1)
plot(posRMSE)
title('Position Error')
xlabel('tracker update')
ylabel('RMSE (m)')

subplot(2,2,2)
plot(velRMSE)
title('Velocity Error')
xlabel('tracker update')
ylabel('RMSE (m/s)')

subplot(2,2,3)
plot(posANEES)
title('Position Error')
xlabel('tracker update')
ylabel('ANEES')

subplot(2,2,4)
plot(velANEES)
title('Velocity Error')
xlabel('tracker update')
ylabel('ANEES')
```



Show the current error metrics for each individual recorded track.

```
currentTrackMetrics(tem)
```

```
ans=2×5 table
TrackID    posRMS    velRMS    posANEES    velANEES
_____
6          44.712    20.988    0.05974    0.31325
8          129.26    12.739    1.6745     0.2453
```

Show the current error metrics for each individual recorded truth object.

```
currentTruthMetrics(tem)
```

```
ans=2×5 table
TruthID    posRMS    velRMS    posANEES    velANEES
_____
2          129.26    12.739    1.6745     0.2453
3          44.712    20.988    0.05974    0.31325
```

Show the cumulative error metrics for each individual recorded track.

```
cumulativeTrackMetrics(tem)
```

---

ans=4×5 table

TrackID	posRMS	velRMS	posANEES	velANEES
1	117.69	43.951	0.58338	0.44127
2	129.7	42.8	0.81094	0.42509
6	371.35	87.083	4.5208	1.6952
8	130.45	53.914	1.0448	0.44813

Show the cumulative error metrics for each individual recorded truth object.

cumulativeTruthMetrics(tem)

ans=2×5 table

TruthID	posRMS	velRMS	posANEES	velANEES
2	258.21	65.078	2.2514	0.93359
3	134.41	48.253	0.96314	0.49183

## See Also

### System Objects

[monostaticRadarSensor](#) | [trackErrorMetrics](#) | [trackOSPAMetric](#) | [trackerGNN](#) | [trackerTOMHT](#)

**Introduced in R2018b**

# currentAssignment

Mapping of tracks to truth

## Syntax

```
[trackIDs,truthIDs] = currentAssignment(assignmentMetric)
```

## Description

[trackIDs,truthIDs] = currentAssignment(assignmentMetric) returns the assignment of tracks to truth after the most recent update of the `assignmentMetric` System object. The assignment is returned as a vector of track identifiers, `trackIDs`, and truth identifiers, `truthIDs`. Corresponding elements of the `trackIDs` and `truthIDs` vectors define the assignments.

## Examples

### Assignment and Error Metrics for Two Tracked Targets

Examine the assignments and errors for a system tracking two targets.

First, load the stored track data.

```
load trackmetricex tracklog truthlog
```

Create objects to analyze assignment and error metrics.

```
tam = trackAssignmentMetrics;  
tem = trackErrorMetrics;
```

Create the output variables.

```
posRMSE = zeros(numel(tracklog),1);  
velRMSE = zeros(numel(tracklog),1);  
posANEES = zeros(numel(tracklog),1);  
velANEES = zeros(numel(tracklog),1);
```

Loop over all tracks to:

- Extract the tracks and ground truth at the  $i$  th tracker update.
- Analyze and retrieve the current track-to-truth assignment.
- Analyze instantaneous error metrics over all tracks and truths.

```
for i=1:numel(tracklog)  
    tracks = tracklog{i};  
    truths = truthlog{i};  
    [trackAM,truthAM] = tam(tracks, truths);  
    [trackIDs,truthIDs] = currentAssignment(tam);  
    [posRMSE(i),velRMSE(i),posANEES(i),velANEES(i)] = ...  
        tem(tracks,trackIDs,truths,truthIDs);  
end
```

Show the track metrics table.

```
trackMetricsTable(tam)
```

TrackID	AssignedTruthID	Surviving	TotalLength	DeletionStatus	DeletionLength
1	NaN	false	1120	false	0
2	NaN	false	1736	false	0
6	3	true	1138	false	0
8	2	true	662	false	0

Show the truth metrics table.

```
truthMetricsTable(tam)
```

TruthID	AssociatedTrackID	DeletionStatus	TotalLength	BreakStatus	BreakCount
2	8	false	2678	false	4
3	6	false	2678	false	3

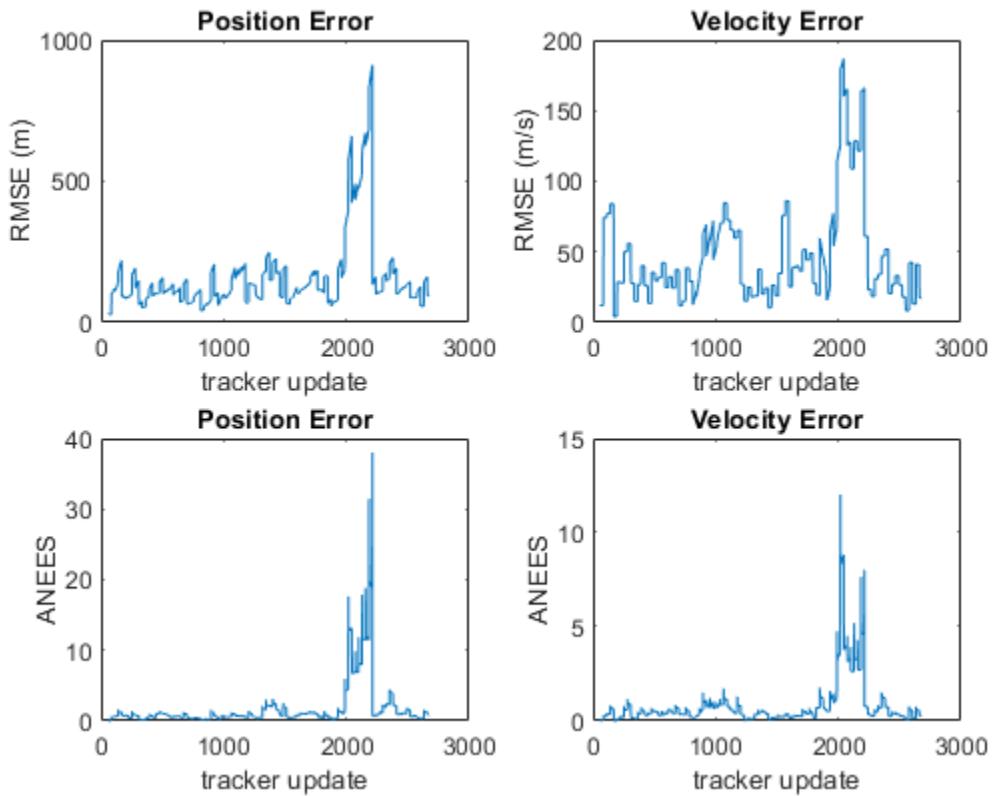
Plot the RMSE and ANEES error metrics.

```
subplot(2,2,1)
plot(posRMSE)
title('Position Error')
xlabel('tracker update')
ylabel('RMSE (m)')

subplot(2,2,2)
plot(velRMSE)
title('Velocity Error')
xlabel('tracker update')
ylabel('RMSE (m/s)')

subplot(2,2,3)
plot(posANEES)
title('Position Error')
xlabel('tracker update')
ylabel('ANEES')

subplot(2,2,4)
plot(velANEES)
title('Velocity Error')
xlabel('tracker update')
ylabel('ANEES')
```



Show the current error metrics for each individual recorded track.

```
currentTrackMetrics(tem)
```

```
ans=2×5 table
  TrackID    posRMS    velRMS    posANEES    velANEES
  _____    _____    _____    _____    _____
  6          44.712   20.988   0.05974   0.31325
  8         129.26    12.739   1.6745    0.2453
```

Show the current error metrics for each individual recorded truth object.

```
currentTruthMetrics(tem)
```

```
ans=2×5 table
  TruthID    posRMS    velRMS    posANEES    velANEES
  _____    _____    _____    _____    _____
  2          129.26   12.739   1.6745    0.2453
  3         44.712   20.988   0.05974   0.31325
```

Show the cumulative error metrics for each individual recorded track.

```
cumulativeTrackMetrics(tem)
```

ans=4x5 table	TrackID	posRMS	velRMS	posANEES	velANEES
	1	117.69	43.951	0.58338	0.44127
	2	129.7	42.8	0.81094	0.42509
	6	371.35	87.083	4.5208	1.6952
	8	130.45	53.914	1.0448	0.44813

Show the cumulative error metrics for each individual recorded truth object.

```
cumulativeTruthMetrics(tem)
```

ans=2x5 table	TruthID	posRMS	velRMS	posANEES	velANEES
	2	258.21	65.078	2.2514	0.93359
	3	134.41	48.253	0.96314	0.49183

## Input Arguments

**assignmentMetric — Track assignment metrics object**

trackAssignmentMetrics System object

Track assignment metrics object, specified as a trackAssignmentMetrics System object.

## Output Arguments

**trackIDs — Track identifiers**

vector

Track identifiers, returned as a vector. trackIDs and truthIDs have the same size. Corresponding elements of trackIDs and truthIDs represent a track-truth assignment.

**truthIDs — Truth identifiers**

vector

Truth identifiers, returned as a vector. trackIDs and truthIDs have the same size. Corresponding elements of trackIDs and truthIDs represent a track-truth assignment.

## Introduced in R2018b

# trackMetricsTable

Compare tracks to truth

## Syntax

```
metricsTable = trackMetricsTable(assignmentMetric)
```

## Description

`metricsTable = trackMetricsTable(assignmentMetric)` returns a table of metrics, `metricsTable`, for all tracks in the track assignment metrics object, `assignmentMetric`.

## Examples

### Assignment and Error Metrics for Two Tracked Targets

Examine the assignments and errors for a system tracking two targets.

First, load the stored track data.

```
load trackmetricex tracklog truthlog
```

Create objects to analyze assignment and error metrics.

```
tam = trackAssignmentMetrics;
tem = trackErrorMetrics;
```

Create the output variables.

```
posRMSE = zeros(numel(tracklog),1);
velRMSE = zeros(numel(tracklog),1);
posANEES = zeros(numel(tracklog),1);
velANEES = zeros(numel(tracklog),1);
```

Loop over all tracks to:

- Extract the tracks and ground truth at the  $i$  th tracker update.
- Analyze and retrieve the current track-to-truth assignment.
- Analyze instantaneous error metrics over all tracks and truths.

```
for i=1:numel(tracklog)
    tracks = tracklog{i};
    truths = truthlog{i};
    [trackAM,truthAM] = tam(tracks, truths);
    [trackIDs,truthIDs] = currentAssignment(tam);
    [posRMSE(i),velRMSE(i),posANEES(i),velANEES(i)] = ...
        tem(tracks,trackIDs,truths,truthIDs);
end
```

Show the track metrics table.

```
trackMetricsTable(tam)
```

```
ans=4×15 table
```

TrackID	AssignedTruthID	Surviving	TotalLength	DeletionStatus	DeletionLength
1	NaN	false	1120	false	0
2	NaN	false	1736	false	0
6	3	true	1138	false	0
8	2	true	662	false	0

Show the truth metrics table.

```
truthMetricsTable(tam)
```

```
ans=2×10 table
```

TruthID	AssociatedTrackID	DeletionStatus	TotalLength	BreakStatus	BreakCount
2	8	false	2678	false	4
3	6	false	2678	false	3

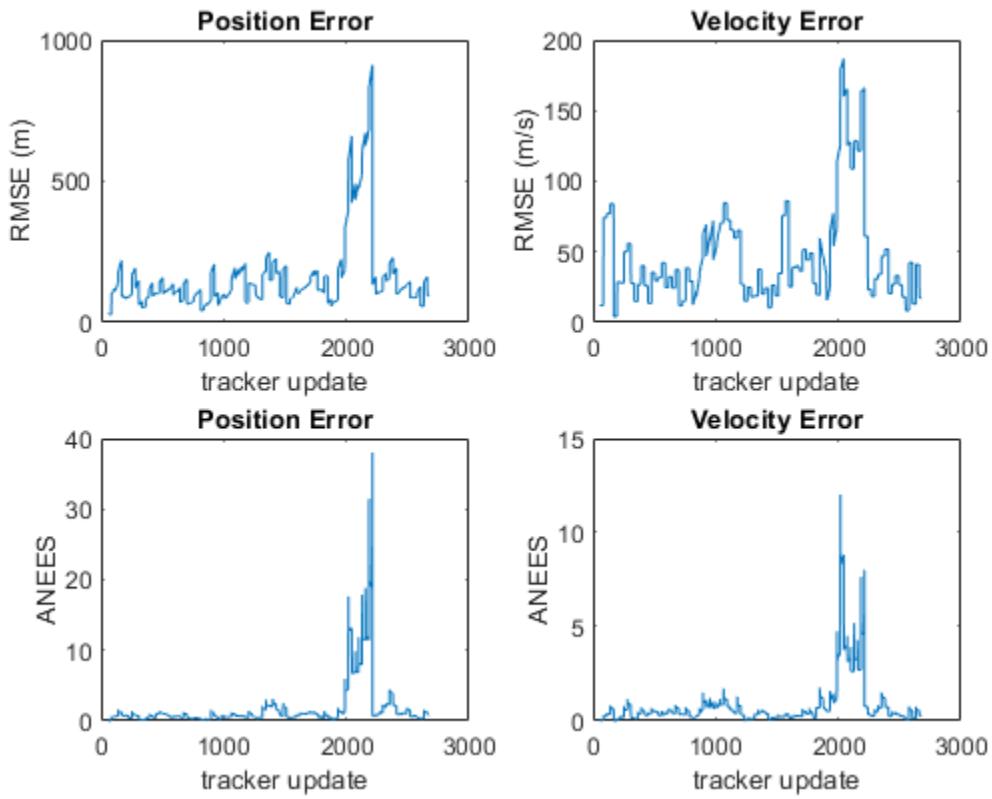
Plot the RMSE and ANEES error metrics.

```
subplot(2,2,1)
plot(posRMSE)
title('Position Error')
xlabel('tracker update')
ylabel('RMSE (m)')

subplot(2,2,2)
plot(velRMSE)
title('Velocity Error')
xlabel('tracker update')
ylabel('RMSE (m/s)')

subplot(2,2,3)
plot(posANEES)
title('Position Error')
xlabel('tracker update')
ylabel('ANEES')

subplot(2,2,4)
plot(velANEES)
title('Velocity Error')
xlabel('tracker update')
ylabel('ANEES')
```



Show the current error metrics for each individual recorded track.

```
currentTrackMetrics(tem)
```

```
ans=2×5 table
TrackID    posRMS    velRMS    posANEES    velANEES
_____
6          44.712    20.988    0.05974    0.31325
8          129.26    12.739    1.6745     0.2453
```

Show the current error metrics for each individual recorded truth object.

```
currentTruthMetrics(tem)
```

```
ans=2×5 table
TruthID    posRMS    velRMS    posANEES    velANEES
_____
2          129.26    12.739    1.6745     0.2453
3          44.712    20.988    0.05974    0.31325
```

Show the cumulative error metrics for each individual recorded track.

```
cumulativeTrackMetrics(tem)
```

```
ans=4×5 table
TrackID    posRMS    velRMS    posANEES    velANEES
_____
1          117.69    43.951    0.58338    0.44127
2          129.7     42.8      0.81094    0.42509
6          371.35    87.083    4.5208     1.6952
8          130.45    53.914    1.0448     0.44813
```

Show the cumulative error metrics for each individual recorded truth object.

```
cumulativeTruthMetrics(tem)
```

```
ans=2×5 table
TruthID    posRMS    velRMS    posANEES    velANEES
_____
2          258.21    65.078    2.2514     0.93359
3          134.41    48.253    0.96314    0.49183
```

## Input Arguments

**assignmentMetric — Track assignment metrics object**

trackAssignmentMetrics System object

Track assignment metrics object, specified as a trackAssignmentMetrics System object.

## Output Arguments

**metricsTable — Track metrics table**

table

Track metrics table, returned as a table. Each row of the table represents a track. The table has these columns:

Column	Description
TrackID	Unique track identifier
AssignedTruthID	Unique truth identifier. If the track is not assigned to any truth, or the track was not reported in the last update, then the value of AssignedTruthID is NaN.
Surviving	True if the track was reported in the last update
TotalLength	Number of updates in which this track was reported
DeletionStatus	True if the track was previously assigned to a truth that was deleted while inside its coverage area.

DeletionLength	The number of updates in which the track was following a deleted truth
DivergenceStatus	True when the divergence distance between this track and its corresponding truth exceeds the divergence threshold
DivergenceCount	Number of times this track entered a divergent state
DivergenceLength	Number of updates in which this track was in a divergent state
RedundancyStatus	True if this track is assigned to a truth already associated with another track
RedundancyCount	Number of times this track entered a redundant state
RedundancyLength	Number of updates for which this track was in a redundant state
FalseTrackStatus	True if the track was not assigned to any truth
FalseTrackLength	Number of updates in which the track was unassigned
SwapCount	Number of times the track was assigned to a new truth object

**Introduced in R2018b**

# truthMetricsTable

Compare truth to tracks

## Syntax

```
metricsTable = truthMetricsTable(assignmentMetric)
```

## Description

`metricsTable = truthMetricsTable(assignmentMetric)` returns a table of metrics, `metricsTable`, for all truths in the `assignmentMetric` System object.

## Examples

### Assignment and Error Metrics for Two Tracked Targets

Examine the assignments and errors for a system tracking two targets.

First, load the stored track data.

```
load trackmetricex tracklog truthtable
```

Create objects to analyze assignment and error metrics.

```
tam = trackAssignmentMetrics;
tem = trackErrorMetrics;
```

Create the output variables.

```
posRMSE = zeros(numel(tracklog),1);
velRMSE = zeros(numel(tracklog),1);
posANEES = zeros(numel(tracklog),1);
velANEES = zeros(numel(tracklog),1);
```

Loop over all tracks to:

- Extract the tracks and ground truth at the  $i$  th tracker update.
- Analyze and retrieve the current track-to-truth assignment.
- Analyze instantaneous error metrics over all tracks and truths.

```
for i=1:numel(tracklog)
    tracks = tracklog{i};
    truths = truthtable{i};
    [trackAM,truthAM] = tam(tracks, truths);
    [trackIDs,truthIDs] = currentAssignment(tam);
    [posRMSE(i),velRMSE(i),posANEES(i),velANEES(i)] = ...
        tem(tracks,trackIDs,truths,truthIDs);
end
```

Show the track metrics table.

```
trackMetricsTable(tam)
```

```
ans=4×15 table
```

TrackID	AssignedTruthID	Surviving	TotalLength	DeletionStatus	DeletionLength
1	NaN	false	1120	false	0
2	NaN	false	1736	false	0
6	3	true	1138	false	0
8	2	true	662	false	0

Show the truth metrics table.

```
truthMetricsTable(tam)
```

```
ans=2×10 table
```

TruthID	AssociatedTrackID	DeletionStatus	TotalLength	BreakStatus	BreakCount
2	8	false	2678	false	4
3	6	false	2678	false	3

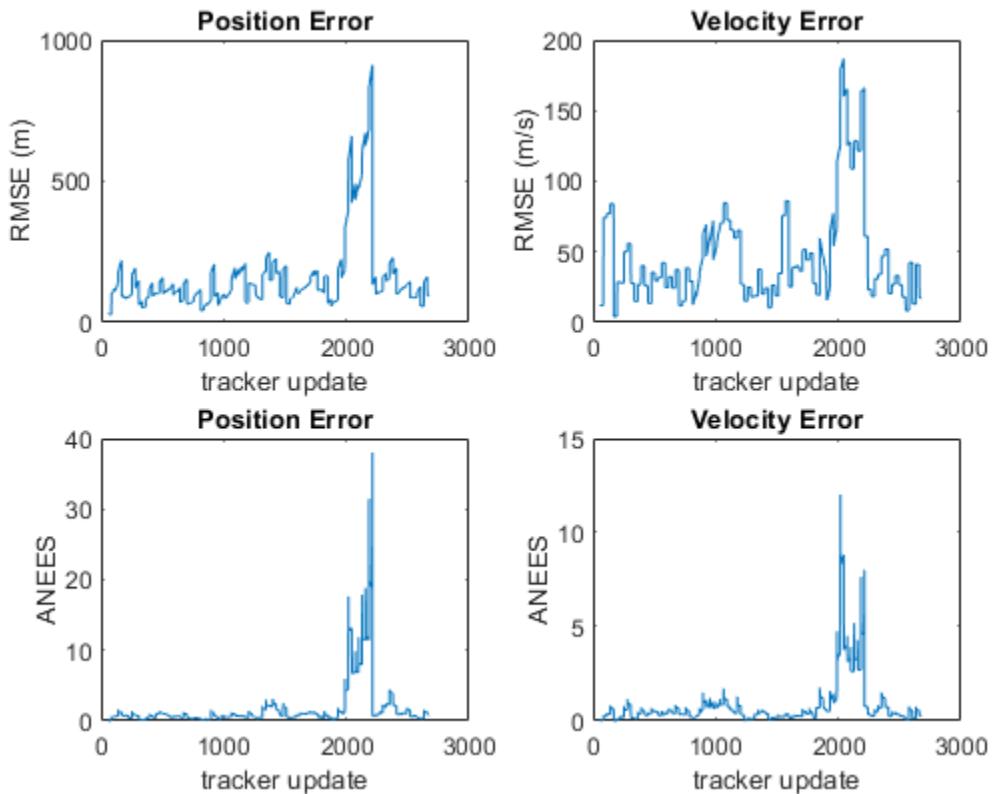
Plot the RMSE and ANEES error metrics.

```
subplot(2,2,1)
plot(posRMSE)
title('Position Error')
xlabel('tracker update')
ylabel('RMSE (m)')

subplot(2,2,2)
plot(velRMSE)
title('Velocity Error')
xlabel('tracker update')
ylabel('RMSE (m/s)')

subplot(2,2,3)
plot(posANEES)
title('Position Error')
xlabel('tracker update')
ylabel('ANEES')

subplot(2,2,4)
plot(velANEES)
title('Velocity Error')
xlabel('tracker update')
ylabel('ANEES')
```



Show the current error metrics for each individual recorded track.

```
currentTrackMetrics(tem)
```

```
ans=2×5 table
TrackID    posRMS    velRMS    posANEES    velANEES
_____
6          44.712    20.988    0.05974    0.31325
8          129.26    12.739    1.6745     0.2453
```

Show the current error metrics for each individual recorded truth object.

```
currentTruthMetrics(tem)
```

```
ans=2×5 table
TruthID    posRMS    velRMS    posANEES    velANEES
_____
2          129.26    12.739    1.6745     0.2453
3          44.712    20.988    0.05974    0.31325
```

Show the cumulative error metrics for each individual recorded track.

```
cumulativeTrackMetrics(tem)
```

ans=4×5 table

TrackID	posRMS	velRMS	posANEES	velANEES
1	117.69	43.951	0.58338	0.44127
2	129.7	42.8	0.81094	0.42509
6	371.35	87.083	4.5208	1.6952
8	130.45	53.914	1.0448	0.44813

Show the cumulative error metrics for each individual recorded truth object.

```
cumulativeTruthMetrics(tem)
```

ans=2×5 table

TruthID	posRMS	velRMS	posANEES	velANEES
2	258.21	65.078	2.2514	0.93359
3	134.41	48.253	0.96314	0.49183

## Input Arguments

**assignmentMetric** — Track assignment metrics object

trackAssignmentMetrics System object

Track assignment metrics object, specified as a trackAssignmentMetrics System object.

## Output Arguments

**metricsTable** — Truth metrics table

table

Truth metrics table, returned as a table. Each row of the table represents a truth. The table has these columns:

TruthID	Unique truth identifier
AssignedTrackID	Unique identifier of the associated track
DeletionStatus	False if the truth was reported in the last update
TotalLength	Number of updates this truth was reported
DeletionLength	The number of updates in which the track was following a deleted truth
BreakStatus	True when an established truth no longer has any track assigned with it
BreakCount	Number of times this truth entered a broken state
BreakLength	Number of updates in which this truth was in a broken state

InCoverageArea	True if this truth object is inside the coverage area
EstablishmentStatus	True if the truth is associated to any track
EstablishmentLength	Number of updates before this truth was associated to any track while inside the coverage area

**Introduced in R2018b**

# trackErrorMetrics

Track error and NEES

## Description

The `trackErrorMetrics` System object provides quantitative comparisons between tracks and known truth trajectories.

To generate track assignment metrics:

- 1 Create the `trackErrorMetrics` 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

```
errorMetrics = trackErrorMetrics  
errorMetrics = trackErrorMetrics(Name,Value)
```

#### Description

`errorMetrics = trackErrorMetrics` creates a `trackErrorMetrics` System object with default property values.

`errorMetrics = trackErrorMetrics(Name,Value)` sets properties for the `trackErrorMetrics` object using one or more name-value pairs. For example, `metrics = trackErrorMetrics('MotionModel','constvel')` creates a `trackErrorMetrics` object with a constant velocity motion model. Enclose property names 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](#).

#### ErrorFunctionFormat — Error function format

'built-in' (default) | 'custom'

Error function format specified as 'built-in' or 'custom'.

- 'built-in' - Enable the `MotionModel` property.

This property is a convenient interface when using tracks reported by any built-in multi-object tracker, and truths reported by the `platformPoses` object function of a `trackingScenario` object. The default estimation error function assumes `tracks` and `truths` are arrays of structures or arrays of objects.

- 'custom' - Enable custom properties: `EstimationErrorLabels`, `EstimationErrorFcn`, `TruthIdentifierFcn`, and `TrackIdentifierFcns`. These properties can be used to construct error functions for arbitrary tracks and truths input arrays.

#### **MotionModel — Desired platform motion model**

'constvel' (default) | 'constacc' | 'constturn' | 'singer'

Desired platform motion model, specified as 'constvel', 'constacc', 'constturn', or 'singer'. This property selects the motion model used by the `tracks` input.

The motion models expect the 'State' field of the `tracks` to have a column vector containing these values:

- 'constvel' — Position is in elements [1 3 5], and velocity is in elements [2 4 6].
- 'constacc' — Position is in elements [1 4 7], velocity is in elements [2 5 8], and acceleration is in elements [3 6 9].
- 'constturn' — Position is in elements [1 3 6], velocity is in elements [2 4 7], and yaw rate is in element 5.
- 'singer' — Position is in elements [1 4 7], velocity is in elements [2 5 8], and acceleration is in elements [3 6 9].

The 'StateCovariance' field of the `tracks` input must have position, velocity, and turn-rate covariances in the rows and columns corresponding to the position, velocity, and turn-rate of the 'State' field of the `tracks` input.

#### **EstimationErrorLabels — Labels for outputs of error estimation function**

'posMSE' (default) | array of strings | cell array of character vectors

Labels for outputs of error estimation function, specified as an array of strings or cell array of character vectors. The number of labels must correspond to the number of outputs of the error estimation function. Specify the error estimation functions using the `EstimationErrorFcn` property.

Example: `{'posMSE', 'velMSE'}`

#### **Dependencies**

To enable this property, set the `ErrorFunctionFormat` property to 'custom'.

Data Types: `char` | `string`

#### **EstimationErrorFcn — Error estimation function**

function handle

Error estimation function, specified as a function handle. The function determines estimation errors of truths to tracks.

The error estimation function can have multiple scalar outputs and must have the following syntax.

```
[out1,out2, ...,outN] = estimationerror(onetrack,onetruth)
```

The number of outputs must match the number of entries in the labels array specified in the **EstimationErrorLabels** property.

**onetrack** is an element of the **tracks** array passed in as input **trackErrorMetric** at object updates. **onetruth** is an element of the **truths** array passed in at object updates. The **trackErrorMetrics** object averages each output arithmetically when reporting across tracks or truths.

Example: `@errorFunction`

#### Dependencies

To enable this property, set the **ErrorFunctionFormat** property to 'custom'.

Data Types: `function_handle`

#### **TrackIdentifierFcn – Track identifier function**

`@trackIDFunction` (default) | `function handle`

Track identifier function, specified as a function handle. Specifies the track identifiers for the **tracks** input at object update. The track identifiers are unique string or numeric values.

The track identifier function must have the following syntax:

```
trackID = trackIDIdentifier(tracks)
```

**tracks** is the same as the **tracks** array passed as input for **trackErrorMetric** at object update. **trackID** is the same size as **tracks**. The default identification function handle, `@defaultTrackIdentifier`, assumes **tracks** is an array of structures or objects with a 'TrackID' field name or property.

#### Dependencies

To enable this property, set the **ErrorFunctionFormat** property to 'custom'.

Data Types: `function_handle`

#### **TruthIdentifierFcn – Truth identifier function**

`@truthIDFunction` (default) | `function handle`

Truth identifier function, specified as a function handle. Specifies the truth identifiers for the **truths** input at object update. The truth identifiers are unique string or numeric values.

The truth identifier function must have the following syntax:

```
truthID = truthIDIdentifier(truths)
```

**truths** is the same as the **truths** array passed as input for **trackErrorMetric** updates. **truthID** must have the same size as **truths**. The default identification function handle, `@defaultTruthIdentifier`, assumes **truths** is an array or structures or objects with a 'PlatformID' field name or property.

#### Dependencies

To enable this property, set the **ErrorFunctionFormat** property to 'custom'.

Data Types: `function_handle`

## Usage

To estimate errors, call the track error metrics object with arguments, as if it were a function (described here).

## Syntax

```
[posRMSE,velRMSE,posANEES,velANEES] = errorMetrics(tracks,trackIDs,truths,
truthIDs)
[posRMSE,velRMSE,accRMSE,posANEES,velANEES,accANEES] = errorMetrics(tracks,
trackIDs,truths,truthIDs)
[posRMSE,velRMSE,yawRateRMSE,posANEES,velANEES,yawRateANEES] = errorMetrics(
tracks,trackIDs,truths,truthIDs)

[out1,out2, ... ,outN] = errorMetrics(tracks,trackIDs,truths,truthIDs)
```

## Description

`[posRMSE,velRMSE,posANEES,velANEES] = errorMetrics(tracks,trackIDs,truths,truthIDs)` returns the metrics

- `posRMSE` – Position root mean squared error
- `velRMSE` – Velocity root mean squared error
- `posANEES` – Position average normalized-estimation error squared
- `velANEES` – Velocity average normalized-estimation error squared

for constant velocity motion at the current time step. `trackIDs` is the set of track identifiers for all tracks. `truthIDs` is the set of truth identifiers. `tracks` are the set of tracks, and `truths` are the set of truths. `trackIDs` and `truthIDs` are each a vector whose corresponding elements match the track and truth identifiers found in `tracks` and `truths`, respectively.

The RMSE and ANEES values for different states are calculated by averaging the errors of all tracks at the current time step. For example, the position RMSE value, `posRMSE`, is defined as:

$$\text{posRMSE} = \sqrt{\frac{1}{M} \sum_{i=1}^M \|\Delta p_i\|^2}$$

where  $M$  is the total number of tracks with associated truth trajectories in the current time step, and

$$\Delta p_i = p_{track,i} - p_{truth,i}$$

is the position difference between the position of track  $i$ ,  $p_{track,i}$ , and the position of the corresponding truth,  $p_{truth,i}$ , at the current time step. The RMSE values for other states (`vel`, `pos`, `acc`, and `yawRate`) are defined similarly.

The position ANEES value, `posANEES`, is defined as:

$$\text{posANEES} = \frac{1}{M} \sum_{i=1}^M \Delta p_i^T C_{p,i}^{-1} \Delta p_i$$

where  $C_{p,i}$  is the covariance matrix corresponding to the position of track  $i$  at the current time step. The ANEES values for other states (`vel`, `pos`, `acc`, and `yawRate`) are defined similarly.

To enable this syntax, set the `ErrorFunctionFormat` property to 'built-in' and the `MotionModel` property to 'constvel'.

`[posRMSE,velRMSE,accRMSE,posANEES,velANEES,accANEES] = errorMetrics(tracks,trackIDs,truths,truthIDs)` also returns the metrics

- `accRMSE` – Acceleration root mean squared error
- `accANEES` – acceleration average normalized-estimation error squared

for constant acceleration motion at the current time step.

To enable this syntax, set the `ErrorFunctionFormat` property to 'built-in' and the `MotionModel` property to 'constacc'.

`[posRMSE,velRMSE,yawRateRMSE,posANEES,velANEES,yawRateANEES] = errorMetrics(tracks,trackIDs,truths,truthIDs)` also returns the metrics

- `yawRateRMSE` – yaw rate root mean squared error
- `yawRateANEES` – yaw rate average normalized-estimation error squared

for constant turn-rate motion at the current time step.

To enable this syntax, set the `ErrorFunctionFormat` property to 'built-in' and the `MotionModel` property to 'constturn'.

`[out1,out2, ...,outN] = errorMetrics(tracks,trackIDs,truths,truthIDs)` returns the user-defined metrics `out1, out2, ..., outN`.

To enable this syntax, set the `ErrorFunctionFormat` property to 'custom'. The number of outputs corresponds to the number of elements listed in the `EstimationErrorLabels` property, and must match the number of outputs in the `EstimationErrorFcn`. The results of the estimation errors are averaged arithmetically over all track-to-truth assignments.

---

**Tip** These usage syntaxes only calculate the RMSE and ANEES values of all tracks with associated truths at the current time step. To obtain the cumulative RMSE and ANEES values for each track and truth, use the `cumulativeTrackMetrics` and `cumulativeTruthMetrics` object functions, respectively. To obtain the current RMSE and ANEES values for each track and truth, use the `currentTrackMetrics` and `currentTruthMetrics` object functions, respectively.

---

## Input Arguments

### **tracks — Track information**

array of structures | array of objects

Track information, specified as an array of structures or objects. For built-in trackers such as `trackerGNN` or `trackerTOMHT`, the `objectTrack` output contains 'State', 'StateCovariance', and 'TrackID' information.

Data Types: `struct`

### **trackIDs — Track identifiers**

real-valued vector

Track identifiers, specified as a real-valued vector. `trackIDs` elements match the tracks found in `tracks`.

**truths — Truth information**

array of structures | array of objects

Truth information, specified as an array of structures or objects. When using a `trackingScenario`, truth information can be obtained from the `platformPoses` object function.

Data Types: struct

**truthIDs — Truth identifiers**

real-valued vector

Truth identifiers, specified as a real-valued vector. `truthIDs` elements match the truths found in `truths`.

**Output Arguments****posRMSE — Position root mean squared error**

scalar

Position root mean squared error for all tracks associated with `truths`, returned as a scalar.

**Dependencies**

To enable this argument, set the `ErrorFunctionFormat` property to 'built-in'.

**velRMSE — Velocity root mean squared error**

scalar

Velocity root mean squared error for all tracks associated with `truths`, returned as a scalar.

**Dependencies**

To enable this argument, set the `ErrorFunctionFormat` property to 'built-in'.

**accRMSE — Acceleration root mean squared error**

scalar

Acceleration root mean squared error for all tracks associated with `truths`, returned as a scalar.

**Dependencies**

To enable this argument, set the `ErrorFunctionFormat` property to 'built-in'.

**yawRateRMSE — Yaw rate root mean squared error**

scalar

Yaw rate root mean squared error for all tracks associated with `truths`, returned as a scalar.

**Dependencies**

To enable this argument, set the `ErrorFunctionFormat` property to 'built-in'.

**posANEES — Position average normalized estimation error squared**

scalar

Position average normalized estimation error squared for all tracks associated with truths, returned as a scalar.

**Dependencies**

To enable this argument, set the `ErrorFunctionFormat` property to 'built-in'.

**velANEES — Velocity average normalized estimation error squared**  
scalar

Velocity average normalized estimation error squared for all tracks associated with truths, returned as a scalar.

**Dependencies**

To enable this argument, set the `ErrorFunctionFormat` property to 'built-in'.

**accANEES — Acceleration average normalized estimation error squared**  
scalar

Acceleration average normalized estimation error squared for all tracks associated with truths, returned as a scalar.

**Dependencies**

To enable this argument, set the `ErrorFunctionFormat` property to 'built-in'.

**yawRateANEES — Yaw rate average normalized estimation error squared**  
scalar

Yaw rate average normalized estimation error squared for all tracks associated with truths, returned as a scalar.

**Dependencies**

To enable this argument, set the `ErrorFunctionFormat` property to 'built-in'.

**out1, out2, outN — Custom error metric outputs**  
scalar

Custom error metric outputs, returned as scalars. These errors are the output of the error estimation function specified in the `EstimationErrorFcn` property.

**Dependencies**

To enable these arguments, set the `ErrorFunctionFormat` property to 'custom'.

## 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 trackErrorMetrics

`cumulativeTrackMetrics` Cumulative metrics for recent tracks

---

cumulativeTruthMetrics	Cumulative metrics for recent truths
currentTrackMetrics	Metrics for recent tracks
currentTruthMetrics	Metrics for recent truths

## Common to All System Objects

release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isLocked	Determine if System object is in use
clone	Create duplicate System object

## Examples

### Assignment and Error Metrics for Two Tracked Targets

Examine the assignments and errors for a system tracking two targets.

First, load the stored track data.

```
load trackmetricex tracklog truthlog
```

Create objects to analyze assignment and error metrics.

```
tam = trackAssignmentMetrics;
tem = trackErrorMetrics;
```

Create the output variables.

```
posRMSE = zeros(numel(tracklog),1);
velRMSE = zeros(numel(tracklog),1);
posANEES = zeros(numel(tracklog),1);
velANEES = zeros(numel(tracklog),1);
```

Loop over all tracks to:

- Extract the tracks and ground truth at the  $i$  th tracker update.
- Analyze and retrieve the current track-to-truth assignment.
- Analyze instantaneous error metrics over all tracks and truths.

```
for i=1:numel(tracklog)
    tracks = tracklog{i};
    truths = truthlog{i};
    [trackAM,truthAM] = tam(tracks, truths);
    [trackIDs,truthIDs] = currentAssignment(tam);
    [posRMSE(i),velRMSE(i),posANEES(i),velANEES(i)] = ...
        tem(tracks,trackIDs,truths,truthIDs);
end
```

Show the track metrics table.

```
trackMetricsTable(tam)
```

```
ans=4x15 table
    TrackID    AssignedTruthID    Surviving    TotalLength    DeletionStatus    DeletionLength
```

1	NaN	false	1120	false	0	
2	NaN	false	1736	false	0	
6	3	true	1138	false	0	
8	2	true	662	false	0	

Show the truth metrics table.

```
truthMetricsTable(tam)
```

ans=2×10 table						
TruthID	AssociatedTrackID	DeletionStatus	TotalLength	BreakStatus	BreakCount	
2	8	false	2678	false	4	
3	6	false	2678	false	3	

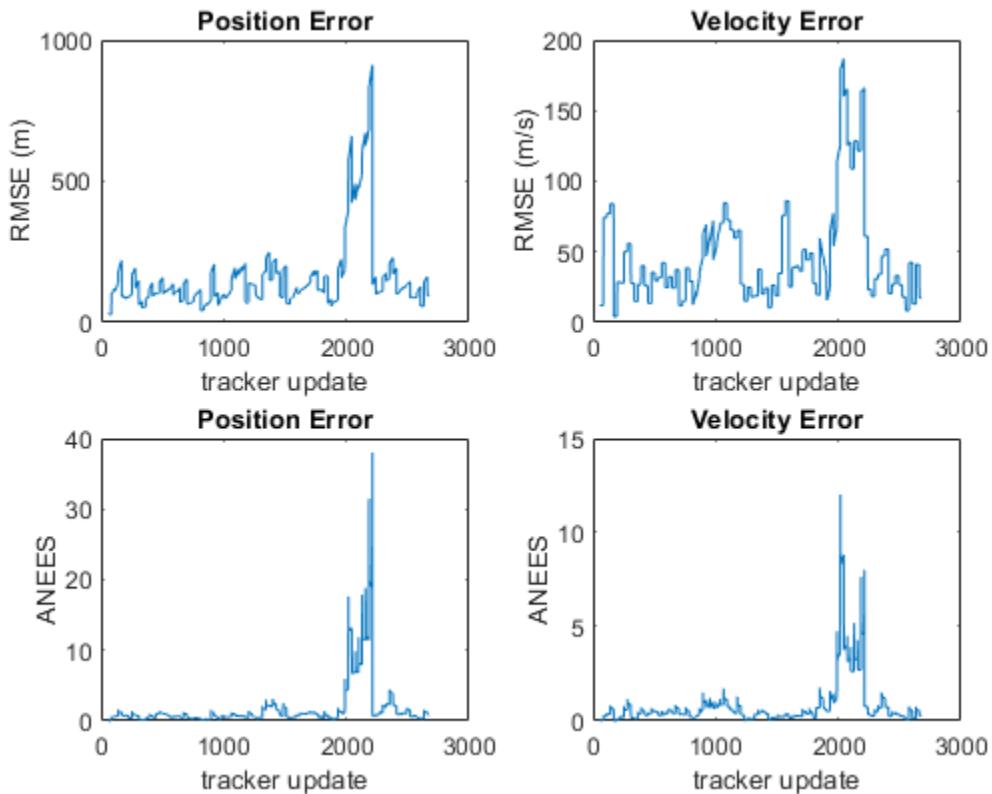
Plot the RMSE and ANEES error metrics.

```
subplot(2,2,1)
plot(posRMSE)
title('Position Error')
xlabel('tracker update')
ylabel('RMSE (m)')

subplot(2,2,2)
plot(velRMSE)
title('Velocity Error')
xlabel('tracker update')
ylabel('RMSE (m/s)')

subplot(2,2,3)
plot(posANEES)
title('Position Error')
xlabel('tracker update')
ylabel('ANEES')

subplot(2,2,4)
plot(velANEES)
title('Velocity Error')
xlabel('tracker update')
ylabel('ANEES')
```



Show the current error metrics for each individual recorded track.

```
currentTrackMetrics(tem)
```

```
ans=2×5 table
TrackID    posRMS    velRMS    posANEES    velANEES
_____
6          44.712    20.988    0.05974    0.31325
8          129.26    12.739    1.6745     0.2453
```

Show the current error metrics for each individual recorded truth object.

```
currentTruthMetrics(tem)
```

```
ans=2×5 table
TruthID    posRMS    velRMS    posANEES    velANEES
_____
2          129.26    12.739    1.6745     0.2453
3          44.712    20.988    0.05974    0.31325
```

Show the cumulative error metrics for each individual recorded track.

```
cumulativeTrackMetrics(tem)
```

ans=4×5 *table*

TrackID	posRMS	velRMS	posANEES	velANEES
1	117.69	43.951	0.58338	0.44127
2	129.7	42.8	0.81094	0.42509
6	371.35	87.083	4.5208	1.6952
8	130.45	53.914	1.0448	0.44813

Show the cumulative error metrics for each individual recorded truth object.

```
cumulativeTruthMetrics(tem)
```

ans=2×5 *table*

TruthID	posRMS	velRMS	posANEES	velANEES
2	258.21	65.078	2.2514	0.93359
3	134.41	48.253	0.96314	0.49183

## See Also

### **System Objects**

[monostaticRadarSensor](#) | [radarSensor](#) | [trackAssignmentMetrics](#) | [trackOSPAMetric](#) | [trackerGNN](#) | [trackerJPDA](#) | [trackerPHD](#) | [trackerTOMHT](#)

**Introduced in R2018b**

# cumulativeTrackMetrics

Cumulative metrics for recent tracks

## Syntax

```
metricsTable = cumulativeTrackMetrics(errorMetrics)
```

## Description

`metricsTable = cumulativeTrackMetrics(errorMetrics)` returns a table of cumulative metrics, `metricsTable`, for every track identifier provided in the most recent update.

## Examples

### Assignment and Error Metrics for Two Tracked Targets

Examine the assignments and errors for a system tracking two targets.

First, load the stored track data.

```
load trackmetricex tracklog truthlog
```

Create objects to analyze assignment and error metrics.

```
tam = trackAssignmentMetrics;
tem = trackErrorMetrics;
```

Create the output variables.

```
posRMSE = zeros(numel(tracklog),1);
velRMSE = zeros(numel(tracklog),1);
posANEES = zeros(numel(tracklog),1);
velANEES = zeros(numel(tracklog),1);
```

Loop over all tracks to:

- Extract the tracks and ground truth at the  $i$  th tracker update.
- Analyze and retrieve the current track-to-truth assignment.
- Analyze instantaneous error metrics over all tracks and truths.

```
for i=1:numel(tracklog)
    tracks = tracklog{i};
    truths = truthlog{i};
    [trackAM,truthAM] = tam(tracks, truths);
    [trackIDs,truthIDs] = currentAssignment(tam);
    [posRMSE(i),velRMSE(i),posANEES(i),velANEES(i)] = ...
        tem(tracks,trackIDs,truths,truthIDs);
end
```

Show the track metrics table.

```
trackMetricsTable(tam)
```

```
ans=4×15 table
```

TrackID	AssignedTruthID	Surviving	TotalLength	DeletionStatus	DeletionLength
1	NaN	false	1120	false	0
2	NaN	false	1736	false	0
6	3	true	1138	false	0
8	2	true	662	false	0

Show the truth metrics table.

```
truthMetricsTable(tam)
```

```
ans=2×10 table
```

TruthID	AssociatedTrackID	DeletionStatus	TotalLength	BreakStatus	BreakCount
2	8	false	2678	false	4
3	6	false	2678	false	3

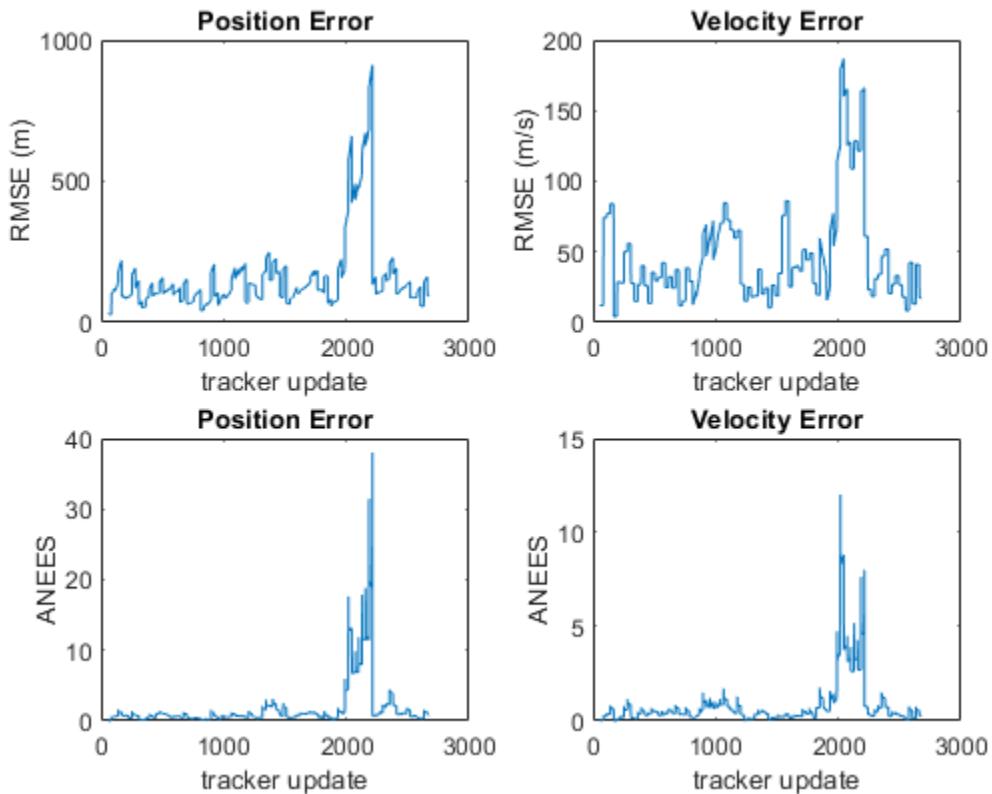
Plot the RMSE and ANEES error metrics.

```
subplot(2,2,1)
plot(posRMSE)
title('Position Error')
xlabel('tracker update')
ylabel('RMSE (m)')

subplot(2,2,2)
plot(velRMSE)
title('Velocity Error')
xlabel('tracker update')
ylabel('RMSE (m/s)')

subplot(2,2,3)
plot(posANEES)
title('Position Error')
xlabel('tracker update')
ylabel('ANEES')

subplot(2,2,4)
plot(velANEES)
title('Velocity Error')
xlabel('tracker update')
ylabel('ANEES')
```



Show the current error metrics for each individual recorded track.

```
currentTrackMetrics(tem)
```

```
ans=2×5 table
TrackID    posRMS    velRMS    posANEES    velANEES
_____
6          44.712    20.988    0.05974    0.31325
8          129.26    12.739    1.6745     0.2453
```

Show the current error metrics for each individual recorded truth object.

```
currentTruthMetrics(tem)
```

```
ans=2×5 table
TruthID    posRMS    velRMS    posANEES    velANEES
_____
2          129.26    12.739    1.6745     0.2453
3          44.712    20.988    0.05974    0.31325
```

Show the cumulative error metrics for each individual recorded track.

```
cumulativeTrackMetrics(tem)
```

ans=4x5 table				
TrackID	posRMS	velRMS	posANEES	velANEES
1	117.69	43.951	0.58338	0.44127
2	129.7	42.8	0.81094	0.42509
6	371.35	87.083	4.5208	1.6952
8	130.45	53.914	1.0448	0.44813

Show the cumulative error metrics for each individual recorded truth object.

```
cumulativeTruthMetrics(tem)
```

ans=2x5 table				
TruthID	posRMS	velRMS	posANEES	velANEES
2	258.21	65.078	2.2514	0.93359
3	134.41	48.253	0.96314	0.49183

## Input Arguments

**errorMetrics — Error metrics object**

trackErrorMetrics System object

Error metrics object, specified as a trackErrorMetrics System object.

## Output Arguments

**metricsTable — Track error metrics**

table

Track error metrics, returned as a table.

- When you set the **ErrorFunctionFormat** property of the input error metrics object to 'built-in', the table columns depend on the setting of the **MotionModel** property.

Motion Model	Table Columns
'constvel'	posRMSE, velRMSE, posANEES, velANEES
'constacc'	posRMSE, velRMSE, accRMSE, posANEES, velANEES, accANEES
'constturn'	posRMSE, velRMSE, yawRateRMSE, posANEES, velANEES, yawRateANEES

RMSE and ANEES denote root mean squared error and average normalized estimation error squared of a track for the entire tracking scenario time history. For example, the cumulative position RMSE value for a track is defined as:

$$\text{posRMSE} = \sqrt{\frac{1}{N} \sum_{t=1}^N \|\Delta p_t\|^2}$$

where  $N$  is the total number of time steps that the track has an associated truth.

$$\Delta p_t = p_{track,t} - p_{truth,t}$$

is the difference between the position of the track at time step  $t$ ,  $p_{track,t}$ , and the position of the associated truth at time step  $t$ ,  $p_{truth,t}$ . The cumulative RMSE values for other states (vel, pos, acc, and yawRate) are defined similarly. The position ANEES value, posANEES is defined as:

$$posANEES = \frac{1}{N} \sum_{t=1}^N \Delta p_t^T C_{p,t}^{-1} \Delta p_t$$

where  $C_{p,t}$  is the covariance corresponding to the position of the track at time step  $t$ . The ANEES values for other states (vel, pos, acc, and yawRate) are defined similarly.

- When you set the `ErrorFunctionFormat` property to 'custom', the table contains the arithmetically averaged values of the custom metrics output from the error function.

### **Introduced in R2018b**

# cumulativeTruthMetrics

Cumulative metrics for recent truths

## Syntax

```
metricsTable = cumulativeTruthMetrics(errorMetrics)
```

## Description

`metricsTable = cumulativeTruthMetrics(errorMetrics)` returns a table of cumulative metrics, `metricsTable`, for every truth identifier provided in the most recent update.

## Examples

### Assignment and Error Metrics for Two Tracked Targets

Examine the assignments and errors for a system tracking two targets.

First, load the stored track data.

```
load trackmetricex tracklog truthlog
```

Create objects to analyze assignment and error metrics.

```
tam = trackAssignmentMetrics;
tem = trackErrorMetrics;
```

Create the output variables.

```
posRMSE = zeros(numel(tracklog),1);
velRMSE = zeros(numel(tracklog),1);
posANEES = zeros(numel(tracklog),1);
velANEES = zeros(numel(tracklog),1);
```

Loop over all tracks to:

- Extract the tracks and ground truth at the  $i$  th tracker update.
- Analyze and retrieve the current track-to-truth assignment.
- Analyze instantaneous error metrics over all tracks and truths.

```
for i=1:numel(tracklog)
    tracks = tracklog{i};
    truths = truthlog{i};
    [trackAM,truthAM] = tam(tracks, truths);
    [trackIDs,truthIDs] = currentAssignment(tam);
    [posRMSE(i),velRMSE(i),posANEES(i),velANEES(i)] = ...
        tem(tracks,trackIDs,truths,truthIDs);
end
```

Show the track metrics table.

```
trackMetricsTable(tam)
```

```
ans=4×15 table
```

TrackID	AssignedTruthID	Surviving	TotalLength	DeletionStatus	DeletionLength
1	NaN	false	1120	false	0
2	NaN	false	1736	false	0
6	3	true	1138	false	0
8	2	true	662	false	0

Show the truth metrics table.

```
truthMetricsTable(tam)
```

```
ans=2×10 table
```

TruthID	AssociatedTrackID	DeletionStatus	TotalLength	BreakStatus	BreakCount
2	8	false	2678	false	4
3	6	false	2678	false	3

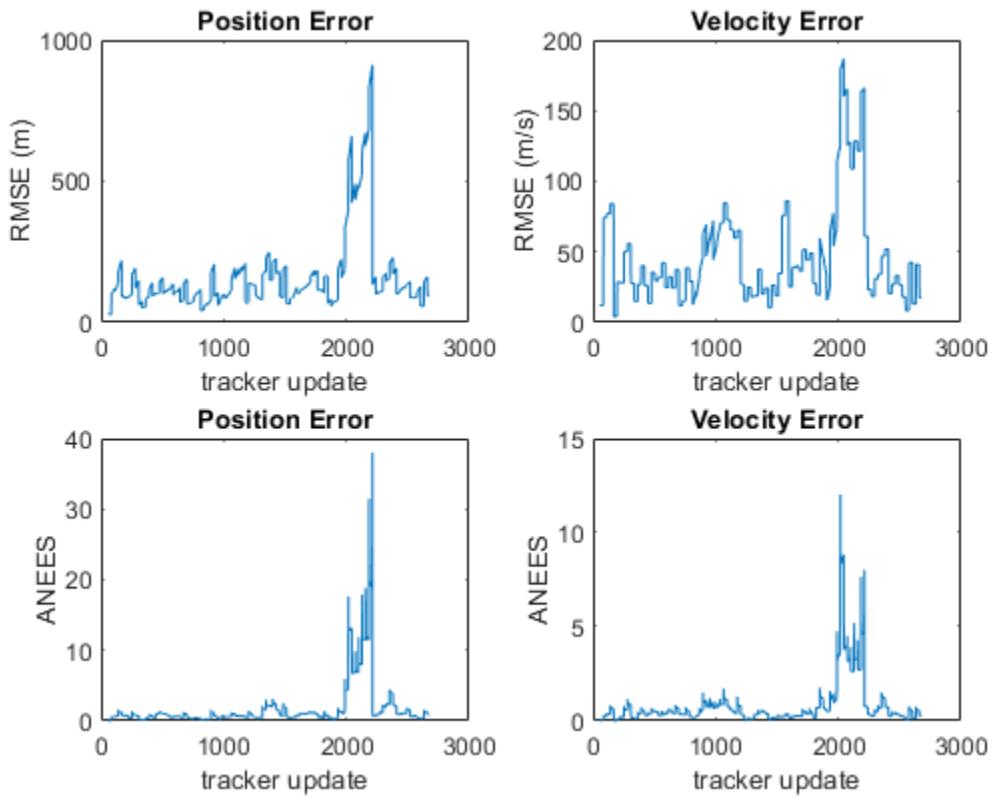
Plot the RMSE and ANEES error metrics.

```
subplot(2,2,1)
plot(posRMSE)
title('Position Error')
xlabel('tracker update')
ylabel('RMSE (m)')

subplot(2,2,2)
plot(velRMSE)
title('Velocity Error')
xlabel('tracker update')
ylabel('RMSE (m/s)')

subplot(2,2,3)
plot(posANEES)
title('Position Error')
xlabel('tracker update')
ylabel('ANEES')

subplot(2,2,4)
plot(velANEES)
title('Velocity Error')
xlabel('tracker update')
ylabel('ANEES')
```



Show the current error metrics for each individual recorded track.

```
currentTrackMetrics(tem)
```

```
ans=2×5 table
TrackID    posRMS    velRMS    posANEES    velANEES
_____
6          44.712    20.988    0.05974    0.31325
8          129.26    12.739    1.6745     0.2453
```

Show the current error metrics for each individual recorded truth object.

```
currentTruthMetrics(tem)
```

```
ans=2×5 table
TruthID    posRMS    velRMS    posANEES    velANEES
_____
2          129.26    12.739    1.6745     0.2453
3          44.712    20.988    0.05974    0.31325
```

Show the cumulative error metrics for each individual recorded track.

```
cumulativeTrackMetrics(tem)
```

ans=4x5 table				
TrackID	posRMS	velRMS	posANEES	velANEES
1	117.69	43.951	0.58338	0.44127
2	129.7	42.8	0.81094	0.42509
6	371.35	87.083	4.5208	1.6952
8	130.45	53.914	1.0448	0.44813

Show the cumulative error metrics for each individual recorded truth object.

```
cumulativeTruthMetrics(tem)
```

ans=2x5 table				
TruthID	posRMS	velRMS	posANEES	velANEES
2	258.21	65.078	2.2514	0.93359
3	134.41	48.253	0.96314	0.49183

## Input Arguments

### errorMetrics — Error metrics object

trackErrorMetrics System object

Error metrics object, specified as a trackErrorMetrics System object.

## Output Arguments

### metricsTable — Truth error metrics

table

Truth error metrics, returned as a table.

- When you set the `ErrorFunctionFormat` property of the input error metrics object to 'built-in', the table columns depend on the setting of the `MotionModel` property.

Motion Model	Table Columns
'constvel'	posRMSE, velRMSE, posANEES, velANEES
'constacc'	posRMSE, velRMSE, accRMSE, posANEES, velANEES, accANEES
'constturn'	posRMSE, velRMSE, yawRateRMSE, posANEES, velANEES, yawRateANEES

RMSE and ANEES denote root mean squared error and average normalized estimation error squared of a truth trajectory for the entire tracking scenario time history. Since a truth trajectory can associate with multiple tracks at a time step, the calculation of cumulative RMSE and ANEES values is each separated into two steps. For example, in the first step of the position RMSE value calculation, the function first calculates the RMSE value at a given time step  $t$  as:

$$S_t = \sum_{k=1}^{K_t} \|\Delta p_{t,k}\|^2$$

where  $K_t$  is the number of tracks associated with the truth at time step  $t$ , and

$$\Delta p_{t,k} = p_{track,t,k} - p_{truth,t}$$

is the position difference between the position of  $k$ th associated track and the position of the truth at time step  $t$ . In the second step, the  $S_t$  values of all the time steps ( $t = 1, 2, \dots, N$ ) are summed and averaged over the total number of associated tracks (denoted by  $R$ ) to obtain the cumulative position RMSE value as:

$$\text{posRMSE} = \sqrt{\frac{1}{\sum_{t=1}^N K_t} \sum_{t=1}^N \sum_{k=1}^{K_t} \|\Delta p_{t,k}\|^2}$$

where the total number of associated tracks,  $R$ , is given by

$$R = \sum_{t=1}^N K_t.$$

The cumulative RMSE values for other states (`vel`, `pos`, `acc`, and `yawRate`) are defined similarly.

The calculation of the cumulative position ANEES value, `posANEES`, for a truth trajectory can also be separated into two steps. In the first step, the function calculates the ANEES value at a given time step  $t$  as:

$$Q_t = \sum_{k=1}^{K_t} \Delta p_{t,k}^T C_{p,t,k}^{-1} \Delta p_{t,k}$$

where  $C_{p,t,k}$  is the covariance corresponding to the position of the  $k$ th associated track at time step  $t$ . In the second step, the  $Q_t$  values for all the time steps ( $t = 1, 2, \dots, N$ ) are summed and averaged over the total number of associated tracks (denoted by  $R$ ) to obtain the cumulative position ANEES value as:

$$\text{posANEES} = \frac{1}{\sum_{t=1}^N K_t} \sum_{t=1}^N \sum_{k=1}^{K_t} \Delta p_{t,k}^T C_{p,t,k}^{-1} \Delta p_{t,k}$$

The cumulative ANEES values for other states (`vel`, `pos`, `acc`, and `yawRate`) are defined similarly.

- When you set the `ErrorFunctionFormat` property to '`custom`', the table contains the arithmetically averaged values of the custom metrics output from the error function.

### Introduced in R2018b

# currentTrackMetrics

Metrics for recent tracks

## Syntax

```
metricsTable = currentTrackMetrics(errorMetrics)
```

## Description

`metricsTable = currentTrackMetrics(errorMetrics)` returns a table of metrics, `metricsTable`, for every track identifier provided in the most recent update.

## Examples

### Assignment and Error Metrics for Two Tracked Targets

Examine the assignments and errors for a system tracking two targets.

First, load the stored track data.

```
load trackmetricex tracklog truthlog
```

Create objects to analyze assignment and error metrics.

```
tam = trackAssignmentMetrics;
tem = trackErrorMetrics;
```

Create the output variables.

```
posRMSE = zeros(numel(tracklog),1);
velRMSE = zeros(numel(tracklog),1);
posANEES = zeros(numel(tracklog),1);
velANEES = zeros(numel(tracklog),1);
```

Loop over all tracks to:

- Extract the tracks and ground truth at the  $i$  th tracker update.
- Analyze and retrieve the current track-to-truth assignment.
- Analyze instantaneous error metrics over all tracks and truths.

```
for i=1:numel(tracklog)
    tracks = tracklog{i};
    truths = truthlog{i};
    [trackAM,truthAM] = tam(tracks, truths);
    [trackIDs,truthIDs] = currentAssignment(tam);
    [posRMSE(i),velRMSE(i),posANEES(i),velANEES(i)] = ...
        tem(tracks,trackIDs,truths,truthIDs);
end
```

Show the track metrics table.

```
trackMetricsTable(tam)
```

```
ans=4×15 table
```

TrackID	AssignedTruthID	Surviving	TotalLength	DeletionStatus	DeletionLength
1	NaN	false	1120	false	0
2	NaN	false	1736	false	0
6	3	true	1138	false	0
8	2	true	662	false	0

Show the truth metrics table.

```
truthMetricsTable(tam)
```

```
ans=2×10 table
```

TruthID	AssociatedTrackID	DeletionStatus	TotalLength	BreakStatus	BreakCount
2	8	false	2678	false	4
3	6	false	2678	false	3

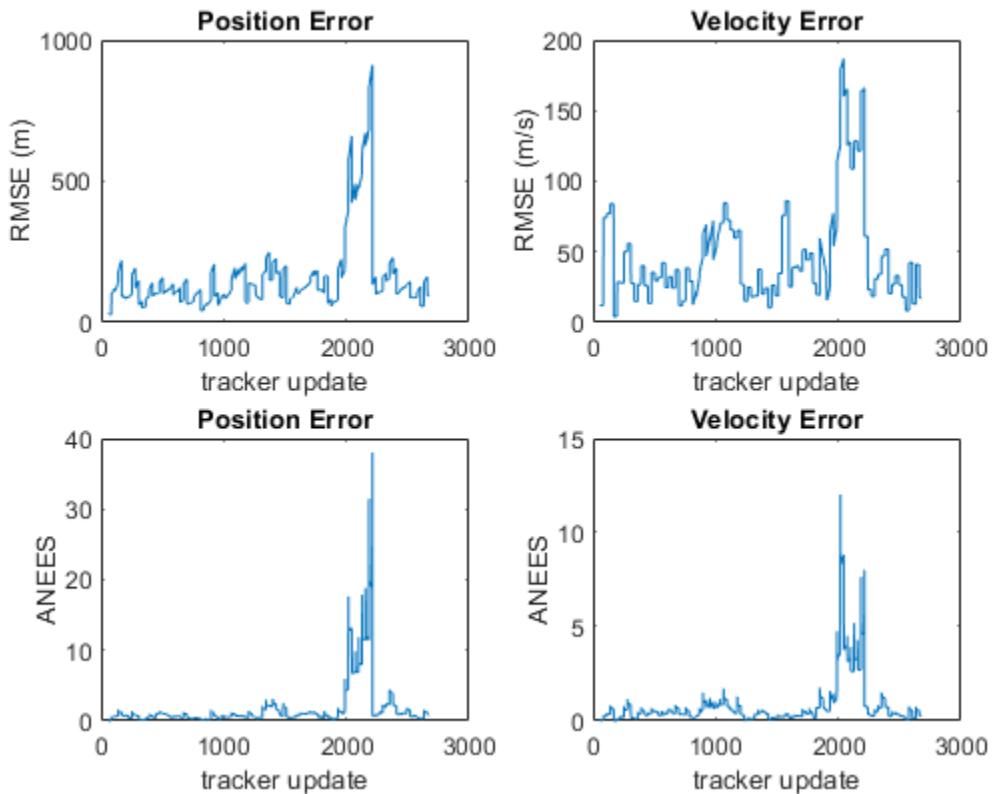
Plot the RMSE and ANEES error metrics.

```
subplot(2,2,1)
plot(posRMSE)
title('Position Error')
xlabel('tracker update')
ylabel('RMSE (m)')

subplot(2,2,2)
plot(velRMSE)
title('Velocity Error')
xlabel('tracker update')
ylabel('RMSE (m/s)')

subplot(2,2,3)
plot(posANEES)
title('Position Error')
xlabel('tracker update')
ylabel('ANEES')

subplot(2,2,4)
plot(velANEES)
title('Velocity Error')
xlabel('tracker update')
ylabel('ANEES')
```



Show the current error metrics for each individual recorded track.

```
currentTrackMetrics(tem)
```

TrackID	posRMS	velRMS	posANEES	velANEES
6	44.712	20.988	0.05974	0.31325
8	129.26	12.739	1.6745	0.2453

Show the current error metrics for each individual recorded truth object.

```
currentTruthMetrics(tem)
```

TruthID	posRMS	velRMS	posANEES	velANEES
2	129.26	12.739	1.6745	0.2453
3	44.712	20.988	0.05974	0.31325

Show the cumulative error metrics for each individual recorded track.

```
cumulativeTrackMetrics(tem)
```

ans=4x5 table				
TrackID	posRMS	velRMS	posANEES	velANEES
1	117.69	43.951	0.58338	0.44127
2	129.7	42.8	0.81094	0.42509
6	371.35	87.083	4.5208	1.6952
8	130.45	53.914	1.0448	0.44813

Show the cumulative error metrics for each individual recorded truth object.

```
cumulativeTruthMetrics(tem)
```

ans=2x5 table				
TruthID	posRMS	velRMS	posANEES	velANEES
2	258.21	65.078	2.2514	0.93359
3	134.41	48.253	0.96314	0.49183

## Input Arguments

**errorMetrics — Error metrics object**

trackErrorMetrics System object

Error metrics object, specified as a trackErrorMetrics System object.

## Output Arguments

**metricsTable — Track error metrics**

table

Track error metrics, returned as a table:

- When you set the **ErrorFunctionFormat** property of the input error metrics object to 'built-in', the table columns depend on the setting of the **MotionModel** property.

Motion Model	Table Columns
'constvel'	posRMSE, velRMSE, posANEES, velANEES
'constacc'	posRMSE, velRMSE, accRMSE, posANEES, velANEES, accANEES
'constturn'	posRMSE, velRMSE, yawRateRMSE, posANEES, velANEES, yawRateANEES

RMSE and ANEES denote root mean squared error and average normalized estimation error squared of a track at the current time step. For example, the position RMSE and ANEES values for a track are defined respectively as:

$$\text{posRMSE} = \|\Delta p_i\| = \|p_{track,i} - p_{truth,i}\|$$

$$\text{posANEES} = \Delta p_i^T C_i^{-1} \Delta p_i$$

where  $p_{track, i}$  is the position of track  $i$ ,  $p_{truth, i}$  is the position of the truth associated to track  $i$ , and  $C_i$  is the position covariance of track  $i$  at the current time step. Note that the RMSE and ANEES values are only calculated for one time step using the `currentTrackMetrics`. The RMSE and ANEES values for other states (`vel`, `pos`, `acc`, and `yawRate`) are defined similarly.

- When you set the `ErrorFunctionFormat` property to '`custom`', the table contains the arithmetically averaged values of the custom metrics output from the error function.

**Introduced in R2018b**

# currentTruthMetrics

Metrics for recent truths

## Syntax

```
metricsTable = currentTruthMetrics(errorMetrics)
```

## Description

`metricsTable = currentTruthMetrics(errorMetrics)` returns a table of metrics, `metricsTable`, for every truth identifier provided in the most recent update.

## Examples

### Assignment and Error Metrics for Two Tracked Targets

Examine the assignments and errors for a system tracking two targets.

First, load the stored track data.

```
load trackmetricex tracklog truthlog
```

Create objects to analyze assignment and error metrics.

```
tam = trackAssignmentMetrics;
tem = trackErrorMetrics;
```

Create the output variables.

```
posRMSE = zeros(numel(tracklog),1);
velRMSE = zeros(numel(tracklog),1);
posANEES = zeros(numel(tracklog),1);
velANEES = zeros(numel(tracklog),1);
```

Loop over all tracks to:

- Extract the tracks and ground truth at the  $i$  th tracker update.
- Analyze and retrieve the current track-to-truth assignment.
- Analyze instantaneous error metrics over all tracks and truths.

```
for i=1:numel(tracklog)
    tracks = tracklog{i};
    truths = truthlog{i};
    [trackAM,truthAM] = tam(tracks, truths);
    [trackIDs,truthIDs] = currentAssignment(tam);
    [posRMSE(i),velRMSE(i),posANEES(i),velANEES(i)] = ...
        tem(tracks,trackIDs,truths,truthIDs);
end
```

Show the track metrics table.

```
trackMetricsTable(tam)
```

```
ans=4×15 table
```

TrackID	AssignedTruthID	Surviving	TotalLength	DeletionStatus	DeletionLength
1	NaN	false	1120	false	0
2	NaN	false	1736	false	0
6	3	true	1138	false	0
8	2	true	662	false	0

Show the truth metrics table.

```
truthMetricsTable(tam)
```

```
ans=2×10 table
```

TruthID	AssociatedTrackID	DeletionStatus	TotalLength	BreakStatus	BreakCount
2	8	false	2678	false	4
3	6	false	2678	false	3

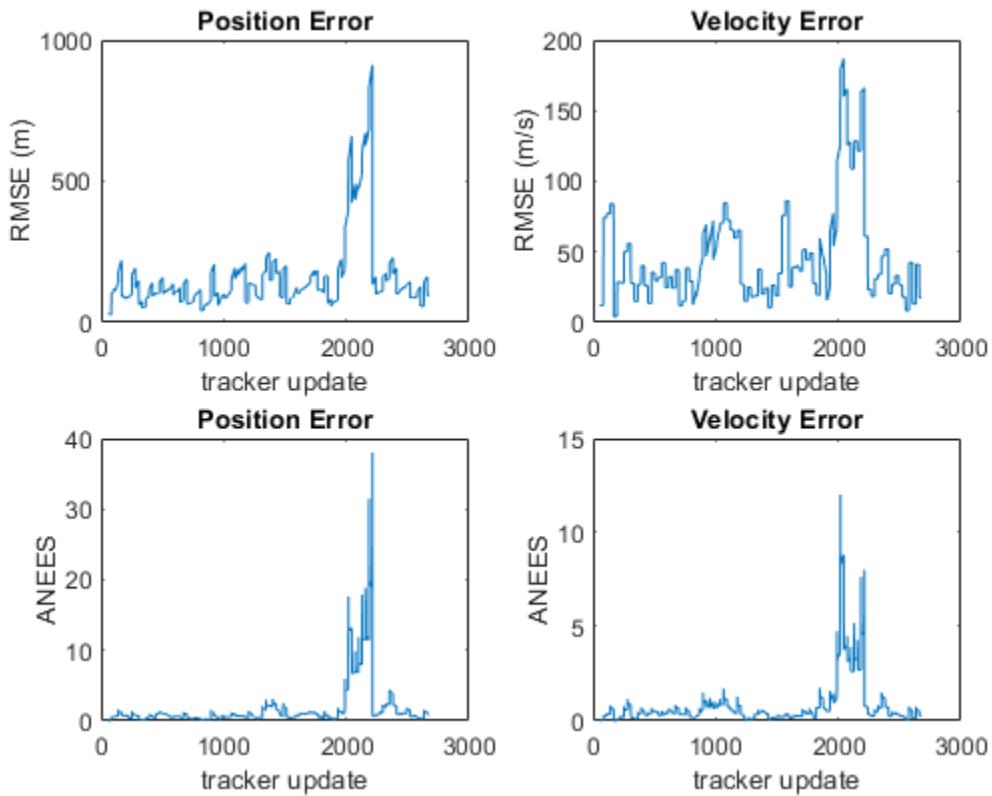
Plot the RMSE and ANEES error metrics.

```
subplot(2,2,1)
plot(posRMSE)
title('Position Error')
xlabel('tracker update')
ylabel('RMSE (m)')

subplot(2,2,2)
plot(velRMSE)
title('Velocity Error')
xlabel('tracker update')
ylabel('RMSE (m/s)')

subplot(2,2,3)
plot(posANEES)
title('Position Error')
xlabel('tracker update')
ylabel('ANEES')

subplot(2,2,4)
plot(velANEES)
title('Velocity Error')
xlabel('tracker update')
ylabel('ANEES')
```



Show the current error metrics for each individual recorded track.

```
currentTrackMetrics(tem)
```

```
ans=2×5 table
TrackID    posRMS    velRMS    posANEES    velANEES
_____
6          44.712    20.988    0.05974    0.31325
8          129.26    12.739    1.6745     0.2453
```

Show the current error metrics for each individual recorded truth object.

```
currentTruthMetrics(tem)
```

```
ans=2×5 table
TruthID    posRMS    velRMS    posANEES    velANEES
_____
2          129.26    12.739    1.6745     0.2453
3          44.712    20.988    0.05974    0.31325
```

Show the cumulative error metrics for each individual recorded track.

```
cumulativeTrackMetrics(tem)
```

ans=4x5 table				
TrackID	posRMS	velRMS	posANEES	velANEES
1	117.69	43.951	0.58338	0.44127
2	129.7	42.8	0.81094	0.42509
6	371.35	87.083	4.5208	1.6952
8	130.45	53.914	1.0448	0.44813

Show the cumulative error metrics for each individual recorded truth object.

```
cumulativeTruthMetrics(tem)
```

ans=2x5 table				
TruthID	posRMS	velRMS	posANEES	velANEES
2	258.21	65.078	2.2514	0.93359
3	134.41	48.253	0.96314	0.49183

## Input Arguments

### errorMetrics — Error metrics object

trackErrorMetrics System object

Error metrics object, specified as a trackErrorMetrics System object.

## Output Arguments

### metricsTable — Truth error metrics

table

Truth error metrics, returned as a table.

- When you set the `ErrorFunctionFormat` property of the input error metrics object to 'built-in', the table columns depend on the setting of the `MotionModel` property.

Motion model	Table Columns
'constvel'	posRMSE, velRMSE, posANEES, velANEES.
'constacc'	posRMSE, velRMSE, accRMSE, posANEES, velANEES, accANEES
'constturn'	posRMSE, velRMSE, yawRateRMSE, posANEES, velANEES, yawRateANEES

RMSE and ANEES denote root mean squared error and average normalized estimation error squared between a truth trajectory and its associated tracks at the current time step. Note that a truth trajectory can associate with multiple tracks. For example, the position RMSE and ANEES values for a truth are defined respectively as:

$$\begin{aligned} posRMSE &= \sqrt{\frac{1}{K} \sum_{k=1}^K \|\Delta p_k\|} \\ posANEEs &= \frac{1}{K} \Delta p_k^T C_k^{-1} \Delta p_k \end{aligned}$$

where  $K$  is the total number of tracks associated with the truth,  $C_k$  is the position covariance of the  $k$ th track at the current time step, and

$$\Delta p_k = p_{track,k} - p_{truth}$$

is the position error between the  $k$ th associated track and the truth. The RMSE and ANEES values for other states (vel, pos, acc, and yawRate) are defined similarly.

- When you set the `ErrorFunctionFormat` property to 'custom', the table contains the arithmetically averaged values of the custom metrics output from the error function.

#### **Introduced in R2018b**

# trackFuser

Single-hypothesis track-to-track fuser

## Description

The `trackFuser` System object fuses tracks generated by tracking sensors or trackers and architect decentralized tracking systems. `trackFuser` uses the global nearest neighbor (GNN) algorithm to maintain a single hypothesis about the objects it tracks. The input tracks are called source or local tracks, and the output tracks are called central tracks.

To fuse tracks using this object:

- 1 Create the `trackFuser` 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

```
fuser = trackFuser  
fuser = trackFuser(Name,Value)
```

#### Description

`fuser = trackFuser` creates a track-to-track fuser that uses the global nearest neighbor (GNN) algorithm to maintain a single hypothesis about the objects it tracks.

`fuser = trackFuser(Name,Value)` sets properties for the fuser using one or more name-value pairs. Unspecified properties have default values. 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](#).

### FuserIndex — Unique index for track fuser

1 (default) | positive integer

Unique index for the fuser, specified as a positive integer. Use this property to distinguish different fuslers in a multiple-fuser environment.

Example: 2

**MaxNumSources — Maximum number of source configurations**

20 (default) | positive integer

Maximum number of source configurations that the fuser can maintain, specified as a positive integer.

Example: 200

**SourceConfigurations — Configurations of source systems**

cell array of fuserSourceConfiguration objects

Configurations of source systems, specified as a cell array of fuserSourceConfiguration objects. The default value is a 1-by- $N$  cell array of fuserSourceConfiguration objects, where  $N$  is the value of the MaxNumSources property. You can specify this property during creation as a Name-Value pair or specify it after creation.

Data Types: object

**Assignment — Assignment algorithm**

'MatchPairs' (default) | 'Munkres' | 'Jonker-Volgenant' | 'Auction' | 'Custom'

Assignment algorithm, specified as 'MatchPairs', 'Munkres', 'Jonker-Volgenant', 'Auction', or 'Custom'. Munkres is the only assignment algorithm that guarantees an optimal solution, but it is also the slowest, especially for large numbers of detections and tracks. The other algorithms do not guarantee an optimal solution but can be faster for problems with 20 or more tracks and detections. Use 'Custom' to define your own assignment function and specify its name in the CustomAssignmentFcn property.

Data Types: char

**CustomAssignmentFcn — Custom assignment function**

function handle

Custom assignment function, specified as a function handle. An assignment function must have the following syntax:

```
[assignments,unassignedCentral,unassignedLocal] = f(cost,costNonAssignment)
```

For an example of assignment function and a description of its arguments, see `assignmunkres`.

**Dependencies**

To enable this property, set the Assignment property to 'Custom'.

Data Types: function handle | string | char

**AssignmentThreshold — Track-to-track assignment threshold**

30\*[1 Inf] (default) | positive scalar | 1-by-2 vector of positive values

Track-to-track assignment threshold, specified as a positive scalar or a 1-by-2 vector of  $[C_1, C_2]$ , where  $C_1 \leq C_2$ . If specified as a scalar, the specified value,  $val$ , is expanded to  $[val, Inf]$ .

Initially, the fuser executes a coarse estimation for the normalized distance between all the local and central tracks. The fuser only calculates the accurate normalized distance for the local and central combinations whose coarse normalized distance is less than  $C_2$ . Also, the fuser can only assign a local track to a central track if their accurate normalized distance is less than  $C_1$ . See the `distance`

function used with tracking filters (`trackingCKF` and `trackingEKF` for example) for an explanation of the distance calculation.

Tips:

- Increase the value of  $C_2$  if there are combinations of local and central tracks that should be calculated for assignment but are not. Decrease it if the calculation takes too much time.
- Increase the value of  $C_1$  if there are local tracks that should be assigned to central tracks but are not. Decrease it if there are local tracks that are assigned to central tracks they should not be assigned to (too far away).

#### **StateTransitionFcn — State transition function**

'constvel' (default) | function handle

State transition function, specified as a function handle. This function calculates the state at time step  $k$  based on the state at time step  $k-1$ .

- If `HasAdditiveProcessNoise` is `true`, the function must use the following syntax:

`x(k) = f(x(k-1),dt)`

where:

- $x(k)$  — The (estimated) state at time  $k$ , specified as a vector or a matrix. If specified as a matrix, then each column of the matrix represents one state vector.
- $dt$  — The time step for prediction.
- If `HasAdditiveProcessNoise` is `false`, the function must use this syntax:

`x(k) = f(x(k-1),w(k-1),dt)`

where:

- $x(k)$  — The (estimated) state at time  $k$ , specified as a vector or a matrix. If specified as a matrix, then each column of the matrix represents one state vector.
- $w(k)$  — The process noise at time  $k$ .
- $dt$  — The time step for prediction.

Example: `@constacc`

Data Types: `function_handle` | `char` | `string`

#### **StateTransitionJacobianFcn — Jacobian of state transition function**

' ' (default) | function handle

Jacobian of the state transition function, specified as a function handle. If not specified, the Jacobian is numerically computed, which may increase processing time and numerical inaccuracy. If specified, the function must support one of these two syntaxes:

- If `HasAdditiveProcessNoise` is `true`, the function must use this syntax:

`Jx(k) = statejacobianfcn(x(k),dt)`

where:

- $x(k)$  — The (estimated) state at time  $k$ , specified as an  $M$ -by-1 vector of real values.

- $dt$  — The time step for prediction.
- $Jx(k)$  — The Jacobian of the state transition function with respect to the state,  $df/dx$ , evaluated at  $x(k)$ . The Jacobian is returned as an  $M$ -by- $M$  matrix.
- If `HasAdditiveProcessNoise` is `false`, the function must use this syntax:

```
[Jx(k), Jw(k)] = statejacobianfcn(x(k), w(k), dt)
```

where:

- $x(k)$  — The (estimated) state at time  $k$ , specified as an  $M$ -by-1 vector of real values.
- $w(k)$  — The process noise at time  $k$ , specified as a  $W$ -by-1 vector of real values.
- $dt$  — The time step for prediction.
- $Jx(k)$  — The Jacobian of the state transition function with respect to the state,  $df/dx$ , evaluated at  $x(k)$ . The Jacobian is returned as an  $M$ -by- $M$  matrix.
- $Jw(k)$  — The Jacobian of the state transition function with respect to the process noise,  $df/dw$ , evaluated at  $x(k)$  and  $w(k)$ . The Jacobian is returned as an  $M$ -by- $W$  matrix.

Example: `@constaccjac`

Data Types: `function_handle` | `char` | `string`

**ProcessNoise — Process noise covariance**

`eye(3)` (default) | positive real scalar | positive definite matrix

Process noise covariance, specified as a positive real scalar or a positive definite matrix.

- When `HasAdditiveProcessNoise` is `true`, specify the process noise covariance as a positive real scalar or a positive definite  $M$ -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  $W$ -by- $W$  matrix.  $W$  is the dimension of the process noise vector.

Example: `[1.0 0.05; 0.05 2]`

Data Types: `single` | `double`

**HasAdditiveProcessNoise — Model additive process noise**

`false` (default) | `true`

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.

**StateParameters — Parameters of 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 specify parameters about the fused central tracks that the track fuser outputs. For example, you can use these parameters to perform a coordinate transformation from a vehicle local frame to another vehicle local frame or a global frame.

**Tunable:** Yes

Data Types: `struct`

**ConfirmationThreshold — Threshold for central track confirmation**

[2 3] (default) | positive integer | 1-by-2 vector of positive integers

Threshold for central track confirmation, specified as a positive integer  $M$ , or a 1-by-2 vector of positive integers  $[M \ N]$  with  $M \leq N$ . A central track is confirmed if it is assigned to local tracks at least  $M$  times in the last  $N$  updates. If specified a positive integer  $M$ , the confirmation threshold is expanded to  $[M, M]$ .

Data Types: single | double

**DeletionThreshold — Threshold for central track deletion**

[5 5] (default) | positive integer | 1-by-2 vector of positive integers

Threshold for central track deletion, specified as a positive integer  $P$ , or a 1-by-2 vector of positive integers  $[P \ R]$  with  $P \leq R$ . A central track is deleted if the track is not assigned to local tracks at least  $P$  times in the last  $R$  updates. If specified a positive integer  $P$ , the confirmation threshold is expanded to  $[P, P]$ .

Example: [5 6]

Data Types: single | double

**FuseConfirmedOnly — Fuse only confirmed local tracks**

true (default) | false

Fuse only confirmed local tracks, specified as `false` or `true`. Set this property to `false` if you want to fuse all local tracks regardless of their confirmation status.

Data Types: logical

**FuseCoasted — Fuse coasted local tracks**

false (default) | true

Fuse coasted local tracks, specified as `true` or `false`. Set this property to `true` if you want to fuse coasted local tracks (`IsCoasted` field or property of the `localTracks` input is `true`). Set it to `false` if you want to only fuse local tracks that are not coasted.

Example: true

Data Types: logical

**StateFusion — State fusion algorithm**

'Cross' (default) | 'Intersection' | 'Custom'

State fusion algorithm, specified as:

- 'Cross' — Uses the cross-covariance fusion algorithm
- 'Intersection' — Uses the covariance intersection fusion algorithm
- 'Custom' — Allows you to specify a customized fusion function

Use the `StateFusionParameters` property to specify additional parameters used by the state fusion algorithm.

Data Types: char

**CustomStateFusionFcn — Custom state fusion function**

' ' (default) | function handle

Custom state fusion function, specified as a function handle. The state fusion function must support one of the following syntaxes:

```
[fusedState,fusedCov] = f(trackState,trackCov)  
[fusedState,fusedCov] = f(trackState,trackCov,fuseParams)
```

where:

- `trackState` is specified as an  $N$ -by- $M$  matrix.  $N$  is the dimension of the track state, and  $M$  is the number of tracks.
- `trackCov` is specified as an  $N$ -by- $N \cdot M$  matrix.  $N$  is the dimension of the track state, and  $M$  is the number of tracks.
- `fuseParams` is optional parameters defined in the `StateFusionParameters` property.
- `fusedState` is returned as an  $N$ -by-1 vector.
- `fusedCov` is returned as an  $N$ -by- $N$  matrix.

#### Dependencies

To enable this property, set the `StateFusion` property to 'Custom'.

Data Types: `function_handle | char | string`

#### **StateFusionParameters — Parameters for state fusion function**

[ ] (default)

Parameters for state fusion function. Depending on the choice of `StateFusion` algorithm, you can specify `StateFusionParameters` as:

- If `StateFusion` is 'Cross', specify it as a scalar in (0,1). See `fusexcov` for more details.
- If `StateFusion` is 'Intersection', specify it as 'det' or 'trace'. See `fusecovint` for more details.
- If `StateFusion` is 'Custom', you can specify these parameters in any variable type, as long as they match the setup of the optional `fuseParams` input of the custom state fusion function specified in the `CustomStateFusionFcn` property.

By default, the property is empty.

#### **NumCentralTracks — Number of central-level tracks**

nonnegative integer

This property is read-only.

Number of central tracks currently maintained by the fuser, returned as a nonnegative integer.

Data Types: `double`

#### **NumConfirmedCentralTracks — Number of confirmed central tracks**

nonnegative integer

This property is read-only.

Number of confirmed central tracks currently maintained by the fuser, returned as a nonnegative integer.

Data Types: `double`

## Usage

### Syntax

```
confirmedTracks = fuser(localTracks,tFusion)
[confirmedTracks,tentativeTracks,allTracks,analysisInformation] = fuser(
    localTracks,tFusion)
```

### Description

`confirmedTracks = fuser(localTracks,tFusion)` returns a list of confirmed tracks from a list of local tracks. Confirmed tracks are predicted to the update time, `tFusion`.

`[confirmedTracks,tentativeTracks,allTracks,analysisInformation] = fuser(
 localTracks,tFusion)` also returns a list of tentative tracks, a list of all tracks, and the analysis information.

### Input Arguments

#### **localTracks — Local tracks**

array of `objectTrack` objects | array of structures

Local tracks, specified as an array of `objectTrack` objects, or an array of structures with field names that match the property names of an `objectTrack` object. Local tracks are tracks generated from trackers in a source track system.

Data Types: `object` | `struct`

#### **tFusion — Update time**

scalar

Update time, specified as a scalar. The fuser predicts all central tracks to this time. Units are in seconds.

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`

### **analysisInformation — Additional information for analyzing track updates**

structure

Additional information for analyzing track updates, returned as a structure. The fields of this structure are:

Field	Description
<code>TrackIDsAtStepBeginning</code>	Track IDs when the step began
<code>CostMatrix</code>	Cost of assignment matrix
<code>Assignments</code>	Assignments returned from the assignment function
<code>UnassignedCentralTracks</code>	IDs of unassigned central tracks
<code>UnassignedLocalTracks</code>	IDs of unassigned local tracks
<code>NonInitializingLocalTracks</code>	IDs of local tracks that were unassigned but were not used to initialize a central track
<code>InitiatedCentralTrackIDs</code>	IDs of central tracks initiated during the step
<code>UpdatedCentralTrackIDs</code>	IDs of central tracks updated during the step
<code>DeletedTrackIDs</code>	IDs of central tracks deleted during the step
<code>TrackIDsAtStepEnd</code>	IDs of central tracks when the step ended

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 trackFuser

<code>predictTrackToTime</code>	Predict track state
<code>initializeTrack</code>	Initialize new track
<code>deleteTrack</code>	Delete existing track

## 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
<code>clone</code>	Create duplicate System object
<code>isLocked</code>	Determine if System object is in use
<code>reset</code>	Reset internal states of System object

## Examples

### Fuse Tracks from Two Sources Using trackFuser

Define two tracking sources: one internal and one external. The `SourceIndex` of each source must be unique.

```
internalSource = fuserSourceConfiguration(1,'IsInternalSource',true);
externalSource = fuserSourceConfiguration(2,'IsInternalSource',false);
```

Create a `trackFuser` with `FuserIndex` equal to 3. The fuser takes the two sources defined above and uses the '`Cross`' `StateFusion` model.

```
fuser = trackFuser('FuserIndex',3, 'MaxNumSources',2, ...
    'SourceConfigurations',{internalSource;externalSource}, ...
    'StateFusion','Cross');
```

Update the fuser with two tracks from the two sources. Use a 3-D constant velocity state, in which the states are given in the order of [x; vx; y; vy; z; vz]. The states of the two tracks are the same, but their covariances are different. For the first track, create a large covariance in the x-axis. For the second track, create a large covariance in the y-axis.

```
tracks = [objectTrack('SourceIndex',1,'State',[10;0;0;0;0;0], ...
    'StateCovariance',diag([100,1000,1,10,1,10])); ...
    objectTrack('SourceIndex',2,'State',[10;0;0;0;0;0], ...
    'StateCovariance',diag([1,10,100,1000,1,10))];
```

Fuse the track with fusion time equal to 0.

```
time = 0;
confirmedTracks = fuser(tracks,time);
```

Obtain the positions and position covariances of the source tracks and confirmed tracks.

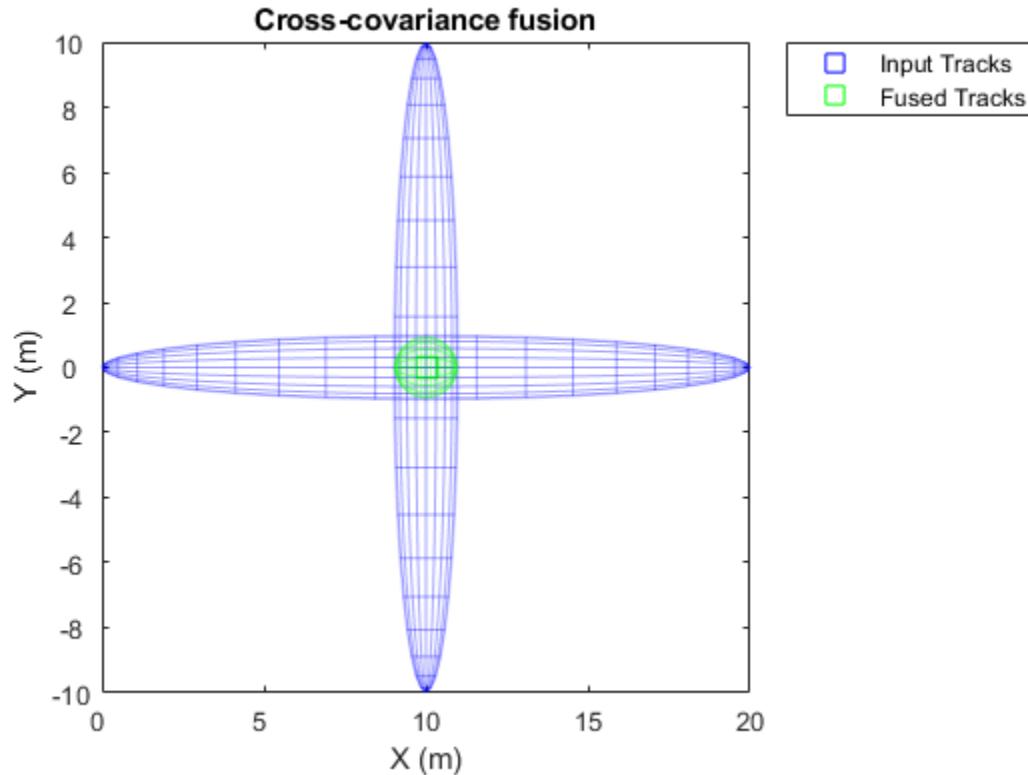
```
positionSelector = [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 1 0]; % [x; y; z]
[inputPos,inputCov] = getTrackPositions(tracks,positionSelector);
[outputPos,outputCov] = getTrackPositions(confirmedTracks,positionSelector);
```

Visualize the results using `trackPlotter`.

```
tPlotter = theaterPlot('XLim',[0, 20], 'YLim',[-10, 10], 'ZLim',[-10, 10]);
tPlotter1 = trackPlotter(tPlotter, 'DisplayName', 'Input Tracks', 'MarkerEdgeColor', 'blue');
```

```
tPlotter2 = trackPlotter(tPlotter, 'DisplayName', 'Fused Tracks', 'MarkerEdgeColor', 'green');

plotTrack(tPlotter1, inputPos, inputCov)
plotTrack(tPlotter2, outputPos, outputCov)
title('Cross-covariance fusion')
```



## References

- [1] Blackman, S. and Popoli, R., 1999. *Design and analysis of modern tracking systems*(Book). Norwood, MA: Artech House, 1999.
- [2] Chong, Chee-Yee, Shozo Mori, William H. Barker, and Kuo-Chu Chang. "Architectures and algorithms for track association and fusion." IEEE Aerospace and Electronic Systems Magazine 15, no. 1 (2000): 5-13.
- [3] Tian, Xin, Yaakov Bar-Shalom, D. Choukroun, Y. Oshman, J. Thienel, and M. Idan. "Track-to-Track Fusion Architectures-A Review." In book "Advances in Estimation, Navigation, and Spacecraft Control". Springer, 2015.

## 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).
- You must specify the `MaxNumSources` property during construction. Also, the property is read only for code generation.
- You must specify the `SourceConfigurations` property for all the sources during construction. Additionally,
  - All elements of `SourceConfigurations` must use the same `LocalToCentralTransformFcn`.
  - All elements of `SourceConfigurations` must use the same `CentralToLocalTransformFcn`.
- The input tracks must be a struct array instead of an `objectTrack` object array.
- The track outputs (all three) are each a struct array instead of an `objectTrack` object array.
- The `ObjectAttributes` structure for all the input source tracks must be in the same format (same field names and data types).
- The `StateParameters` structure for all the input source tracks must be in the same format (same field names and data types) as the `StateParameters` structure of the track fuser.

## See Also

`objectTrack` | `trackerGNN` | `trackerJPDA` | `trackerPHD` | `trackerTOMHT`

## External Websites

“Introduction to Track-To-Track Fusion”

## Introduced in R2019b

# deleteTrack

Delete existing track

## Syntax

```
deleted = deleteTrack(obj,trackID)
```

## Description

`deleted = deleteTrack(obj,trackID)` deletes the track specified by `trackID` in the tracker or track fuser object, `obj`.

## Examples

### Delete track

Create a track using detections in a GNN tracker.

```
tracker = trackerGNN;
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

### obj — Tracker or fuser object

trackerTOMHT object | trackerJPDA object | trackerGNN object | trackerPHD object | trackFuser object

Tracker or fuser object, specified as a trackerTOMHT, trackerJPDA, trackerGNN, trackerPHD, or trackFuser 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

trackFuser | trackerGNN | trackerJPDA | trackerPHD | trackerTOMHT

## Introduced in R2020a

# initializeTrack

Initialize new track

## Syntax

```
trackID = initializeTrack(obj,track)
trackID = initializeTrack(obj,track,filter)
```

## Description

`trackID = initializeTrack(obj,track)` initializes a new track in the tracker or track fuser object, `obj`. The tracker or fuser must be updated at least once before initializing a track. If the track is initialized successfully, the tracker or fuser 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 or track fuser 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.

---

**Note** This syntax doesn't support using the `trackingGSF`, `trackingPF`, or `trackingIMM` filter object as the internal tracking filter for the tracker or track fuser. Use the second syntax in these cases.

---

`trackID = initializeTrack(obj,track,filter)` initializes a new track in the tracker or track fuser object, `obj`, using a specified tracking filter, `filter`.

---

## Note

- If the tracking filter used in the tracker or track fuser is `trackingGSF`, `trackingPF`, or `trackingIMM`, you must use this syntax instead of the first syntax.
  - This syntax does not support using `trackFuser` as the `obj` input.
- 

## Examples

### Initialize Track

Create a GNN tracker and update the tracker with detections at  $t = 0$  and  $t = 1$  second.

```
tracker = trackerGNN
tracker =
    trackerGNN with properties:
        TrackerIndex: 0
```

```

FilterInitializationFcn: 'initcvekf'
    Assignment: 'MatchPairs'
    AssignmentThreshold: [30 Inf]
        MaxNumTracks: 100
        MaxNumSensors: 20

        TrackLogic: 'History'
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 =
    trackerGNN with properties:

        TrackerIndex: 0
    FilterInitializationFcn: 'initcvekf'
        Assignment: 'MatchPairs'
        AssignmentThreshold: [30 Inf]
            MaxNumTracks: 100
            MaxNumSensors: 20

        TrackLogic: 'History'
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 =
    trackerGNN with properties:

    TrackerIndex: 0
    FilterInitializationFcn: 'initcvekf'
        Assignment: 'MatchPairs'
    AssignmentThreshold: [30 Inf]
        MaxNumTracks: 100
        MaxNumSensors: 20

        TrackLogic: 'History'
    ConfirmationThreshold: [2 3]
        DeletionThreshold: [5 5]

        HasCostMatrixInput: false
    HasDetectableTrackIDsInput: false
        StateParameters: [1x1 struct]

    NumTracks: 2
    NumConfirmedTracks: 2
```

## Input Arguments

### **obj — Tracker or fuser object**

`trackerTOMHT` object | `trackerJPDA` object | `trackerGNN` object | `trackFuser` object

Tracker or fuser object, specified as a `trackerTOMHT`, `trackerJPDA`, `trackerGNN`, or `trackFuser` 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` | `trackingABF` | `trackingCKF` | `trackingMSCEKF` |  
`trackingPF` | `trackingIMM` | `trackingGSF`

Filter object, specified as a `trackingKF`, `trackingEKF`, `trackingUKF`, `trackingABF`,  
`trackingCKF`, `trackingIMM`, `trackingGSF`, `trackingPF`, or `trackingMSCEKF` 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

`trackFuser` | `trackerGNN` | `trackerJPDA` | `trackerTOMHT`

## Introduced in R2020a

# trackOSPA Metric

Optimal subpattern assignment (OSPA) metric

## Description

`trackOSPA Metric` System object computes the optimal subpattern assignment metric between a set of tracks and the known truths. An OSPA metric contains three components:

- Localization error component — Accounts for state estimation errors between assigned tracks and truths
- Cardinality error component— Accounts for the number of unassigned tracks and truths
- Labelling error component — Accounts for the error of incorrect assignment

For more details, see “OSPA Metric” on page 3-362 and [2].

To use `trackOSPA Metric`:

- 1 Create the `trackOSPA Metric` 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

```
OSPA Metric = trackOSPA Metric  
OSPA Metric = trackOSPA Metric(Name, Value)
```

#### Description

`OSPA Metric = trackOSPA Metric` creates a `trackOSPA Metric` System object, `OSPA Metric`, with default property values.

`OSPA Metric = trackOSPA Metric(Name, Value)` sets properties for the `trackOSPA Metric` object using one or more name-value pairs. For example, `OSPA Metric = trackOSPA Metric('CutoffDistance', 5)` creates a `trackOSPA Metric` object with the cut off distance equal to 5. Enclose property names 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.

**CutoffDistance — Threshold for cutoff distance between track and truth**

30 (default) | real positive scalar

Threshold for cutoff distance between track and truth, specified as a real positive scalar. If the computed distance between a track and the assigned truth is higher than the threshold, the actual distance incorporated in the metric is reduced to the threshold.

Example: 40

Data Types: single | double

**Order — Order of OSPA metric**

2 (default) | positive integer

Order of OSPA metric, specified as a positive integer.

Example: 10

Data Types: single | double

**LabelingError — Penalty for incorrect assignment**

0 (default) | real scalar in [0, CutoffDistance]

Penalty for incorrect assignment of track to truth, specified as a real positive scalar. The function decides if an assignment is correct based on the provided known `assignment` input. If the assignment is not provided as an input, the last known "optimal" assignment is assumed to be correct.

Example: 5

Data Types: single | double

**Distance — Distance type**

'posnees' (default) | 'velnees' | 'posabserr' | 'velabserr'

Distance type, specified as 'posnees', 'velnees', 'posabserr', or 'velabserr'. The distance type specifies the physical quantity used for distance calculations:

- 'posnees' - Normalized estimation error squared (NEES) of track position
- 'velnees' - NEES error of track velocity
- 'posabserr' - Absolute error of track position
- 'velabserr' - Absolute error of track velocity
- 'custom' - Custom distance error

If you specify the `Distance` property as 'custom', you must also specify the distance function in the `DistanceFcn` property.

**DistanceFcn — Custom distance function**

function handle

Custom distance function, specified as a function handle. The function must support the following syntax:

```
d = myCustomFcn(Track,Truth)
```

where `Track` is a structure or an object of track information, `Truth` is a structure or an object of truth information, and `d` is the distance between the truth and the track. See `objectTrack` for an example on how to organize track information.

Example: `@myCustomFcn`

#### Dependencies

To enable this property, set the `Distance` property to '`custom`'.

#### MotionModel — Desired platform motion model

`'constvel'` (default) | `'constacc'` | `'constturn'` | `'singer'`

Desired platform motion model, specified as '`constvel`', '`constacc`', '`constturn`', or '`singer`'. This property selects the motion model used by the `tracks` input.

The motion models expect the '`State`' field of the `tracks` to have a column vector containing these values:

- '`constvel`' — Position is in elements [1 3 5], and velocity is in elements [2 4 6].
- '`constacc`' — Position is in elements [1 4 7], velocity is in elements [2 5 8], and acceleration is in elements [3 6 9].
- '`constturn`' — Position is in elements [1 3 6], velocity is in elements [2 4 7], and yaw rate is in element 5.
- '`singer`' — Position is in elements [1 4 7], velocity is in elements [2 5 8], and acceleration is in elements [3 6 9].

The '`StateCovariance`' field of the `tracks` input must have position, velocity, and turn-rate covariances in the rows and columns corresponding to the position, velocity, and turn-rate of the '`State`' field of the `tracks` input.

#### TrackIdentifierFcn — Track identifier function

`@defaultTrackIdentifier` (default) | function handle

Track identifier function, specified as a function handle. The function extracts track ID from the track input. The function must support the following syntax:

```
Trackids = trackIdentifier(Tracks)
```

where `Tracks` is an array of structures or objects containing the information of tracks, and `Trackids` is a numeric array of the same size as `Tracks`. For an example of track object, see `objectTrack`. For the default identifier function, `defaultTrackIdentifier`, the track ID must be contained in `Tracks` as the value of the `TrackID` field or property.

Example: `@myTrackIdentifier`

#### TruthIdentifierFcn — Truth identifier function

`@defaultTruthIdentifier` (default) | function handle

Truth identifier function, specified as a function handle. The function extracts truth ID from truth input. The function must support the following syntax:

```
TruthIDs = truthIdentifier(Truths)
```

where `Truths` is an array of structures or objects containing the information of truths, and `TruthIDs` is a numeric array of the same size as `Truths`. For the use of the default identifier

function, `defaultTruthIdentifier`, the truth ID must be contained in `Truth` as a value of the `PlatformID` field or property.

Example: `@myTruthIdentifier`

#### **HasAssignmentInput — Enable assignment input**

`false` (default) | `true`

Enable assignment input, specified as `true` or `false`.

Data Types: `logical`

## **Usage**

### **Syntax**

```
OSPA = OSPAMetric(tracks, truths)
OSPA = OSPAMetric(tracks, truths, assignment)
[OSPA, localOSPA] = OSPAMetric(__)
[OSPA, localOSPA, cardOSPA] = OSPAMetric(__)
[OSPA, localOSPA, cardOSPA, labelOSPA] = OSPAMetric(__)
```

### **Description**

`OSPA = OSPAMetric(tracks, truths)` returns the OSPA metric between the set of tracks and truths.

`OSPA = OSPAMetric(tracks, truths, assignment)` allows you to specify the known assignment between tracks and truths at the current time step. To use this syntax, specify the `HasAssignmentInput` property as `true`.

`[OSPA, localOSPA] = OSPAMetric(__)` also returns the localization error component of the OSPA metric. You can use any of the input combinations in the previous syntaxes as the input.

`[OSPA, localOSPA, cardOSPA] = OSPAMetric(__)` also returns the cardinality error component of the OSPA metric.

`[OSPA, localOSPA, cardOSPA, labelOSPA] = OSPAMetric(__)` also returns the labeling error component of the OSPA metric.

### **Input Arguments**

#### **tracks — Track information**

`array of structures` | `array of objects`

Track information, specified as an array of structures or objects for noncustomized (built-in) distance functions. Each structure or object must contain `State` as a field or property. Additionally, if an NEES-based distance (`posnees` or `velnees`) is specified in the `Distance` property, each structure or object must also contain `StateCovariance` as a field or property. Moreover, if the default track identifier function is used in the `TrackIdentifierFcn` property, then each structure or object must also contain `TrackID` as a field or property.

Data Types: `struct` | `object`

**truths — Truth information**

array of structures | array of objects

Truth information, specified as an array of structures or objects for noncustomized (built-in) distance functions. Each structure or object must contain **Position** and **velocity** as fields or properties. If the default truth identifier function is used in the **TruthIdentifierFcn** property, then each structure or object must also contain **PlatformID** as a field or property.

Data Types: `struct` | `object`**assignment — Known assignment**`N`-by-2 matrix of nonnegative integers

Known assignment, specified as an  $N$ -by-2 matrix of nonnegative integers. The first column elements are track IDs, and the second column elements are truth IDs. The IDs in the same row are assigned to each other. If a track or truth is not assigned, specify 0 as the same row element.

Note that the assignment must be a unique assignment between tracks and truths. Redundant or false tracks should be treated as unassigned tracks by assigning them to the "0" **TruthID**.

Data Types: `single` | `double`**Output Arguments****OSPA — OSPA metric**

nonnegative real scalar

OSPA metric, returned as a nonnegative real scalar.

Example: 10.1

**localOSPA — Localization error component**

nonnegative real scalar

Localization error component, returned as a nonnegative real scalar.

Example: 8.5

**cardOSPA — Cardinality error component**

nonnegative real scalar

Cardinality error component, returned as a nonnegative real scalar.

Example: 6

**labelOSPA — Labeling error component**

nonnegative real scalar

Labeling error component, returned as a nonnegative real scalar.

Example: 7.5

## 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
clone	Create duplicate System object

## Examples

### Evaluate Tracking Result Using trackOSPAmetric

Load prerecorded track data and truth data.

```
load trackmetricex tracklog truthlog
```

Construct a `trackOSPAmetric` object.

```
tom = trackOSPAmetric;
```

Initialize output variables.

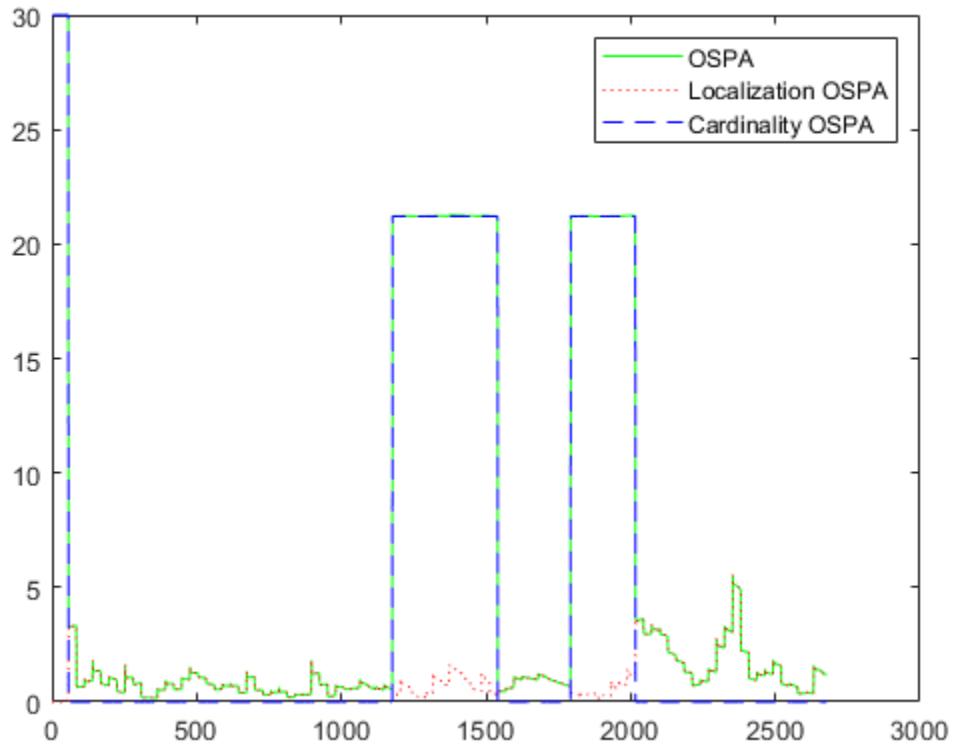
```
ospa = zeros(numel(tracklog),1);
card0spa = zeros(numel(tracklog),1);
loc0spa = zeros(numel(tracklog),1);
```

Calculate three OSPA components in a loop.

```
for i = 1:numel(tracklog)
    tracks = tracklog{i};
    truths = truthlog{i};
    [ospa(i), loc0spa(i), card0spa(i)] = tom(tracks, truths);
end
```

Visualize the results.

```
figure()
plot(ospa,'g');
hold on;
plot(loc0spa,'r:');
plot(card0spa,'b--');
legend('OSPA','Localization OSPA','Cardinality OSPA');
```



## More About

### OSPA Metric

At time  $t_k$ , a list of truths is:

$$X = [x_1, x_2, \dots, x_m]$$

At the same time, a tracker obtains a list of tracks:

$$Y = [y_1, y_2, \dots, y_n]$$

The OSPA metric (used to evaluate the tracking performance based on the truth) is:

$$OSPA = (d_{loc}^p + d_{card}^p + d_{lab}^p)^{1/p}$$

Assuming  $m \leq n$ , the three components ( $d_{loc}$ ,  $d_{card}$ , and  $d_{lab}$ ) are calculated as following. The localization error component  $d_{loc}$  is computed as:

$$d_{loc} = \left\{ \frac{1}{n} \sum_{i=1}^m d_c^p(x_i, y_{\Pi(i)}) \right\}^{1/p}$$

where  $p$  is the order of the OSPA metric,  $d_c$  is the cutoff-based distance, and  $y_{\Pi(i)}$  represents the track assigned to truth  $x_i$ . The cutoff-based distance  $d_c$  is defined as:

$$d_c(x, y) = \min\{d_b(x, y), c\}$$

where  $c$  is the cutoff distance threshold, and  $d_b(x, y)$  is the base distance between track  $x$  and truth  $y$  calculated by a distance function specified by the `Distance` property. The cutoff based distance  $d_c$  takes the smaller value of  $d_b$  and  $c$ .

The cardinality error component  $d_{card}$  is:

$$d_{card} = \left\{ \frac{n - m}{n} c^p \right\}^{1/p}$$

The labelling error component  $d_{lab}$  is:

$$d_{lab} = \left\{ \frac{1}{n} \sum_{i=1}^m \alpha^p \gamma(L(x_i), L(y_{\Pi(i)})) \right\}^{1/p}$$

where  $\alpha$  is the penalty for incorrect assignment defined by the `LabelingError` property,  $L(x_i)$  represents the truth ID of  $x_i$ , and  $L(y_{\Pi(i)})$  represents the track ID of  $y_{\Pi(i)}$ . The function  $\gamma = 1$  if the IDs of the truth and track pair ( $L(x_i)$  and  $L(y_{\Pi(i)})$ ) agree with the known assignment given by the `assignment` input, or agree with the assignment in last update if the known assignment is not given. Otherwise,  $\gamma = 0$ .

If  $m > n$ , simply exchange  $m$  and  $n$  in the formulation to obtain the OSPA metric.

## References

- [1] Schuhmacher, B., B. -T. Vo, and B. -N. Vo. "A Consistent Metric for Performance Evaluation of Multi-Object Filters." *IEEE Transactions on Signal Processing*, Vol. 56, No. 8, pp. 3447–3457, 2008.
- [2] Ristic, B., B. -N. Vo, D. Clark, and B. -T. Vo. "A Metric for Performance Evaluation of Multi-Target Tracking Algorithms." *IEEE Transactions on Signal Processing*, Vol. 59, No. 7, pp. 3452–3457, 2011.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[trackAssignmentMetrics](#) | [trackErrorMetrics](#)

### Introduced in R2019b

# trackGOSPAMetric

Generalized optimal subpattern assignment (GOSPA) metric

## Description

`trackGOSPAMetric` System object computes the generalized optimal subpattern assignment metric between a set of tracks and the known truths.

For more details, see “GOSPA Metric” on page 3-371 and [1].

To compute the generalized subpattern alignment metric:

- 1 Create the `trackGOSPAMetric` 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

```
GOSPAmetric = trackGOSPAmetric
GOSPAmetric = trackGOSPAmetric(Name,Value)
```

#### Description

`GOSPAmetric = trackGOSPAmetric` creates a `trackGOSPAmetric` System object with default property values.

`GOSPAmetric = trackGOSPAmetric(Name,Value)` sets properties for the `trackGOSPAmetric` object using one or more name-value pairs. For example, `GOSPAmetric = trackGOSPAmetric('CutoffDistance',5)` creates a `trackGOSPAmetric` object with the cutoff distance equal to 5. Enclose property names 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](#).

#### CutoffDistance — Threshold for cutoff distance between track and truth

30 (default) | real positive scalar

Threshold for cutoff distance between track and truth, specified as a real positive scalar. A truth is assigned to a track only if the distance between the track and the known truth is less than this distance.

Example: 40

Data Types: single | double

**Order — Order of GOSPA metric**

2 (default) | positive integer

Order of GOSPA metric, specified as a positive integer.

Example: 1

Data Types: single | double

**Alpha — Alpha parameter of GOSPA metric**

2 (default) | positive scalar in range [0, 2]

Alpha parameter of GOSPA metric, specified as a positive scalar in the range [0, 2].

Example: 1

Data Types: single | double

**SwitchingPenalty — Penalty for assignment switching**

0 | nonnegative real scalar

Penalty for assignment switching, specified as a nonnegative real scalar.

Example: 1.2

**Distance — Distance type**

'posnees' (default) | 'velnees' | 'posabserr' | 'velabserr' | 'custom'

Distance type, specified as 'posnees', 'velnees', 'posabserr', 'velabserr', or 'custom'. This property specifies the physical quantity used for distance calculations:

- 'posnees' - Normalized estimation error squared (NEES) of track position
- 'velnees' - NEES error of track velocity
- 'posabserr' - Absolute error of track position
- 'velabserr' - Absolute error of track velocity
- 'custom' - Custom distance error

If you specify the `Distance` property as 'custom', you must also specify the distance function in the `DistanceFcn` property.

**DistanceFcn — Custom distance function**

function handle

Custom distance function, specified as a function handle. The function must support this syntax:

```
d = myCustomFcn(track,truth)
```

where `track` is a structure or an object of track information, `truth` is a structure or an object of truth information, and `d` is the distance between the truth and the track. See `objectTrack` for an example on how to organize information for estimated tracks and truth tracks.

Example: `@myCustomFcn`

### Dependencies

To enable this property, set the `Distance` property to 'custom'.

#### **MotionModel — Desired platform motion model**

`'constvel'` (default) | `'constacc'` | `'constturn'` | `'singer'`

Desired platform motion model, specified as '`constvel`', '`constacc`', '`constturn`', or '`singer`'. This property selects the motion model used by the `tracks` input.

The motion models expect the '`State`' field of the `tracks` input to have a column vector containing these values:

- '`constvel`' — Constant velocity motion model of the form  $[x;vx;y;vy;z;vz]$ , where  $x$ ,  $y$ , and  $z$  are position coordinates and  $vx$ ,  $vy$ ,  $vz$  are velocity coordinates.
- '`constacc`' — Constant acceleration motion model of the form  $[x;vx;ax;y;vy;ay;z;vz;az]$ , where  $x$ ,  $y$ , and  $z$  are position coordinates,  $vx$ ,  $vy$ ,  $vz$  are velocity coordinates, and  $ax$ ,  $ay$ ,  $az$  are acceleration coordinates.
- '`constturn`' — Constant turn motion model of the form  $[x;vx;y;vy;theta;z;vz]$ , where  $x$ ,  $y$ , and  $z$  are position coordinates,  $vx$ ,  $vy$ ,  $vz$  are velocity coordinates, and  $theta$  is the yaw rate.
- '`singer`' — Singer acceleration motion model of the form  $[x;vx;ax;y;vy;ay;z;vz;az]$ , where  $x$ ,  $y$ , and  $z$  are position coordinates,  $vx$ ,  $vy$ ,  $vz$  are velocity coordinates, and  $ax$ ,  $ay$ ,  $az$  are acceleration coordinates.

The '`StateCovariance`' field of the `tracks` input must have position, velocity, and turn-rate covariances in the rows and columns corresponding to the position, velocity, and turn-rate of the '`State`' field of the `tracks` input. '`StateCovariance`' is required only if '`posnees`' or '`velnees`' is selected in the `Distance` property.

#### **TrackIdentifierFcn — Track identifier function**

`@defaultTrackIdentifier` (default) | function handle

Track identifier function, specified as a function handle. The function extracts track ID from the `tracks` input. The function must support the following syntax:

```
trackids = trackIdentifier(tracks)
```

where

- `tracks` is an array of structures or objects containing the information of tracks.
- `trackids` is a numeric array of the same size as `tracks`.

For an example of a track object, see `objectTrack`. If you use the default identifier function, `defaultTrackIdentifier`, you must include track ID in `tracks` as the value of the `TrackID` field or property.

Example: `@myTrackIdentifier`

#### **TruthIdentifierFcn — Truth identifier function**

`@defaultTruthIdentifier` (default) | function handle

Truth identifier function, specified as a function handle. The function extracts truth ID from `truths` input. The function must support the following syntax:

```
truthIDs = truthIdentifier(truths)
```

where

- `truths` is an array of structures or objects containing the information of truths.
- `truthIDs` is a numeric array of the same size as `truths`.

If you use the default identifier function, `defaultTruthIdentifier`, you must include the truth ID in `truths` as a value of the `PlatformID` field or property.

Example: `@myTruthIdentifier`

**HasAssignmentInput — Enable assignment input**  
`false` (default) | `true`

Enable assignment input, specified as `true` or `false`. This property enables providing the `assignment` input at each time step. The computed GOSPA metric uses the input assignment to compute the localization component.

Data Types: logical

## Usage

### Syntax

```
sGOSPA = GOSPAMetric(tracks, truths)
[sGOSPA, GOSPA, switching] = OSPAMetric(tracks, truths)
[___] = GOSPAMetric(tracks, truths, assignment)
[sGOSPA, GOSPA, switching, localization, missTarget, falseTrack] = GOSPAMetric(_____
)
```

### Description

`sGOSPA = GOSPAMetric(tracks, truths)` returns the GOSPA metric between the set of tracks and truths, including the switching penalty. The value of the switching penalty included in the metric depends on the `SwitchingPenalty` property. By default, the metric uses the global nearest neighbor (GNN) assignments at the current and the previous step to decide if the tracks are switched.

`[sGOSPA, GOSPA, switching] = OSPAMetric(tracks, truths)` also returns the GOSPA component and the switching component.

`[___] = GOSPAMetric(tracks, truths, assignment)` allows you to specify the current assignments between tracks and truths used in the metric evaluation. You can return outputs as any of the previous syntaxes.

To use this syntax, set the `HasAssignmentInput` property to `true`.

`[sGOSPA, GOSPA, switching, localization, missTarget, falseTrack] = GOSPAMetric(____)` also returns the localization component, missed target component, and the false track component. You can use any of the input combinations in the previous syntaxes.

To use this syntax, set the value of the `Alpha` property to 2.

## Input Arguments

### **tracks — Track information**

array of structures | array of objects

Track information, specified as an array of structures or objects for built-in distance functions. Each structure or object must contain **State** as a field or property. Additionally, if a NEES-based distance (**posnees** or **velnees**) is specified in the **Distance** property, each structure or object must also contain **StateCovariance** as a field or property. Moreover, if the default track identifier function is used in the **TrackIdentifierFcn** property, then each structure or object must also contain **TrackID** as a field or property. See **objectTrack** for an example of track object.

Data Types: struct | object

### **truths — Truth information**

array of structures | array of objects

Truth information, specified as an array of structures or objects for built-in distance functions. Each structure or object must contain **Position** and **Velocity** as fields or properties. If the default truth identifier function is used in the **TruthIdentifierFcn** property, then each structure or object must also contain **PlatformID** as a field or property.

Data Types: struct | object

### **assignment — Known current assignment**

N-by-2 matrix of nonnegative integers

Known current assignment, specified as an N-by-2 matrix of nonnegative integers. The first column elements are track IDs, and the second column elements are truth IDs. The IDs in the same row are tracks and truths assigned to each other. If a track (or a truth) is not assigned, specify 0 as the same row element for the truth (or the track).

Note that the assignment must be a unique assignment between tracks and truths. Redundant or false tracks should be treated as unassigned tracks by assigning them to the "0" **TruthID**.

Data Types: single | double

## Output Arguments

### **sGOSPA — GOSPA metric including switching component**

nonnegative real scalar

GOSPA metric including switching component, returned as a nonnegative real scalar.

### **GOSPA — GOSPA metric**

nonnegative real scalar

GOSPA metric, returned as a nonnegative real scalar.

### **switching — Switching component**

nonnegative real scalar

Switching component, returned as a nonnegative real scalar.

### **localization — Localization component**

nonnegative real scalar

Localization component, returned as a nonnegative real scalar.

**missTarget — Missed target component**

nonnegative real scalar

Missed target component, returned as a nonnegative real scalar.

**falseTrack — False track component**

nonnegative real scalar

False track component, returned as a nonnegative real scalar.

## 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
clone	Create duplicate System object
isLocked	Determine if System object is in use

## Examples

### Evaluate Tracking Results Using GOSPA Metric

Load prerecorded data.

```
load trackmetricex tracklog truthlog;
```

Create a `trackGOSPA` object and set the `SwitchingPenalty` to 5.

```
tgm = trackGOSPA('SwitchingPenalty',5);
```

Create output variables.

```
lgospa = zeros(numel(tracklog),1);
gospa = zeros(numel(tracklog),1);
switching = zeros(numel(tracklog),1);
localization = zeros(numel(tracklog),1);
missTarget = zeros(numel(tracklog),1);
falseTracks = zeros(numel(tracklog),1);
```

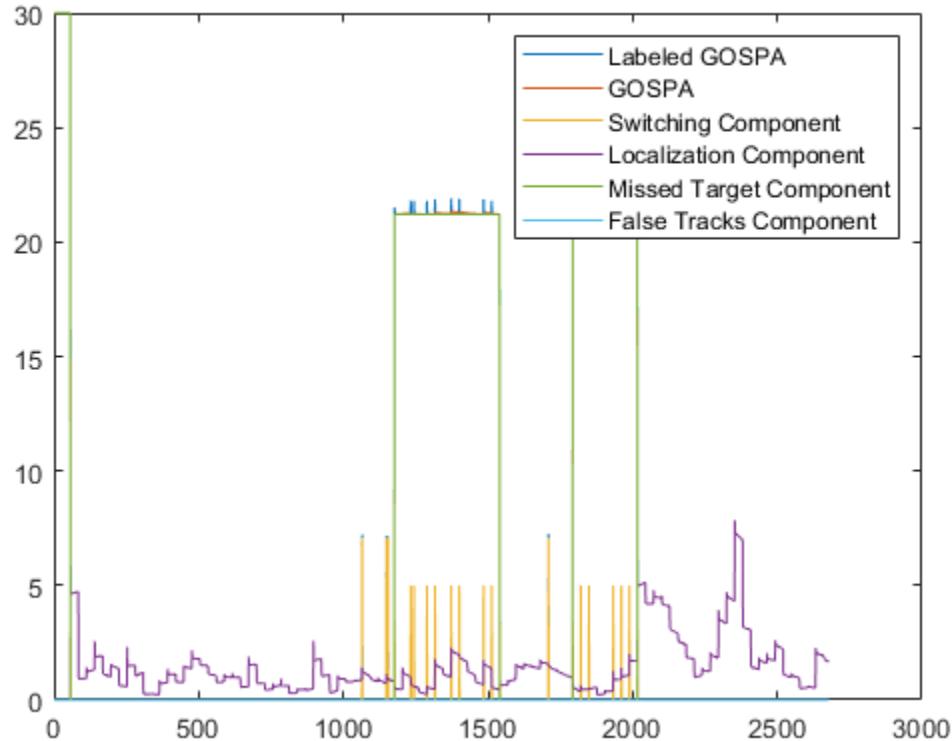
After extracting the tracks and ground truths, run the GOSPA metric.

```
for i = 1:numel(tracklog)
    tracks = tracklog{i};
    truths = truthlog{i};
```

```
[lgospa(i),gospa(i),switching(i),localization(i),missTarget(i),falseTracks(i)] = tgm(tracks,i)
end
```

Visualize the results.

```
plot([lgospa gospa switching localization missTarget falseTracks])
legend('Labeled GOSPA','GOSPA','Switching Component',...
'Localization Component','Missed Target Component','False Tracks Component')
```



## Algorithms

### GOSPA Metric

At time  $t_k$ , a list of truths is:

$$X = [x_1, x_2, \dots, x_m]$$

At time  $t_k$ , a tracker obtains a list of tracks:

$$Y = [y_1, y_2, \dots, y_n]$$

In general, the GOSPA metric including the switching component (*SGOSPA*) is:

$$SGOSPA = (GOSPA^p + SC^p)^{1/p}$$

where  $p$  is the order of the metric,  $SC$  is the switching component, and  $GOSPA$  is the basic GOSPA metric.

Assuming  $m \leq n$ ,  $GOSPA$  is:

$$GOSPA = \left[ \sum_{i=0}^m d_c^p(x_i, y_{\Pi(i)}) + \frac{c^p}{\alpha}(n - m) \right]^{1/p}$$

where  $d_c$  is the cutoff-based distance and  $y_{\Pi(i)}$  represents the track assigned to truth  $x_i$ . The cutoff-based distance  $d_c$  is defined as:

$$d_c(x, y) = \min\{d_b(x, y), c\}$$

where  $c$  is the cutoff distance threshold, and  $d_b(x, y)$  is the base distance between track  $x$  and truth  $y$  calculated by a distance function specified in the `Distance` property. The cutoff based distance  $d_c$  is the smaller value of  $d_b$  and  $c$ .  $\alpha$  is the alpha parameter defined in the `Alpha` property.

The switching component  $SC$  is:

$$SC = SP \times n_s^{1/p}$$

where  $SP$  is the switching penalty defined by the `SwitchingPenalty` property and  $n_s$  is the number of switches. When a track switches assignment from one truth to another truth, the number of switching is counted as 1. When a track switches from assigned to unassigned or switches from unassigned to assigned, the number of switching is counted as 0.5. For example, as shown in the table, Tracks 1 and 2 both switched to different truths, whereas Track 3 switched from assigned to unassigned. Therefore, the total number of switching is 2.5.

### Track Switching Scenario

Previous		Current	
Tracks	Truths	Tracks	Truths
1	3	1	7
2	5	2	3
3	7	3	0

When  $\alpha = 2$ , the GOSPA metric can reduce to three components:

$$GOSPA = [loc^p + miss^p + false^p]^{1/p}$$

The localization component (*loc*) is calculated as:

$$loc = \left[ \sum_{i=1}^h d_b^p(x_i, y_{\Pi(i)}) \right]^{1/p}$$

where  $h$  is the number of nontrivial assignments. A trivial assignment is when a track is assigned to no truth. The missed target component is calculated as:

$$miss = \frac{C}{2^{1/p}}(n_{miss})^{1/p}$$

where  $n_{miss}$  is the number of missed targets. The false track component is calculated as:

$$false = \frac{C}{2^{1/p}}(n_{false})^{1/p}$$

where  $n_{false}$  is the number of false tracks.

If  $m > n$ , simply exchange  $m$  and  $n$  in the formulation to obtain the GOSPA metric.

## References

- [1] Rahmathullah, A. S., A. F. García-Fernández, and L. Svensson. "Generalized Optimal Sub-Pattern Assignment Metric." *20th International Conference on Information Fusion (Fusion)*, pp. 1-8, 2017.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[trackAssignmentMetrics](#) | [trackErrorMetrics](#) | [trackOSPAMetric](#)

### Introduced in R2020a

# trackerTOMHT

Multi-hypothesis, multi-sensor, multi-object tracker

## Description

The `trackerTOMHT` System object is a multi-hypothesis tracker capable of processing detections of many targets from multiple sensors. The tracker initializes, confirms, predicts, corrects, and deletes tracks. Inputs to the tracker are detection reports generated by `objectDetection`, `radarSensor`, `monostaticRadarSensor`, `irSensor`, or `sonarSensor` objects. The tracker estimates the state vector and state vector covariance matrix for each track. The tracker assigns detections based on a track-oriented, multi-hypothesis approach. Each detection is assigned to at least one track. If the detection cannot be assigned to any track, the tracker creates a track.

Any new track starts in a *tentative* state. If enough detections are assigned to a tentative track, its status changes to *confirmed*. If the detection already has a known classification (the `ObjectClassID` field of the returned track is nonzero), that track is confirmed immediately. When a track is confirmed, the multi-object tracker considers the track to represent a physical object. If detections are not assigned to the track within a specifiable number of updates, the track is deleted. For an overview of how the tracker functions, see “Algorithms” on page 3-386.

To track objects using the multi-hypothesis tracker:

- 1** Create the `trackerTOMHT` 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 = trackerTOMHT
tracker = trackerTOMHT(Name,Value)
```

### Description

`tracker = trackerTOMHT` creates a `trackerTOMHT` System object with default property values.

`tracker = trackerTOMHT(Name,Value)` sets properties for the multi-object tracker using one or more name-value pairs. For example, `trackerTOMHT('FilterInitializationFcn',@initcvukf,'MaxNumTracks',100)` creates a multi-object tracker that uses a constant-velocity, unscented Kalman filter and allows 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. You must specify this property as a positive integer to use the track outputs as inputs to a track fuser.

Example: 1

### **FilterInitializationFcn — Filter initialization function**

@initcvekf (default) | function handle | character vector

Filter initialization function, specified as a function handle or as a character vector containing the name of a filter initialization function. The tracker uses a filter initialization function when creating new tracks.

Sensor Fusion and Tracking Toolbox supplies many initialization functions that you can use to specify `FilterInitializationFcn`.

Initialization Function	Function Definition
<code>initcvabf</code>	Initialize constant-velocity alpha-beta filter
<code>initcaabf</code>	Initialize constant-acceleration alpha-beta filter
<code>initcvekf</code>	Initialize constant-velocity extended Kalman filter.
<code>initcackf</code>	Initialize constant-acceleration cubature filter.
<code>initctckf</code>	Initialize constant-turn-rate cubature filter.
<code>initcvckf</code>	Initialize constant-velocity cubature filter.
<code>initcapf</code>	Initialize constant-acceleration particle filter.
<code>initctpf</code>	Initialize constant-turn-rate particle filter.
<code>initcvpf</code>	Initialize constant-velocity particle 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.

Initialization Function	Function Definition
<code>initcaukf</code>	Initialize constant-acceleration unscented Kalman filter.
<code>initctekf</code>	Initialize constant-turn-rate extended Kalman filter.
<code>initctukf</code>	Initialize constant-turn-rate unscented Kalman filter.
<code>initcvmscekf</code>	Initialize constant-velocity modified spherical coordinates extended Kalman filter.
<code>initrpekf</code>	Initialize constant-velocity range-parametrized extended Kalman filter.
<code>initapekf</code>	Initialize constant-velocity angle-parametrized extended Kalman filter.
<code>initekfimm</code>	Initialize tracking IMM filter.
<code>initsingerekf</code>	Initialize singer acceleration extended 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 `objectDetection`. The output of this function must be a filter object: `trackingKF`, `trackingEKF`, `trackingUKF`, `trackingCKF`, `trackingPF`, `trackingMSCEKF`, `trackingGSF`, `trackingIMM`, 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
```

Data Types: `function_handle` | `char`

**MaxNumTracks — Maximum number of tracks**

`100` (default) | positive integer

Maximum number of tracks that the tracker can maintain, specified as a positive integer.

Data Types: `single` | `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.

`MaxNumSensors` must be greater than or equal to the largest value of `SensorIndex` found in all the detections used to update the tracker. `SensorIndex` is a property of an `objectDetection` object. The `MaxNumSensors` property determines how many sets of `ObjectAttributes` fields each output track can have.

Data Types: `single` | `double`

**StateParameters — Parameters of track state reference frame**

`struct([ ])` (default) | `struct` array

Parameters of the track state reference frame, specified as a structure array. Use this property to define the track state reference frame and how to transform the track from the tracker coordinate system, the source system, to the fuser coordinate system.

**Tunable:** Yes

Data Types: `struct`

**MaxNumHypotheses — Maximum number of hypotheses to maintain**

5 (default) | positive integer

Maximum number of hypotheses maintained by the tracks in cases of ambiguity, specified as a positive integer. Larger values increase the computational load.

Example: 10

Data Types: `single` | `double`

**MaxNumTrackBranches — Maximum number of track branches per track**

3 (default) | positive scalar

Set the maximum number of track branches (hypotheses) allowed for each track. Larger values increase the computational load.

Data Types: `single` | `double`

**MaxNumHistoryScans — Maximum number of scans maintained in the branch history**

4 (default) | positive integer

Maximum number of scans maintained in the branch history, specified as a positive integer. The number of track history scans is typically from 2 through 6. Larger values increase the computational load.

Example: 6

Data Types: `single` | `double`

**AssignmentThreshold — Detection assignment threshold**

`30*[0.3 0.7 1 Inf]` (default) | positive scalar | 1-by-3 vector of positive values | 1-by-4 vector of positive values

Detection assignment threshold, specified as a positive scalar, an 1-by-3 vector of non-decreasing positive values,  $[C_1, C_2, C_3]$ , or an 1-by-4 vector of non-decreasing positive values,  $[C_1, C_2, C_3, C_4]$ . If specified as a scalar, the specified value,  $val$ , will be expanded to  $[0.3, 0.7, 1, Inf]*val$ . If specified as  $[C_1, C_2, C_3]$ , it will be expanded as  $[C_1, C_2, C_3, Inf]$ .

The thresholds control (1) the assignment of a detection to a track, (2) the creation of a new branch from a detection, and (3) the creation of a new branch from an unassigned track. The threshold values must satisfy:  $C_1 \leq C_2 \leq C_3 \leq C_4$ .

- $C_1$  defines a distance such that if a track has an assigned detection with lower distance than  $C_1$ , the track is no longer considered unassigned and does not create an unassigned track branch.
- $C_2$  defines a distance that if a detection has been assigned to a track with lower distance than  $C_2$ , the detection is no longer considered unassigned and does not create a new track branch.
- $C_3$  defines the maximum distance for assigning a detection to a track.
- $C_4$  defines combinations of track and detection for which an accurate normalized cost calculation is performed. 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  $C_4$ .

Tips:

- Increase the value of  $C_3$  if there are detections that should be assigned to tracks but are not. Decrease the value if there are detections that are assigned to tracks they should not be assigned to (too far away).
- Increasing the values  $C_1$  and  $C_2$  helps control the number of track branches that are created. However, doing so reduces the number of branches (hypotheses) each track has.
- Increase the value of  $C_4$  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.

Data Types: single | double

**ConfirmationThreshold — Minimum score required to confirm track**

20 (default) | positive scalar

Minimum score required to confirm a track, specified as a positive scalar. Any track with a score higher than this threshold is confirmed.

Example: 12

Data Types: single | double

**DeletionThreshold — Maximum score drop for track deletion**

-7 (default) | scalar

The maximum score drop before a track is deleted, specified as a scalar. Any track with a score that falls by more than this parameter from the maximum score is deleted. Deletion threshold is affected by the probability of false alarm.

Example: 12

Data Types: single | double

**DetectionProbability — Probability of detection used for track score**

0.9 (default) | positive scalar between 0 and 1

Probability of detection, specified as a positive scalar between 0 and 1. This property is used to compute track score.

Example: 0.5

Data Types: single | double

**FalseAlarmRate — Probability of false alarm used for track score**

1e-6 (default) | scalar

The probability of false alarm, specified as a scalar. This property is used to compute track score.

Example: 1e-5

Data Types: single | double

**Beta — Rate of new tracks per unit volume**

1 (default) | positive scalar

The rate of new tracks per unit volume, specified as a positive scalar. The rate of new tracks is used in calculating the track score during track initialization.

Example: 2.5

Data Types: single | double

**Volume — Volume of sensor measurement bin**

1 (default) | positive scalar

The volume of a sensor measurement bin, specified as a positive scalar. For example, if a radar produces a 4-D measurement, which includes azimuth, elevation, range, and range rate, the 4-D volume is defined by the radar angular beam width, the range bin width and the range-rate bin width. Volume is used in calculating the track score when initializing and updating a track.

Example: 1.5

Data Types: single | double

**MinBranchProbability — Minimum probability required to keep track**

.001 (default) | positive scalar

Minimum probability required to keep a track, specified as a positive scalar less than one. Any track with lower probability is pruned. Typical values are 0.001 to 0.005.

Example: .003

Data Types: single | double

**NScanPruning — N-scan pruning method**

'None' (default) | 'Hypothesis'

N-scan pruning method, specified as 'None' or 'Hypothesis'. In N-scan pruning, branches that belong to the same track are pruned (deleted) if, in the N-scans history, they contradict the most likely branch for the same track. The most-likely branch is defined in one of two ways:

- 'None' - No N-scan pruning is performed.
- 'Hypothesis' - The chosen branch is in the most likely hypothesis.

Example: 'Hypothesis'

**HasCostMatrixInput — Enable cost matrix input**

false (default) | true

Enable a cost matrix, specified as false or true. If true, you can provide an assignment cost matrix as an input argument when calling the object.

Data Types: logical

**HasDetectableBranchIDsInput — Enable input of detectable branch IDs**

false (default) | true

Enable the input of detectable branch IDs at each object update, specified as false or true. Set this property to true if you want to provide a list of detectable branch IDs. This list tells the tracker of all branches that the sensors are expected to detect and, optionally, the probability of detection for each branch.

Data Types: logical

**OutputRepresentation — Track output method**

'Tracks' (default) | 'Hypothesis' | 'Clusters'

Track output method, specified as 'Tracks', 'Hypothesis', or 'Clusters'.

- 'Tracks' - Output the centroid of each track based on its track branches.
- 'Hypothesis' - Output branches that are in certain hypotheses. If you choose this option, list the hypotheses to output using the **HypothesesToOutput** property.
- 'Clusters' - Output the centroid of each cluster. Similar to 'Tracks' output, but includes all tracks within a cluster.

Data Types: char

**HypothesesToOutput — Indices of hypotheses to output**

1 (default) | positive integer | array of positive integers

Indices of hypotheses to output, specified as an array of positive integers. The indices must all be less than or equal to the maximum number of hypotheses provided by the tracker.

**Tunable:** Yes

Data Types: single | double

**NumTracks — Number of tracks maintained by tracker**

nonnegative integer

This property is read-only.

Number of tracks maintained by the tracker, returned as a nonnegative integer.

Data Types: double

**NumConfirmedTracks — Number of confirmed tracks**

nonnegative integer

This property is read-only.

Number of confirmed tracks, returned as a nonnegative integer. If the **IsConfirmed** field of an output track structure is **true**, the track is confirmed.

Data Types: double

## Usage

To process detections and update tracks, call the tracker with arguments, as if it were a function (described here).

## Syntax

```

confirmedTracks = tracker(detections,time)
confirmedTracks = tracker(detections,time,costMatrix)
confirmedTracks = tracker(___,detectableBranchIDs)
[confirmedTracks,tentativeTracks,allTracks] = tracker(___)
[confirmedTracks,tentativeTracks,allTracks,analysisInformation] = tracker(___)
    
```

## Description

`confirmedTracks = tracker(detections, time)` returns a list of confirmed tracks that are updated from a list of detections, `detections`, at the update time, `time`. Confirmed tracks are corrected and predicted to the update time.

`confirmedTracks = tracker(detections, time, costMatrix)` also specifies a cost matrix, `costMatrix`.

To enable this syntax, set the `HasCostMatrixInput` property to `true`.

`confirmedTracks = tracker(____, detectableBranchIDs)` also specifies a list of expected detectable branches, `detectableBranchIDs`.

To enable this syntax, set the `HasDetectableBranchIDsInput` property to `true`.

`[confirmedTracks, tentativeTracks, allTracks] = tracker(____)` also returns a list of tentative tracks, `tentativeTracks`, and a list of all tracks, `allTracks`.

`[confirmedTracks, tentativeTracks, allTracks, analysisInformation] = tracker(____)` also returns information, `analysisInformation`, useful for track analysis.

## 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 update time, `time`, and greater than the previous time value used to update the tracker. Also, the `Time` differences between different `objectDetection` objects in the cell array do not need to be equal.

### **time — Time of update**

scalar

Time of update, specified as a 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 tracker.

Data Types: `single` | `double`

### **costMatrix — Cost matrix**

real-valued  $N$ -by- $M$  matrix

Cost matrix, specified as a real-valued  $N$ -by- $M$  matrix, where  $N$  is the number of branches, and  $M$  is the number of current detections. The cost matrix rows must be in the same order as the list of branches. The columns must be in the same order as the list of detections. Obtain the correct order of the list of branches using the `getBranches` object function. Matrix columns correspond to the detections.

At the first update of the object or when the tracker has no previous tracks, specify the cost matrix to have a size of `[0, numDetections]`. Note that the cost must be calculated so that lower costs indicate a higher likelihood of assigning a detection to a track. To prevent certain detections from being assigned to certain tracks, set the appropriate cost matrix entry to `Inf`.

## Dependencies

To enable this argument, set the `HasCostMatrixInput` property to `true`.

Data Types: `double` | `single`

### **detectableBranchIDs — Detectable branch IDs**

real-valued  $M$ -by-1 vector | real-valued  $M$ -by-2 matrix

Detectable branch IDs, specified as a real-valued  $M$ -by-1 vector or  $M$ -by-2 matrix. Detectable branches are branches that the sensors expect to detect. The first column of the matrix contains a list of branch IDs of tracks reported in the `branchID` field of the track output arguments. The second column contains the detection probability for the branch. Sensors can report detection probability, but if not reported, detection probabilities are obtained from the `DetectionProbability` property.

Branches whose identifiers are not included in `detectableBranchIDs` are considered as undetectable. The track deletion logic does not count the lack of detection as a 'miss' for branch deletion purposes.

## Dependencies

To enable this input argument, set the `HasDetectableBranchIDs` 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`

### **analysisInformation — Additional information for analyzing track updates**

structure

Additional information for analyzing track updates, returned as a structure. The fields of this structure are:

Field	Description
<code>BranchIDsAtStepBeginning</code>	Branch IDs when update began.
<code>CostMatrix</code>	Cost of assignment matrix.
<code>Assignments</code>	Assignments returned from <code>assignTOMHT</code> .
<code>UnassignedTracks</code>	IDs of unassigned branches returned from the tracker
<code>UnassignedDetections</code>	Indices of unassigned detections in the <code>detections</code> input.
<code>InitialBranchHistory</code>	Branch history after branching and before pruning.
<code>InitialBranchScores</code>	Branch scores before pruning.
<code>KeptBranchHistory</code>	Branch history after initial pruning.
<code>KeptBranchScores</code>	Branch scores after initial pruning.
<code>Clusters</code>	Logical array mapping branches to clusters. Branches belong in the same cluster if they share detections in their history or belong to the same track, either directly or through other branches. Such branches are incompatible.
<code>TrackIncompatibility</code>	Branch incompatibility matrix. The $(i, j)$ element is true if the $i$ -th and $j$ -th branches have shared detections in their history or belong to the same track.
<code>GlobalHypotheses</code>	Logical matrix mapping branches to global hypotheses. Compatible branches can belong in the same hypotheses.
<code>GlobalHypScores</code>	Total score of global hypotheses.
<code>PrunedBranches</code>	Logical array of branches that the <code>pruneTrackBranches</code> function determines to be pruned.
<code>GlobalBranchProbabilities</code>	Global probability of each branch existing in the global hypotheses.
<code>BranchesDeletedByPruning</code>	Branches deleted by the tracker.
<code>BranchIDsAtStepEnd</code>	Branch IDs when the update ended.

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 trackerTOMHT

<code>getTrackFilterProperties</code>	Obtain track filter properties
<code>setTrackFilterProperties</code>	Set track filter properties
<code>getBranches</code>	Lists track branches
<code>predictTrackToTime</code>	Predict track state
<code>initializeTrack</code>	Initialize new track
<code>deleteTrack</code>	Delete existing track
<code>initializeBranch</code>	Initialize new track branch
<code>deleteBranch</code>	Delete existing track branch

### Common to All System Objects

<code>release</code>	Release resources and allow changes to System object property values and input characteristics
<code>reset</code>	Reset internal states of System object
<code>isLocked</code>	Determine if System object is in use
<code>clone</code>	Create duplicate System object

## Examples

### Track Two Objects Using trackerTOMHT

Create the `trackerTOMHT` System object with a constant-velocity Kalman filter initialization function, `initcvkf`.

```
tracker = trackerTOMHT('FilterInitializationFcn',@initcvkf, ...
    'ConfirmationThreshold',20, ...
    'DeletionThreshold',-7, ...
    'MaxNumHypotheses',10);
```

Update the tracker with two detections having nonzero `ObjectClassID`. The detections immediately create confirmed tracks.

```
detections = {objectDetection(1,[10;0], 'SensorIndex',1, ...
    'ObjectClassID',5, 'ObjectAttributes',{struct('ID',1)}); ...
    objectDetection(1,[0;10], 'SensorIndex',1, ...
    'ObjectClassID',2, 'ObjectAttributes',{struct('ID',2)})};
time = 2;
tracks = tracker(detections,time);
```

Find and display the positions and velocities.

```
positionSelector = [1 0 0 0; 0 0 1 0];
velocitySelector = [0 1 0 0; 0 0 0 1];
positions = getTrackPositions(tracks,positionSelector)

positions = 2×2
```

```
10.0000      0  
0      10.0000  
  
velocities = getTrackVelocities(tracks,velocitySelector)  
velocities = 2x2  
  
0      0  
0      0
```

## Algorithms

### Tracker Logic Flow

When you process detections using the tracker, track creation and management follow these steps.

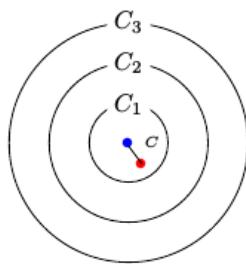
- 1** The tracker attempts to assign detections to existing tracks.
- 2** The track allows for multiple hypotheses about the assignment of detections to tracks.
- 3** Unassigned detections result in the creation of new tracks.
- 4** Assignments of detections to tracks create branches for the assigned tracks.
- 5** Tracks with no assigned detections are coasted (predicted).
- 6** All track branches are scored. Branches with low initial scores are pruned.
- 7** Clusters of branches that share detections (incompatible branches) in their history are generated.
- 8** Global hypotheses of compatible branches are formulated and scored.
- 9** Branches are scored based on their existence in the global hypotheses. Low-scored branches are pruned.
- 10** Additional pruning is performed based on N-scan history.
- 11** All tracks are corrected and predicted to the input time.

### Assignment Thresholds for Multi-Hypothesis Tracker

Three assignment thresholds,  $C_1$ ,  $C_2$ , and  $C_3$ , control (1) the assignment of a detection to a track, (2) the creation of a new branch from a detection, and (3) the creation of a new branch from an unassigned track. The threshold values must satisfy:  $C_1 \leq C_2 \leq C_3$ .

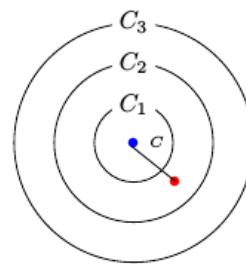
If the cost of an assignment is  $C = \text{costmatrix}(i, j)$ , the following hypotheses are created based on comparing the cost to the values of the assignment thresholds. Below each comparison, there is a list of the possible hypotheses.

- Track
- Detection



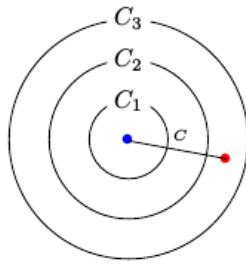
$C \leq C_1$   
Single Hypothesis

- (1) Detection is assigned to track. A branch is created updating the track with this detection.



$C_1 < C \leq C_2$   
Two Hypotheses

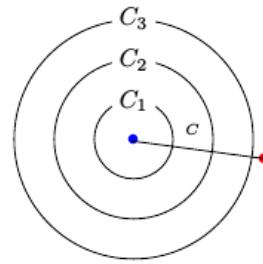
- (1) Detection is assigned to track. A branch is created updating the track with this detection.  
(2) Track is not assigned to detection and is coasted.



$C_2 < C \leq C_3$

Three Hypotheses

- (1) Detection is assigned to track. A branch is created updating the track with this detection.  
(2) Track is not assigned to detection and is coasted.  
(3) Detection is not assigned and creates a new track (branch).



$C_3 < C$

Single Hypothesis

- (1) Detection is not assigned and creates a new track (branch).

Tips:

- Increase the value of  $C_3$  if there are detections that should be assigned to tracks but are not. Decrease the value if there are detections that are assigned to tracks they should not be assigned to (too far away).
- Increasing the values  $C_1$  and  $C_2$  helps control the number of track branches that are created. However, doing so reduces the number of branches (hypotheses) each track has.
- To allow each track to be unassigned, set  $C_1 = 0$ .
- To allow each detection to be unassigned, set  $C_2 = 0$ .

### Data Precision

All numeric inputs can be single or double precision, but they all must have the same precision.

## References

- [1] Werthmann, J. R. "Step-by-Step Description of a Computationally Efficient Version of Multiple Hypothesis Tracking." In *International Society for Optics and Photonics*, Vol. 1698, pp. 228-301, 1992.
- [2] Blackman, S., and R. Popoli. *Design and Analysis of Modern Tracking Systems*. Artech House Radar Library, Boston, 1999.

## 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 a multi-object tracker must have properties with the same sizes and types.
- If you use the `ObjectAttributes` field within an `objectDetection` object, you must specify this field as 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.
- If `ObjectAttributes` are contained in the detection, the `SensorIndex` value of the detection cannot be greater than 10.
- The first update to the multi-object tracker must contain at least one detection.

## See Also

### Functions

`getTrackPositions` | `getTrackVelocities`

### Objects

`objectDetection` | `objectTrack` | `trackingABF` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

### System Objects

`irSensor` | `monostaticRadarSensor` | `radarSensor` | `sonarSensor` | `trackerGNN`

### Introduced in R2018b

# deleteBranch

Delete existing track branch

## Syntax

```
deleted = delteTrack(tracker,branchID)
```

## Description

`deleted = delteTrack(tracker,branchID)` deletes the track branch specified by `branchID` in the tracker.

## Input Arguments

### **tracker — TOMHT tracker**

trackerTOMHT object

TOMHT tracker, specified a `trackerTOMHT` object.

### **branchID — Track branch identifier**

positive integer

Track branch identifier, specified as a positive integer.

Example: 21

## Output Arguments

### **deleted — Indicate if track branch was successfully deleted**

true | false

Indicate if the track branch was successfully deleted or not, returned as `true` or `false`. If the track branch specified by the `branchID` input existed and was successfully deleted, it returns as true. If the track branch did not exist, a warning is issued and it returns as false.

## Extended Capabilities

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

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

## See Also

`trackFuser` | `trackerGNN` | `trackerJPDA` | `trackerTOMHT`

## Introduced in R2020a

# initializeBranch

Initialize new track branch

## Syntax

```
branchID = initializeTrack(tracker,branch)
branchID = initializeTrack(tracker,branch,filter)
```

## Description

`branchID = initializeTrack(tracker,branch)` initializes a new track branch in the tracker. The tracker must be updated at least once before initializing a track branch. If the track is initialized successfully, the tracker assigns the output `branchID` to the branch, set the `UpdateTime` of the branch equal to the last step time, and synchronizes the data in the input `branch` to the initialized branch.

A warning is issued if the tracker already maintains the maximum number of track branches specified by the `MaxNumTrackBranches` property of the tracker. In this case, the `branchID` is returned as zero, which indicates a failure to initialize the branch.

---

**Note** This syntax doesn't support using the `trackingGSF`, `trackingPF`, or `trackingIMM` filter object as the internal tracking filter for the tracker. Use the second syntax for these cases.

---

`branchID = initializeTrack(tracker,branch,filter)` initializes a new track branch in the tracker using a specified tracking filter, `filter`.

---

**Note** If the tracking filter used in the tracker is `trackingGSF`, `trackingPF`, or `trackingIMM`, you must use this syntax instead of the first syntax.

---

## Input Arguments

**tracker — TOMHT tracker**  
trackerTOMHT object

TOMHT tracker, specified a `trackerTOMHT` object.

**branch — New track branch to be initialized**  
objectTrack | 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 outputted by the tracker.

Data Types: struct | object

**filter — Filter object**

trackingKF | trackingEKF | trackingUKF | trackingABF | trackingCKF | trackingMSCEKF | trackingPF | trackingIMM | trackingGSF

Filter object, specified as a trackingKF, trackingEKF, trackingUKF, trackingABF, trackingCKF, trackingIMM, trackingGSF, trackingPF, or trackingMSCEKF object.

## Output Arguments

**branchID — Track branch identifier**

nonnegative integer

Track identifier, returned as a nonnegative integer. trackID is returned as 0 if the branch is not initialized successfully.

Example: 2

**See Also**

trackerTOMHT

**Introduced in R2020a**

# getTrackFilterProperties

Obtain track filter properties

## Syntax

```
filtervalues = getTrackFilterProperties(tracker,branchID,properties)
filtervalues = getTrackFilterProperties(tracker,trackID,properties)
```

## Description

`filtervalues = getTrackFilterProperties(tracker,branchID,properties)` returns the values, `filtervalues`, of tracking filter properties, `properties`, for the specified branch, `branchID`.

This syntax applies when you create the tracker using `trackerTOMHT`.

`filtervalues = getTrackFilterProperties(tracker,trackID,properties)` returns the values, `filtervalues`, of tracking filter properties, `properties`, for the specified track, `trackID`.

This syntax applies when you create the tracker using `trackerGNN` or `trackerJPDA`.

## Examples

### Get Multi-Hypothesis Track Filter Properties

Create a track filter with default properties from one detection. Obtain the values of the `MeasurementNoise` and `ProcessNoise` track filter properties.

```
tracker = trackerTOMHT;
detection = objectDetection(0,[0;0;0]);
tracker(detection,0);
branches = getBranches(tracker);
branchID = branches(1).BranchID;
values = getTrackFilterProperties(tracker, branchID, ...
    'MeasurementNoise', 'ProcessNoise')

values=2×1 cell array
{3x3 double}
{3x3 double}

disp(values{1})

1     0     0
0     1     0
0     0     1
```

## Get Global Nearest-Neighbor Track Filter Properties

Create a track filter from one detection. Assume default properties. Obtain the values of the `MeasurementNoise` and `ProcessNoise` track filter properties.

```
tracker = trackerGNN;
detection = objectDetection(0,[0;0;0]);
[~,tracks] = tracker(detection,0);
values = getTrackFilterProperties(tracker,tracks.TrackID, ...
    'MeasurementNoise','ProcessNoise')

values=2x1 cell array
{3x3 double}
{3x3 double}

disp(values{1})

1     0     0
0     1     0
0     0     1
```

## Input Arguments

### **tracker — Target tracker**

`trackerTOMHT` object | `trackerGNN` object

Target tracker, specified as a `trackerTOMHT` or `trackerGNN` object.

### **branchID — Branch identifier**

positive integer

Branch identifier, specified as a positive integer. The identifier must be a valid `BranchID` reported in the list of branches returned by the `getBranches` object function.

Example: 21

### **Dependencies**

Data Types: `uint32`

### **trackID — Track identifier**

positive integer

Track identifier, specified as a positive integer. `trackID` must be a valid track identifier as reported from the previous track update.

Example: 21

Data Types: `uint32`

### **properties — Filter properties**

comma-delimited list of properties

Filter properties, specified as a comma-delimited list of valid tracker properties to obtain. Enclose each property in single quotes.

Example: `'MeasurementNoise','ProcessNoise'`

Data Types: char

## Output Arguments

**filtervalues — Filter property values**  
cell array

Filter property values, returned as a cell array. Filter values are returned in the same order as the list of properties.

**Introduced in R2018b**

# setTrackFilterProperties

Set track filter properties

## Syntax

```
setTrackFilterProperties(tracker,branchID,'Name',Value)
setTrackFilterProperties(tracker,trackID,'Name',Value)
```

## Description

`setTrackFilterProperties(tracker,branchID,'Name',Value)` sets the values of tracking filter properties of the tracker, `tracker`, for the branch specified by, `branchID`. Use valid Name-Value pairs to set properties for the branch. You can specify as many Name-Value pairs as you wish. Property names must match the names of public filter properties. This syntax applies when you create the tracker using `trackerTOMHT`.

`setTrackFilterProperties(tracker,trackID,'Name',Value)` sets the values of tracking filter properties of the tracker, `tracker`, for the track, `trackID`. Use Name-Value pairs to set properties for the track. You can specify as many Name-Value pairs as you wish. Property names must match the names of public filter properties. This syntax applies when you create the tracker using `trackerGNN` or `trackerJPDA`.

## Examples

### Set Multi-Hypothesis Tracking Filter Properties

Create a tracker using `trackerTOMHT`. Assign values to the `MeasurementNoise` and `ProcessNoise` properties and verify the assignment.

```
tracker = trackerTOMHT;
detection = objectDetection(0,[0;0;0]);
tracker(detection,0);
branches = getBranches(tracker);
branchID = branches(1).BranchID;
setTrackFilterProperties(tracker,branchID,'MeasurementNoise',2,'ProcessNoise',5);
values = getTrackFilterProperties(tracker,branchID,'MeasurementNoise','ProcessNoise');
```

Show the measurement noise.

```
disp(values{1})
```

2.0000	0	0
0	2.0000	0
0	0	2.0000

Show the process noise.

```
disp(values{2})
```

```
5.0000      0      0
 0    5.0000      0
 0      0    5.0000
```

### Set Global Nearest-Neighbor Track Filter Properties

Create a tracker using `trackerGNN`. Assign values to the `MeasurementNoise` and `ProcessNoise` properties and verify the assignment.

```
tracker = trackerGNN;
detection = objectDetection(0,[0;0;0]);
[~, tracks] = tracker(detection,0);
setTrackFilterProperties(tracker,1,'MeasurementNoise',2,'ProcessNoise',5);
values = getTrackFilterProperties(tracker,1,'MeasurementNoise','ProcessNoise');
```

Show the measurement noise.

```
disp(values{1})
```

```
2.0000      0      0
 0    2.0000      0
 0      0    2.0000
```

Show the process noise.

```
disp(values{2})
```

```
5.0000      0      0
 0    5.0000      0
 0      0    5.0000
```

## Input Arguments

### **tracker — Target tracker**

`trackerTOMHT` object | `trackerGNN` object

Target tracker, specified as a `trackerTOMHT` or `trackerGNN` object.

### **branchID — Branch identifier**

positive integer

Branch identifier, specified as a positive integer. The identifier must be a valid `BranchID` reported in the list of branches returned by the `getBranches` object function.

Example: 21

Data Types: `uint32`

### **trackID — Track identifier**

positive integer

Track identifier, specified as a positive integer. `trackID` must be a valid track identifier as reported from the previous track update.

Example: 21

Data Types: uint32

**Introduced in R2018b**

# getBranches

Lists track branches

## Syntax

```
branches = getBranches(tracker)
```

## Description

`branches = getBranches(tracker)` returns a list of track branches maintained by the tracker. The tracker must be updated at least once before calling this object function. Use `isLocked(tracker)` to test whether the tracker has been updated.

## Examples

### Get Multi-Hypothesis Tracker Branches

Create a multi-hypothesis tracker with one detection and obtain its branches.

```
tracker = trackerTOMHT;
detection = objectDetection(0,[0;0;0]);
tracker(detection,0);
branches = getBranches(tracker)

branches =
  objectTrack with properties:

    TrackID: 1
    BranchID: 1
    SourceIndex: 0
    UpdateTime: 0
    Age: 1
    State: [6x1 double]
    StateCovariance: [6x6 double]
    StateParameters: [1x1 struct]
    ObjectClassID: 0
    TrackLogic: 'Score'
    TrackLogicState: [13.7102 13.7102]
    IsConfirmed: 0
    IsCoasted: 0
    IsSelfReported: 1
    ObjectAttributes: [1x1 struct]
```

## Input Arguments

### tracker — Target tracker

`trackerTOMHT` object | `trackerGNN` object

Target tracker, specified as a `trackerTOMHT` or `trackerGNN` object.

## Output Arguments

### **branches — List of track branches**

structure | array of structures

List of track branches, returned as an array of track structure or array of track structures.

Field	Description
TrackID	Integer that identifies the track.
BranchID	Unique integer that identifies the track branch (hypothesis).
UpdateTime	Time to which the track is updated.
Age	Number of times the track was updated with either a hit or a miss.
State	Value of state vector at update time.
StateCovariance	Uncertainty covariance matrix.
TrackLogic	The track logic used. Values are either 'History' or 'Score'.
TrackLogicState	<p>The current state of the track logic.</p> <ul style="list-style-type: none"> <li>For 'History' track logic, a 1-by-<math>Q</math> logical array, where <math>Q</math> is the greater of <math>N</math> or <math>R</math> from the confirmation and deletion thresholds.</li> <li>For 'Score' track logic, a 1-by-2 numerical array in the form: [currentScore, maxScore].</li> </ul>
IsConfirmed	True if the track is assumed to be of a real target.
IsCoasted	True if the track has been updated without a detection (predicted).
ObjectClassID	An integer value representing the object classification. Zero is reserved for 'unknown'.
ObjectAttributes	A cell array of cells. Each cell captures the object attributes reported by the corresponding sensor.

Data Types: struct

**Introduced in R2018b**

## **predictTracksToTime**

Predict track state

### **Syntax**

```
predictedtracks = predictTracksToTime(obj,trackid,time)
predictedtracks = predictTracksToTime(obj,category,time)
predictedtracks = predictTracksToTime(obj,type,id,time)
predictedtracks = predictTracksToTime(obj,type,category,time)
predictedtracks = predictTracksToTime(___, 'WithCovariance',tf)
```

### **Description**

`predictedtracks = predictTracksToTime(obj,trackid,time)` returns the predicted tracks, `predictedtracks`, of the tracker or fuser object, `obj`, at the specified time, `time`. Specify the track identifier, `trackid`. The tracker or fuser must be updated at least once before calling this object function. Use `isLocked(obj)` to test whether the tracker or fuser has been updated.

This syntax applies when you create the `obj` using `trackerGNN`, `trackerJPDA`, `trackerPHD`, `trackerGridRFS`, or `trackFuser`.

---

**Note** This function only outputs the predicted tracks and does not update the internal track states of the `tracker` or `fuser`.

---

`predictedtracks = predictTracksToTime(obj,category,time)` returns all predicted tracks for a specified category, `category`, of tracked objects.

This syntax applies when you create the `obj` using `trackerGNN`, `trackerJPDA`, `trackerPHD`, `trackerGridRFS`, or `trackFuser`.

`predictedtracks = predictTracksToTime(obj,type,id,time)` returns the predicted tracks or branches, `predictedtracks`, of the tracker or fuser object, `obj`, at the specified time, `time`. Specify the type, `type`, of tracked object and the object ID, `id`. The tracker or fuser must be updated at least once before calling this object function. Use `isLocked(trackObj)` to test whether the tracker or fuser has been updated.

This syntax applies when you create the `obj` using `trackerTOMHT`.

`predictedtracks = predictTracksToTime(obj,type,category,time)` returns all predicted tracks or branches for a specified category, `category`, of tracked objects.

This syntax applies when you create the `obj` using `trackerTOMHT`.

`predictedtracks = predictTracksToTime(___, '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

Create a track from a detection and predict its state later on.

```
tracker = trackerTOMHT;
detection = objectDetection(0,[0;0;0]);
tracker(detection,0);
branches = getBranches(tracker);
predictedtracks = predictTracksToTime(tracker, 'branch',1,1)

predictedtracks =
objectTrack with properties:

    TrackID: 1
    BranchID: 1
    SourceIndex: 0
    UpdateTime: 1
    Age: 1
    State: [6x1 double]
    StateCovariance: [6x6 double]
    StateParameters: [1x1 struct]
    ObjectClassID: 0
    TrackLogic: 'Score'
    TrackLogicState: [13.7102 13.7102]
    IsConfirmed: 0
    IsCoasted: 0
    IsSelfReported: 1
    ObjectAttributes: [1x1 struct]
```

## Input Arguments

### **obj — Tracker or fuser object**

trackerTOMHT object | trackerJPDA object | trackerGNN object | trackerGridRFS object | trackerFuser object

Tracker or fuser object, specified as a `trackerTOMHT`, `trackerJPDA` object, `trackerGNN` object, `trackerGridRFS` object, or `trackFuser` object.

### **type — Tracked object type**

'track' | 'branch'

Tracked object type, specified as 'track' or 'branch'.

### **id — Track or branch identifier**

positive integer

Track or branch identifier, specified as a positive integer.

Example: 21

Data Types: `single` | `double`

**trackid — Track identifier**

positive integer

Track identifier, specified as a positive integer.

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 structures or an array of objectTrack objects. If the obj input is specified as a trackerGNN, trackerJPDA, or trackFuser object, it is returned as an array of objectTrack objects in MATLAB, and returned as an array of structures with field names same as the property names of objectTrack in code generation. If the obj input is specified as a trackerPHD object, it is returned as an array of structures, in which each structure contains the following fields:

Field	Description
TrackID	Unique integer that identifies the track.
SouceIndex	Unique identifier the tracker in a multiple tracker environment. The SourceIndex is exactly the same with the TrackerIndex.
UpdateTime	The time the track was updated.
Age	Number of times the track survived.
State	Value of state vector at the update time.
StateCovariance	Uncertainty covariance matrix.

Extent	Spatial extent estimate of the tracked object, returned as a $d$ -by- $d$ matrix, where $d$ is the dimension of the object. This field is only returned when the tracking filter is specified as a <code>ggiwphd</code> filter.
MeasurementRate	Expected number of detections from the tracked object. This field is only returned when the tracking filter is specified as a <code>ggiwphd</code> filter.
IsConfirmed	True if the track is assumed to be of a real target.
IsCoasted	<code>trackerPHD</code> does not support the <code>IsCoasted</code> field. The value is always 0.
ObjectClassID	<code>trackerPHD</code> does not support the <code>ObjectClassID</code> field. The value is always 0.
StateParamaters	Parameters about the track state reference frame specified in the <code>StateParameters</code> property of the PHD tracker.
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.

Data Types: struct | object

**Introduced in R2018b**

# trackerGNN

Multi-sensor, multi-object tracker using GNN assignment

## Description

The `trackerGNN` System object is a tracker capable of processing detections of many targets from multiple sensors. The tracker uses a global nearest-neighbor (GNN) assignment algorithm. The tracker initializes, confirms, predicts, corrects, and deletes tracks. Inputs to the tracker are detection reports generated by `objectDetection`, `radarSensor`, `monostaticRadarSensor`, `irSensor`, or `sonarSensor` objects. The tracker estimates the state vector and state vector covariance matrix for each track. Each detection is assigned to at most one track. If the detection cannot be assigned to any track, the tracker initializes a new track.

Any new track starts in a *tentative* state. If enough detections are assigned to a tentative track, its status changes to *confirmed*. If the detection already has a known classification (the `ObjectClassID` field of the returned track is nonzero), that track is confirmed immediately. When a track is confirmed, the tracker considers the track to represent a physical object. If detections are not assigned to the track within a specifiable number of updates, the track is deleted.

To track objects using this object:

- 1 Create the `trackerGNN` 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 = trackerGNN  
tracker = trackerGNN(Name,Value)
```

## Description

`tracker = trackerGNN` creates a `trackerGNN` System object with default property values.

`tracker = trackerGNN(Name,Value)` sets properties for the tracker using one or more name-value pairs. For example,

`trackerGNN('FilterInitializationFcn',@initcvukf,'MaxNumTracks',100)` creates a multi-object tracker that uses a constant-velocity, unscented Kalman filter and allows 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. You must specify this property as a positive integer to use the track outputs as inputs to a track fuser.

Example: 1

#### **FilterInitializationFcn — Filter initialization function**

@initcvekf (default) | function handle | character vector

Filter initialization function, specified as a function handle or as a character vector containing the name of a filter initialization function. The tracker uses a filter initialization function when creating new tracks.

Sensor Fusion and Tracking Toolbox supplies many initialization functions that you can use to specify `FilterInitializationFcn`.

Initialization Function	Function Definition
<code>initcvabf</code>	Initialize constant-velocity alpha-beta filter
<code>initcaabf</code>	Initialize constant-acceleration alpha-beta filter
<code>initcvekf</code>	Initialize constant-velocity extended Kalman filter.
<code>initcackf</code>	Initialize constant-acceleration cubature filter.
<code>initctckf</code>	Initialize constant-turn-rate cubature filter.
<code>initcvckf</code>	Initialize constant-velocity cubature filter.
<code>initcapf</code>	Initialize constant-acceleration particle filter.
<code>initctpf</code>	Initialize constant-turn-rate particle filter.
<code>initcvpf</code>	Initialize constant-velocity particle 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-turn-rate extended Kalman filter.

Initialization Function	Function Definition
<code>initctukf</code>	Initialize constant-turn-rate unscented Kalman filter.
<code>initcvmscekf</code>	Initialize constant-velocity modified spherical coordinates extended Kalman filter.
<code>initrpekf</code>	Initialize constant-velocity range-parametrized extended Kalman filter.
<code>initapekf</code>	Initialize constant-velocity angle-parametrized extended Kalman filter.
<code>initekfimm</code>	Initialize tracking IMM filter.
<code>initsingerekf</code>	Initialize singer acceleration extended 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 `objectDetection`. The output of this function must be a filter object: `trackingKF`, `trackingEKF`, `trackingUKF`, `trackingCKF`, `trackingPF`, `trackingMSCEKF`, `trackingGSF`, `trackingIMM`, 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
```

Data Types: `function_handle` | `char`

#### **Assignment — Assignment algorithm**

'MatchPairs' (default) | 'Munkres' | 'Jonker-Volgenant' | 'Auction' | 'Custom'

Assignment algorithm, specified as '`MatchPairs`', '`Munkres`', '`Jonker-Volgenant`', '`Auction`', or '`Custom`'. `Munkres` is the only assignment algorithm that guarantees an optimal solution, but it is also the slowest, especially for large numbers of detections and tracks. The other algorithms do not guarantee an optimal solution but can be faster for problems with 20 or more tracks and detections. Use '`Custom`' to define your own assignment function and specify its name in the `CustomAssignmentFcn` property.

Example: '`Custom`'

Data Types: `char`

#### **CustomAssignmentFcn — Custom assignment function**

character vector

Custom assignment function name, specified as a character string. An assignment function must have the following syntax:

```
[assignment,unTrs,unDets] = f(cost,costNonAssignment)
```

For an example of an assignment function and a description of its arguments, see `assignmunkres`.

## Dependencies

To enable this property, set the **Assignment** property to 'Custom'.

Data Types: char

### **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, **trackingCKF** and **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).

### **TrackLogic — Confirmation and deletion logic type**

'History' (default) | 'Score'

Confirmation and deletion logic type, specified as 'History' or 'Score'.

- '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.

### **ConfirmationThreshold — Threshold for track confirmation**

scalar | 1-by-2 vector

Threshold for track confirmation, specified as a scalar or a 1-by-2 vector. The threshold depends on the type of track confirmation and deletion logic you set using the **TrackLogic** property.

- History - Specify the confirmation threshold as 1-by-2 vector [M N]. A track is confirmed if it receives at least M detections in the last N updates. The default value is [2, 3].
- Score - Specify the confirmation threshold as a scalar. A track is confirmed if its score is at least as high as the confirmation threshold. The default value is 20.

Data Types: single | double

### **DeletionThreshold — Minimum score required to delete track**

[5 5] or -7 (default) | scalar | real-valued 1-by-2 vector of positive values

Minimum score required to delete track, specified as a scalar or a real-valued 1-by-2 vector. The threshold depends on the type of track confirmation and deletion logic you set using the **TrackLogic** property:

- History - Specify the confirmation threshold as [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.
- Score - A track is deleted if its score decreases by at least the threshold from the maximum track score.

Example: 3

Data Types: single | double

**DetectionProbability — Probability of detection used for track score**

0.9 (default) | positive scalar between 0 and 1

Probability of detection, specified as a positive scalar between 0 and 1. This property is used to compute track score.

Example: 0.5

Data Types: single | double

**FalseAlarmRate — Probability of false alarm used for track score**

1e-6 (default) | scalar

The probability of false alarm, specified as a scalar. This property is used to compute track score.

Example: 1e-5

Data Types: single | double

**Beta — Rate of new tracks per unit volume**

1 (default) | positive scalar

The rate of new tracks per unit volume, specified as a positive scalar. The rate of new tracks is used in calculating the track score during track initialization.

Example: 2.5

Data Types: single | double

**Volume — Volume of sensor measurement bin**

1 (default) | positive scalar

The volume of a sensor measurement bin, specified as a positive scalar. For example, if a radar produces a 4-D measurement, which includes azimuth, elevation, range, and range rate, the 4-D volume is defined by the radar angular beam width, the range bin width and the range-rate bin width. Volume is used in calculating the track score when initializing and updating a track.

Example: 1.5

Data Types: single | double

**MaxNumTracks — Maximum number of tracks**

100 (default) | positive integer

Maximum number of tracks that the tracker can maintain, specified as a positive integer.

Data Types: single | 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. **MaxNumSensors** must be greater than or equal to the largest value of **SensorIndex** found in all the detections used to update the tracker. **SensorIndex** is a property of an **objectDetection** object. The **MaxNumSensors** property determines how many sets of **ObjectAttributes** fields each output track can have.

Data Types: single | double

**StateParameters — Parameters of track state reference frame**

struct([]) (default) | struct array

Parameters of the track state reference frame, specified as a structure array. Use this property to define the track state reference frame and how to transform the track from the tracker coordinate system, the source system, to the fuser coordinate system.

**Tunable:** Yes

Data Types: struct

**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

**HasCostMatrixInput — Enable cost matrix input**

false (default) | true

Enable a cost matrix, specified as **false** or **true**. If **true**, you can provide an assignment cost matrix as an input argument when calling the object.

Data Types: logical

**NumTracks — Number of tracks maintained by tracker**

nonnegative integer

This property is read-only.

Number of tracks maintained by the tracker, returned as a nonnegative integer.

Data Types: double

**NumConfirmedTracks — Number of confirmed tracks**

nonnegative integer

This property is read-only.

Number of confirmed tracks, returned as a nonnegative integer. If the **IsConfirmed** field of an output track structure is **true**, the track is confirmed.

Data Types: double

## Usage

To process detections and update tracks, call the tracker with arguments, as if it were a function (described here).

## Syntax

```
confirmedTracks = tracker(detections,time)
confirmedTracks = tracker(detections,time,costMatrix)
confirmedTracks = tracker(_____,detectableTrackIDs)
[confirmedTracks,tentativeTracks,allTracks] = tracker(_____)
[confirmedTracks,tentativeTracks,allTracks,analysisInformation] = tracker(_____
)
```

### Description

`confirmedTracks = tracker(detections,time)` returns a list of confirmed tracks that are updated from a list of detections, `detections`, at the update time, `time`. Confirmed tracks are corrected and predicted to the update time.

`confirmedTracks = tracker(detections,time,costMatrix)` also specifies a cost matrix, `costMatrix`.

To enable this syntax, set the `HasCostMatrixInput` property to `true`.

`confirmedTracks = tracker(_____,detectableTrackIDs)` also specifies a list of expected detectable tracks, `detectableTrackIDs`.

To enable this syntax, set the `HasDetectableTrackIDsInput` property to `true`.

`[confirmedTracks,tentativeTracks,allTracks] = tracker(_____)` also returns a list of tentative tracks, `tentativeTracks`, and a list of all tracks, `allTracks`.

`[confirmedTracks,tentativeTracks,allTracks,analysisInformation] = tracker(_____)` also returns information, `analysisInformation`, which can be used for track analysis.

### 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 update time, `time`, and greater than the previous time value used to update the tracker. Also, the `Time` differences between different `objectDetection` objects in the cell array do not need to be equal.

#### **time — Time of update**

scalar

Time of update, specified as a 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 tracker.

Data Types: `single` | `double`

#### **costMatrix — Cost matrix**

real-valued  $N$ -by- $M$  matrix

Cost matrix, specified as a real-valued  $N$ -by- $M$  matrix, where  $N$  is the number of existing tracks, and  $M$  is the number of current detections. The cost matrix rows must be in the same order as the list of tracks. The columns must be in the same order as the list of detections. Obtain the correct order of the list of tracks from the third output argument, `allTracks`, when is the tracker is updated.

At the first update of the object or when the tracker has no previous tracks, specify the cost matrix to have a size of `[0, numDetections]`. Note that the cost must be calculated so that lower costs indicate a higher likelihood of assigning a detection to a track. To prevent certain detections from being assigned to certain tracks, set the appropriate cost matrix entry to `Inf`.

#### **Dependencies**

To enable this argument, set the `HasCostMatrixInput` property to `true`.

Data Types: `double` | `single`

#### **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 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`**analysisInformation — Additional information for analyzing track updates**

structure

Additional information for analyzing track updates, returned as a structure. The fields of this structure are:

Field	Description
<code>TrackIDsAtStepBeginning</code>	Track IDs when step began
<code>CostMatrix</code>	Cost of assignment matrix
<code>Assignments</code>	Assignments returned from the assignment function.
<code>UnassignedTracks</code>	IDs of unassigned tracks returned from the tracker
<code>UnassignedDetections</code>	Indices of unassigned detections in the <code>detections</code> input.
<code>InitiatedTrackIDs</code>	IDs of tracks initiated during the step
<code>DeletedTrackIDs</code>	IDs of tracks deleted during the step
<code>TrackIDsAtStepEnd</code>	Track IDs when the step ended

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 trackerGNN

getTrackFilterProperties	Obtain track filter properties
setTrackFilterProperties	Set track filter properties
predictTrackToTime	Predict track state
initializeTrack	Initialize new track
deleteTrack	Delete existing track

## Common to All System Objects

release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isLocked	Determine if System object is in use
clone	Create duplicate System object

## Examples

### Track Two Objects Using trackerGNN

Construct a `trackerGNN` object with the default 2-D constant-velocity Kalman filter initialization function, `initcvkf`.

```
tracker = trackerGNN('FilterInitializationFcn', @initcvkf, ...
    'ConfirmationThreshold', [4 5], ...
    'DeletionThreshold', 10);
```

Update the tracker with two detections both having nonzero `ObjectClassID`. These detections immediately create confirmed tracks.

```
detections = {objectDetection(1,[10;0], 'SensorIndex',1, ...
    'ObjectClassID',5, 'ObjectAttributes',{struct('ID',1)}); ...
    objectDetection(1,[0;10], 'SensorIndex',1, ...
    'ObjectClassID',2, 'ObjectAttributes',{struct('ID',2)}});
time = 2;
tracks = tracker(detections,time);
```

Find the positions and velocities.

```
positionSelector = [1 0 0 0; 0 0 1 0];
velocitySelector = [0 1 0 0; 0 0 0 1];

positions = getTrackPositions(tracks,positionSelector)
positions = 2x2
```

```
10      0
 0      10
```

```
velocities = getTrackVelocities(tracks,velocitySelector)
```

```
velocities = 2x2
```

```
0      0
 0      0
```

## Algorithms

### Tracker Logic Flow

When a GNN tracker processes detections, track creation and management follow these steps.

- 1 The tracker divides detections by originating sensor.
- 2 For each sensor:
  - a The tracker calculates the distances from detections to existing tracks and forms a `costMatrix`.
  - b Based on the costs, the tracker performs global nearest neighbor assignment using the algorithm specified in the `Assignment` property.
  - c The assignment algorithm divides the detections and tracks into three groups:
    - Assigned one-to-one detection and track pairs
    - Unassigned detections
    - Unassigned tracks
- 3 Unassigned detections initialize new tracks. Using the unassigned detection, the tracker initializes a new track filter specified by the `FilterInitializationFcn` property. The track logic for the new track is initialized as well.

The tracker checks if any of the unassigned detections from other sensors can be assigned to the new track. If so, the tracker updates the new track with the assigned detections from the other sensors. As a result, these detections no longer initialize new tracks.

- 4 The pairs of assigned tracks and detections are used to update each track. The track filter is updated using the `correct` method provided by the specified tracking filter. Also, the track logic is updated with a 'hit'. The tracker checks if the track meets the criteria for confirmation. If so, the tracker confirms the track and sets the `IsCoasted` property to `false`.
- 5 Unassigned tracks are updated with a 'miss' and their `IsCoasted` flag is set to `true`. The tracker checks if the track meets the criteria for deletion. If so, the tracker removes the track from the maintained track list.
- 6 All tracks are predicted to the latest time value (either the time input if provided, or the latest mean cluster time stamp).

## References

- [1] Blackman, S., and R. Popoli. *Design and Analysis of Modern Tracking Systems*. Artech House Radar Library, Boston, 1999.

## 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 a multi-object tracker must have properties with the same sizes and types.
- If you use the **ObjectAttributes** field within an **objectDetection** object, you must specify this field as 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.
- If **ObjectAttributes** are contained in the detection, the **SensorIndex** value of the detection cannot be greater than 10.
- The first update to the multi-object tracker must contain at least one detection.

## See Also

### Functions

`assignTOMHT | assignauction | assignjv | assignkbest | assignkbestsd | assignmunkres | assignsd | clusterTrackBranches | compatibleTrackBranches | fusecovint | fusecovunion | fusexcov | getTrackPositions | getTrackVelocities | pruneTrackBranches | triangulateLOS`

### Objects

`objectDetection | objectTrack | trackHistoryLogic | trackScoreLogic | trackingABF | trackingCKF | trackingEKF | trackingGSF | trackingIMM | trackingKF | trackingMSCEKF | trackingPF | trackingUKF`

### System Objects

`staticDetectionFuser | trackerJPDA | trackerTOMHT`

### Topics

“Introduction to Multiple Target Tracking”  
“Introduction to Assignment Methods in Tracking Systems”

### Introduced in R2018b

# trackerGridRFS

Grid-based multi-object tracker

## Description

The `trackerGridRFS` System object is a tracker capable of processing detections of multiple targets from multiple sensors in a 2-D environment. The tracker tracks dynamic objects around an autonomous system using high resolution sensor data such as point clouds and radar detections. The tracker uses the random finite set (RFS) based approach combined with Dempster-Shafer approximations defined in [1] to estimate the dynamic characteristics of the grid cells. To extract objects from the grid, the tracker uses a cell-to-track association scheme [2]. For more details, see “Algorithms” on page 3-430.

To track targets using this object:

- 1 Create the `trackerGridRFS` 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 = trackerGridRFS  
tracker = trackerGridRFS(Name,Value)
```

#### Description

`tracker = trackerGridRFS` creates a `trackerGridRFS` System object with default property values.

`tracker = trackerGridRFS(Name,Value)` sets properties for the tracker using one or more name-value pairs. For example, `trackerGridRFS('MaxNumTracks',100)` creates a grid-based multi-object tracker that allows 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](#).

## Tracker Configuration

### 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. You must specify this property as a positive integer to use the track outputs as inputs to a track fuser.

Example: 1

### SensorConfigurations — Configurations of tracking sensors

cell array of `trackingSensorConfiguration` objects

Configuration of tracking sensors, specified as a cell array of `trackingSensorConfiguration` objects. This property provides the tracking sensor configuration information, such as sensor detection limits, sensor resolution, and sensor mounting, to the tracker. There are no default values for the `SensorConfigurations` property, and you must specify the `SensorConfigurations` property before using the tracker. You can update the configuration, if the `HasSensorConfigurationsInput` property is set to `true`, by specifying the configuration input argument `configs`.

When specifying the `trackingSensorConfiguration` object, the following properties must be specified with these formats:

Property Name	Format
<code>SensorIndex</code>	Unique identifier of the sensor, specified as a positive integer.
<code>IsValidTime</code>	Indicate if the sensor data should be used to update tracks, specified as <code>true</code> or <code>false</code> .
<code>SensorTransformParameters</code>	Parameters of sensor transform function, specified as a $p$ -element array of measurement parameter structures. $p$ is the number of sensors. The structure should contain fields with the same names as the measurement parameters used in a measurement function, such as the <code>cvmeas</code> function.  The first structure must describe the transformation from the autonomous system to the sensor coordinates. The subsequent structure describes the transformation from the autonomous system to the tracking coordinate frame. If only one structure is provided, the tracker assumes tracking is performed in the coordinate frame of the autonomous system.
<code>SensorLimits</code>	Sensor detection limits, specified as a 2-by-2 matrix of scalars. The first row specifies the lower and upper limits of the azimuth angle in degrees. The second row specifies the lower and upper limits of the detection range in meters.

The tracker ignores the `FilterInitializationFcn`, `SensorTransformFcn`, and `MaxNumDetsPerObject` properties of the `trackingSensorConfiguration` object.

**HasSensorConfigurationsInput — Enable updating sensor configurations with time**  
false (default) | true

Enable updating sensor configurations with time, specified as `false` or `true`. Set this property to `true` if you want the configurations of the sensors updated with time. When this property is set to `true`, you must specify the configuration input `configs` when using this object.

Data Types: logical

**StateParameters — Parameters of track state reference frame**  
`struct([ ]) (default) | struct array`

Parameters of the track state reference frame, specified as a structure array. Use this property to define the track state reference frame and how to transform the track from the tracker coordinate system, the source system, to the fuser coordinate system.

**Tunable:** Yes

Data Types: `struct`

**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. `MaxNumSensors` must be greater than or equal to the largest value of `SensorIndex` found in all the sensor data and configurations used to update the tracker.

Data Types: `single` | `double`

**MaxNumTracks — Maximum number of tracks**  
100 (default) | positive integer

Maximum number of tracks that the tracker can maintain, specified as a positive integer.

Data Types: `single` | `double`

**Grid Definition**

**GridLength — x-direction dimension of grid**  
100 (default) | positive scalar

x-direction dimension of the grid in the local coordinates, specified as a positive scalar in meters.

**GridWidth — y-direction dimension of grid**  
100 (default) | positive scalar

y-direction dimension of the grid in the local coordinates, specified as a positive scalar in meters.

**GridResolution — Resolution of grid**  
1 (default) | positive scalar

Resolution of the grid, specified as a positive scalar. `GridResolution` represents the number of cells per meter of the grid for both the x- and y-direction of the grid.

**GridOriginInLocal — Location of grid origin in local coordinate frame**

[-50 -50] (default) | two-element vector of scalar

Location of the grid origin in the local coordinate frame, specified as a two-element vector of scalars in meters. The grid origin represents the bottom-left corner of the grid.

**Particle Filtering****MotionModel — Motion model for tracking**

'constant-velocity' (default) | 'constant-acceleration' | 'constant-turn-rate'

Motion model for tracking, specified as 'constant-velocity', 'constant-acceleration', or 'constant-turn-rate'. The particle state and object state for each motion model are:

MotionModel	Particle State	Object State
'constant-velocity'	[x; vx; y; vy]	[x; vx; y; vy; yaw; L; W]
'constant-acceleration'	[x; vx; ax; y; vy; ay]	[x; vx; ax; y; vy; ay; yaw; L; W]
'constant-turn-rate'	[x; vx; y; vy; w]	[x; vx; y; vy; w; yaw; L; W]

where:

- x — Position of the object in the x direction of the local tracking frame (m)
- y — Position of the object in the y direction of the local tracking frame (m)
- vx — Velocity of the object in the x direction of the local tracking frame (m/s)
- vy — Velocity of the object in the y direction of the local tracking frame (m/s)
- ax — Acceleration of the object in the x direction of the local tracking frame (m/s<sup>2</sup>)
- ay — Acceleration of the object in the y direction of the local tracking frame (m/s<sup>2</sup>)
- w — Yaw-rate of the object in the local tracking frame (degree/s)
- yaw — Yaw angle of the object in the local tracking frame (deg)
- L — Length of the object (m)
- W — Width of the object (m)

**VelocityLimits — Minimum and maximum velocity of objects**

[-10 10; -10 10] (default) | 2-by-2 matrix of scalar

Minimum and maximum velocity of objects, specified as a 2-by-2 matrix of scalars in m/s. The first row specifies the lower and upper velocity limits in the x-direction and the second row specifies the lower and upper velocity limits in the y-direction. The tracker uses these limits to sample new particles in the grid using a uniform distribution.

**AccelerationLimits — Minimum and maximum acceleration of objects**

[-5 5; -5 5] (default) | 2-by-2 matrix of scalar

Minimum and maximum acceleration of objects, specified as a 2-by-2 matrix of scalars in m/s<sup>2</sup>. The first row specifies the lower and upper acceleration limits in the x-direction and the second row specifies the lower and upper acceleration limits in the y-direction . The tracker uses these limits to sample new particles in the grid using a uniform distribution.

This property is only active when the `MotionModel` property is set to 'constant-acceleration'.

**TurnrateLimits — Minimum and maximum turn rate of objects**

[ -5; 5] (default) | two-element vector of scalar

Minimum and maximum turn rate of objects, specified a two-element vector of scalars in degree/s. The first element defines the minimum turn rate and the second element defines the maximum turn-rate.

This property is only active when the `MotionModel` property is set to 'constant-turnrate'.

**ProcessNoise — Process noise covariance**

$N$ -by- $N$  identity matrix (default) |  $N$ -by- $N$  matrix of scalar

Process noise covariance, specified as an  $N$ -by- $N$  matrix of scalars. This property specifies the process noise for positions of particles and the geometric centers of targets.

- When the `HasAdditiveProcessNoise` property is set to `true`, the process directly adds to the prediction model. In this case,  $N$  is equal to the dimension of the particle state.
- When the `HasAdditiveProcessNoise` property is set to `false`, define the process noise according to the selected motion model. The process noise is added to the higher order terms, such as the acceleration for the 'constant-acceleration' model.

MotionModel	Number of Terms ( $N$ )	Meaning of Terms
'constant-velocity'	2	Acceleration in the x and y directions
'constant-acceleration'	2	Jerk in the x and y directions
'constant-turn-rate'	3	Acceleration in the x and y directions as well as the angular acceleration

Example: [1.0 0.05; 0.05 2]

**HasAdditiveProcessNoise — Enable to model process noise as additive**

false (default) | true

Enable to model process noise as additive, specified as `true` or `false`. When this property is `true`, process noise is added directly to the state vector. Otherwise, noise is incorporated in the motion model.

Example: true

**NumParticles — Number of particles per grid**

10000 (default) | positive integer

Number of particles per grid, specified as a positive integer. A higher number of particles can improve estimation quality, but can increase computational cost.

**NumBirthParticles — Number of newborn particles per time step**

1000 (default) | positive scalar

Number of newborn (initialized) particles per time step, specified as a positive integer. The tracker determines the locations of these new-born particles by using the mismatch between the predicted

and the updated occupancy belief masses and the **BirthProbability** property. A reasonable value of the **NumBirthParticles** property is approximately ten percent of the number of particles specified by the **NumParticles** property.

**BirthProbability — Probability of target birth in a cell per step**

0.01 (default) | scalar in [0 1)

Probability of target birth in a cell per step, specified as a scalar in the range [0 1). The birth probability controls the probability that new particles are generated in a cell.

Example: 1e-4

**DeathRate — Death rate of particles per unit time**

1e-3 (default) | positive scalar

Death rate of particles per unit time, specified as a positive scalar. Death rate indicates the possibility that a particle or target vanishes after one time step. Death rate influences the survival probability ( $P_s$ ) of a component between successive time steps as:

$$P_s = (1 - \text{DeathRate})^{\Delta T}$$

where  $\Delta T$  is the time step.

Example: 1e-4

**FreeSpaceDiscountFactor — Confidence in free space prediction**

0.8 (default) | scalar

Confidence in free space prediction, specified as a scalar. In the prediction stage of the tracker, the belief mass of a cell to be in the "free" (unoccupied) state is reduced by the **FreeSpaceDiscountFactor**:

$$m_k | k - 1(F) = \alpha^{\Delta T} m_{k - 1}(F)$$

where  $k$  is the time step index,  $m$  is the belief mass,  $\alpha$  is the free space discount factor, and  $\Delta T$  is the time step.

**Track Initialization**

**Clustering — Clustering method used for new object extraction**

'DBSCAN' (default) | 'Custom'

Clustering method used for new object extraction, specified as 'DBSCAN' or 'Custom'.

- 'DBSCAN' — Cluster unassigned dynamic grid cells using the density-based spatial clustering of applications with noise (DBSCAN) algorithm. You can configure the DBSCAN algorithm by specifying the **ClusteringThreshold** and **MinNumCellsPerCluster** properties of the tracker.
- 'Custom' — Cluster unassigned dynamic grid cells using a custom clustering function specified in the **CustomClusteringFcn** property of the tracker.

**ClusteringThreshold — Threshold for DBSCAN clustering**

5 (default) | positive scalar

Threshold for DBSCAN clustering, specified as a positive scalar.

To enable this property, set the *Clustering* property to 'DBSCAN'.

**CustomClusteringFcn — Custom function for clustering unassigned grid cells**

function handle

Custom function for clustering unassigned grid cells, specified as a function handle. The function must support this signature:

```
function indices = myFunction(dynamicGridCells)
```

where *dynamicGridCells* is a structure defining a set of grid cells initializing the track. It must have these fields:

Field	Description
Width	Width of the cell, specified as a positive scalar.
GridIndices	Indices of the grid cells, specified as an <i>N</i> -by-2 array, where <i>N</i> is the number of unassigned cells. The first element specifies the grid index in the <i>x</i> -direction and the second element specifies the grid index in the <i>y</i> -direction.
State	States of the grid cells, specified as a <i>P</i> -by- <i>N</i> array of scalars, where <i>P</i> is the dimension of the state and <i>N</i> is the number of unassigned cells.
StateCovariance	State covariances of the grid cells, specified as a <i>P</i> -by- <i>P</i> -by- <i>N</i> array of scalars, where <i>P</i> is the dimension of the state and <i>N</i> is the number of unassigned cells.
OccupancyMass	Occupancy belief mass of the cells, specified as an <i>N</i> -element array of scalars, where <i>N</i> is the number of unassigned cells.
FreeMass	Free belief mass of the cells, specified as an <i>N</i> -element array of scalars, where <i>N</i> is the number of unassigned cells.

*indices* must be returned as an *N*-element vector of indices defining the cluster index for each dynamic grid cell.

To enable this property, set the *Clustering* property to 'Custom'.

**MinNumCellsPerCluster — Minimum number of cells per cluster for DBSCAN**

2 (default) | positive integer

Minimum number of cells per cluster for DBSCAN, specified as a positive scalar. This property affects whether a point is a core point in the DBSCAN algorithm.

To enable this property, set the *Clustering* property to 'DBSCAN'.

**TrackInitializationFcn — Function to initialize new track**

'trackerGridRFS.defaultTrackInitialization' (default) | function handle

Function to initialize new tracks, specified as a function handle. The initialization function initiates a track from a set of dynamic grid cells.

The default initialization function merges the Gaussian estimate from each cell to describe the state of the object. The orientation of the object is aligned with the direction of its mean velocity. With a defined orientation, the length and width of the object are extracted using the geometric properties of the cells. The object calculates uncertainties in length, width, and orientation estimates are calculated using linear approximations.

If you choose to customize your own initialization function, the function must support the following signature:

```
function track = myFunction(dynamicGridCells)
```

where `dynamicGridCells` is a structure defining a set of grid cells initializing the track. It has the following fields:

Field	Description
<code>Width</code>	Width of the cell, specified as a positive scalar.
<code>GridIndices</code>	Indices of the grid cells, specified as an $N$ -by-2 array, where $N$ is the number of unassigned cells. The first element specifies the grid index in the $x$ -direction and the second element specifies the grid index in the $y$ -direction.
<code>State</code>	States of the grid cells, specified as a $P$ -by- $N$ array of scalars, where $P$ is the dimension of the state and $N$ is the number of unassigned cells.
<code>StateCovariance</code>	State covariances of the grid cells, specified as a $P$ -by- $P$ -by- $N$ array of scalars, where $P$ is the dimension of the state and $N$ is the number of unassigned cells.
<code>OccupancyMass</code>	Occupancy belief mass of the cells, specified as an $N$ -element array of scalars, where $N$ is the number of unassigned cells.
<code>FreeMass</code>	Free belief mass of the cells, specified as an $N$ -element array of scalars, where $N$ is the number of unassigned cells.

`track` must be returned as an `objectTrack` object or a structure whose field names are the same as the property names of an `objectTrack` object. The dimension of the state must be the same as the state dimension specified in the `MotionModel` property.

**Track Management****TrackUpdateFcn — Function to update existing track**

'trackerGridRFS.defaultTrackUpdate' (default) | function handle

Function to update an existing track using its associated set of dynamic grid cells, specified as a function handle.

The default update function updates the `State` and `StateCovariance` properties of the track using the new estimate from the dynamic grid cells associated with the track. The update process is similar to the initialization process for the `TrackInitializationFcn` property. The tracker does not apply filtering to the state and state covariance.

If you choose to customize your own update function, the function must support this signature:

```
function updatedTrack = TrackUpdateFcn(predictedTrack,dynamicGridCells)
```

where:

- `predictedTrack` is the predicted track of an object, specified as an `objectTrack` object.
- `dynamicGridCells` is a structure defining a set of dynamic grid cells associated to the track. The structure has these fields:

Field	Description
<code>Width</code>	Width of the cell, specified as a positive scalar.
<code>GridIndices</code>	Indices of the grid cells, specified as an $N$ -by-2 array, where $N$ is the number of unassigned cells. The first element specifies the grid index in the x-direction and the second element specifies the grid index in the y-direction.
<code>State</code>	States of the grid cells, specified as a $P$ -by- $N$ array of scalars, where $P$ is the dimension of the state and $N$ is the number of unassigned cells.
<code>StateCovariance</code>	State covariances of the grid cells, specified as a $P$ -by- $P$ -by- $N$ array of scalars, where $P$ is the dimension of the state and $N$ is the number of unassigned cells.
<code>OccupancyMass</code>	Occupancy belief mass of the cells, specified as an $N$ -element array of scalars, where $N$ is the number of unassigned cells.
<code>FreeMass</code>	Free belief mass of the cells, specified as an $N$ -element array of scalars, where $N$ is the number of unassigned cells.

- `updatedTrack` is the updated track, returned as an `objectTrack` object or a structure whose field names are the same as the property names of an `objectTrack` object.

#### **AssignmentThreshold — Threshold for assigning dynamic grid cells to tracks**

30 (default) | positive scalar

Threshold for assigning dynamic grid cells to tracks, specified as a positive scalar. A dynamic grid cell can only be associated to a track if its distance (represented by the negative log-likelihood) to the track is less than the `AssignmentThreshold` value.

- Increase the threshold if a dynamic cell is not being assigned to a track that it should be assigned to.
- Decrease the threshold if there are dynamic cells being assigned to a track that they should not be assigned to.

Example: 18.1

**ConfirmationThreshold — Threshold for track confirmation**

[2 3] (default) | 2-element vector of scalar

Threshold for track confirmation, specified as a 2-element vector of scalars [M N]. A track is confirmed if it has been assigned to any dynamic grid cell in at least M updates of the last N updates.

**DeletionThreshold — Threshold for track deletion**

[5 5] (default) | 2-element vector of scalar

Threshold for track deletion, specified as a 2-element vector of scalars [P R]. A track is deleted if has not been assigned to any dynamic grid cell in at least P updates of the last R updates.

Example: 0.01

Data Types: single | double

**NumTracks — Number of tracks maintained by tracker**

nonnegative integer

This property is read-only.

Number of tracks maintained by the tracker, returned as a nonnegative integer.

Data Types: double

**NumConfirmedTracks — Number of confirmed tracks**

nonnegative integer

This property is read-only.

Number of confirmed tracks, returned as a nonnegative integer. If the IsConfirmed property of an output track object is true, the track is confirmed.

Data Types: double

**UseGPU — Enable using GPU for estimation of dynamic grid map**

false (default) | true

This property is read-only.

Enable using GPU for estimation of the dynamic grid map, specified as true or false. Enabling GPU computation requires the Parallel Computing Toolbox.

## Usage

### Syntax

```
confirmedTracks = tracker(sensorData,time)
confirmedTracks = tracker(sensorData,configs,time)
[confirmedTracks,tentativeTracks,allTracks] = tracker(__)
```

## Description

`confirmedTracks = tracker(sensorData, time)` returns a list of confirmed tracks that are updated from a list of sensor data `sensorData` at the update time `time`. Confirmed tracks are corrected and predicted to the update time.

`confirmedTracks = tracker(sensorData, configs, time)` also specifies the configurations of sensors `configs`. To enable this syntax, set the `HasSensorConfigurationsInput` property to `true`.

`[confirmedTracks, tentativeTracks, allTracks] = tracker(____)` also returns a list of tentative tracks `tentativeTracks` and a list of all tracks `allTracks`. You can use any combination of input arguments from previous syntaxes.

## Input Arguments

### **sensorData — Sensor data**

*N*-element array of structure

Sensor data, specified as an *N*-element array of structures. Each structure must define the measurement from a high resolution sensor using the these fields:

Fields	Description
Time	Time at which the sensor reports the data, specified as a nonnegative scalar.
SensorIndex	Unique identifier of the sensor, specified as a positive integer.
Measurement	Measurements of the sensor, specified a <i>K</i> -by- <i>M</i> matrix of scalars. <i>K</i> is the dimension of measurements, and <i>M</i> is the number of measurements. Each measurement defines the positional aspects of the detection in a rectangular or spherical frame.
MeasurementParameters	Measurement parameters, specified as a structure describing the transformation from the particle state to measurement. See “Object Detections” on page 3-111 for more details.

The `Time` value must be less than or equal to the current update time, `time`, and greater than the previous time value used to update the tracker.

### **time — Time of update**

scalar

Time of update, specified as a scalar. The tracker updates all tracks to this time. Units are in seconds.

`time` must be greater than or equal to the largest `Time` field value of the `sensorData` structures. `time` must increase in value with each update to the tracker.

Data Types: `single` | `double`

### **configs — Sensor configurations**

array of structures | cell array of structures | cell array of `trackingSensorConfiguration` objects

Sensor configurations, specified as an array of structures, a cell array of structures, or a cell array of `trackingSensorConfiguration` objects. If you specify the value using an array of structures or a cell array of structures, you must include `SensorIndex` as a field in each structure. The other optional fields in each structure must have the same names as the `trackingSensorConfiguration` object properties. You only need to specify sensor configurations that need to be updated. For example, if you want to update the `IsValidTime` property for only the fifth sensor, specify `configs` as `struct('SensorIndex',5,'IsValidTime',false)`.

### **Dependencies**

To enable this argument, set the `HasSensorConfigurationsInput` property to `true`.

### **Output Arguments**

#### **confirmedTracks — Confirmed tracks**

array of `objectTrack` objects

Confirmed tracks updated to the current time, returned as an array of `objectTrack` objects, where each element represents the track of an object. The state form of each track follows the form specified in the `MotionModel` property.

#### **tentativeTracks — Tentative tracks**

array of `objectTrack` objects

Tentative tracks, returned as an array of `objectTrack` objects, where each element represents the track of an object. The state form of each track follows the form specified in the `MotionModel` property.

#### **allTracks — All tracks**

structure | array of objects

All tracks, returned as an array of `objectTrack` objects, where each element represents the track of an object. The state form of each track follows the form specified in the `MotionModel` property.

## **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 trackerGridRFS**

`predictTracksToTime` Predict track state

`showDynamicMap` Plot dynamic occupancy grid map

### **Common to All System Objects**

`step` Run `System` object algorithm

`release` Release resources and allow changes to `System` object property values and input characteristics

`isLocked` Determine if `System` object is in use

`clone` Create duplicate `System` object

`reset` Reset internal states of `System` object

## Examples

### Track Targets Using Grid-Based Tracker

Create a tracking scenario.

```
scene = trackingScenario('UpdateRate',5,'StopTime',5);
rng(2021); % For reproducible results
```

Add a platform with a mounted lidar sensor to the tracking scenario.

```
plat = platform(scene);
lidar = monostaticLidarSensor(1,'DetectionCoordinates','Body','HasOrganizedOutput',false);
```

Add two targets with random positions and velocities to the scenario. Also, define the trajectory, mesh, and dimension of each platform.

```
for i = 1:2
    target = platform(scene);
    x = 50*(2*rand - 1);
    y = 50*(2*rand - 1);
    vx = 5*(2*rand - 1);
    vy = 5*(2*rand - 1);
    target.Trajectory.Position = [x y 0];
    target.Trajectory.Velocity = [vx vy 0];
    % Align the orientation of the target with the direction of motion.
    target.Trajectory.Orientation = quaternion([atan2d(vy,vx),0,0],'eulerd','ZYX','frame');
    target.Mesh = extendedObjectMesh('sphere');
    target.Dimensions = struct('Length',4,'Width',4,'Height',2,'OriginOffset',[0 0 0]);
end
```

Define the configuration of the lidar sensor.

```
config = trackingSensorConfiguration(1,...
    'SensorLimits',[-180 180;0 100],...
    'SensorTransformParameters',struct,...
    'IsValidTime',true);
```

Create a grid-based tracker.

```
tracker = trackerGridRFS('SensorConfigurations',config,...%
    'AssignmentThreshold',5,...%
    'MinNumCellsPerCluster',4,...%
    'ClusteringThreshold',3);
```

Define a `theaterPlot` object and two associated plotters for visualizing the tracking scene.

```
tp = theaterPlot('XLimits',[-50 50], 'YLimits',[-50 50]);
trkPlotter = trackPlotter(tp,'DisplayName','Tracks','MarkerFaceColor','g');
tthPlotter = platformPlotter(tp,'DisplayName','Truths','MarkerFaceColor','r','ExtentAlpha',0.2);
```

Advance the scenario and run the tracker with the lidar data.

```
while advance(scene)
    time = scene.SimulationTime;
```

```
% Generate point cloud.
tgtMeshes = targetMeshes(plat);
[ptCloud, config] = lidar(tgtMeshes, time);

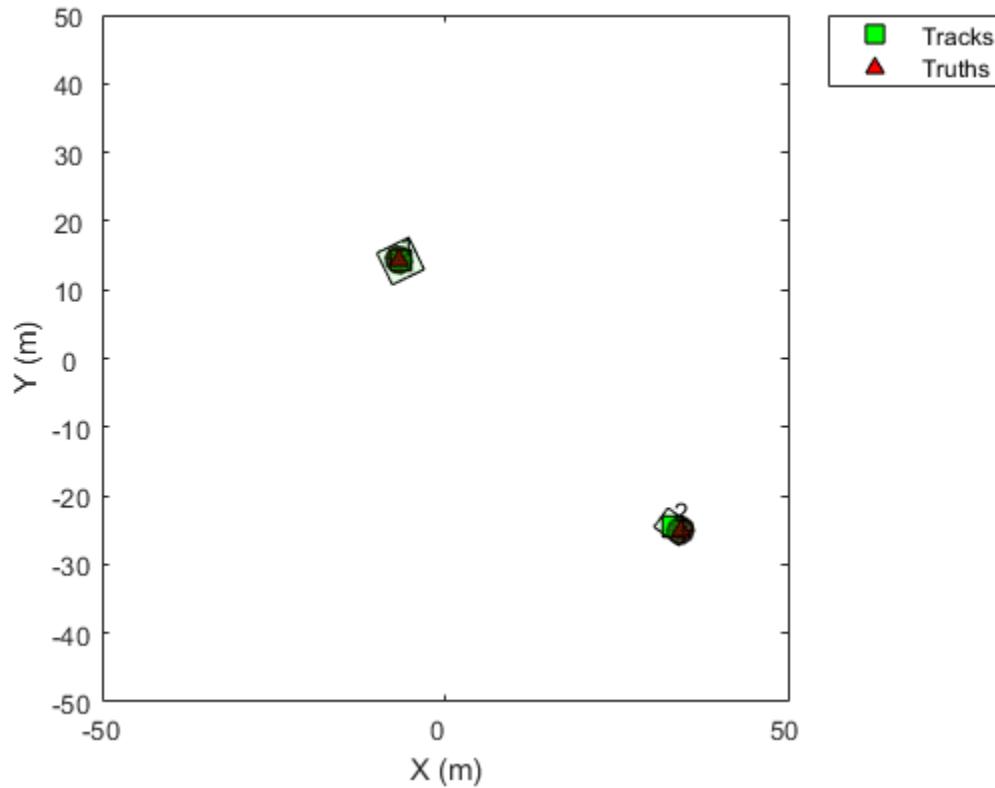
% Format the data for the tracker.
sensorData = struct('Time',time,...  
    'SensorIndex',1,...  
    'Measurement',ptCloud,...  
    'MeasurementParameters',struct...  
    );

% Update the tracker using the sensor data.
tracks = tracker(sensorData, time);

% Visualize tracks.
pos = zeros(numel(tracks),3);
vel = zeros(numel(tracks),3);
orient = quaternion.ones(numel(tracks),1);
dim = repmat(plat.Dimensions,numel(tracks),1);
ids = string([tracks.TrackID]);

for i = 1:numel(tracks)
    vel(i,:) = [tracks(i).State(2);tracks(i).State(4);0];
    pos(i,:) = [tracks(i).State(1);tracks(i).State(3);0];
    dim(i).Length = tracks(i).State(6);
    dim(i).Width = tracks(i).State(7);
    orient(i) = quaternion([tracks(i).State(5) 0 0], 'eulerd', 'ZYX', 'frame');
end
trkPlotter.plotTrack(pos,dim,orient,ids);

% Visualize platforms.
pos = vertcat(tgtMeshes.Position);
meshes = vertcat(tgtMeshes.Mesh);
orient = vertcat(tgtMeshes.Orientation);
tthPlotter.plotPlatform(pos,meshes,orient);
end
```



## Algorithms

### Tracker Logic Flow

The `trackerGridRFS` system object initializes, confirms, and deletes tracks automatically by using this algorithm:

- 1 The tracker projects sensor data from all sensors on a two-dimensional grid map to represent the occupancy and free evidence in a Dempster-Shafer framework.
- 2 The tracker uses a particle-based approach to estimate the dynamic state of the 2-D grid. This helps the tracker to classify each cell as dynamic or static.
- 3 The tracker manages tracks based on this logic:
  - a The tracker associates each dynamic grid cell with the existing tracks using a gated nearest-neighbor approach.
  - b The tracker initializes new tracks using unassigned dynamic grid cells. A track is created with a `Tentative` status, and the status will change to `Confirmed` after enough updates. For more information, see the `ConfirmationThreshold` property.
  - c Alternatively, the tracker confirms a track immediately if the `ObjectClassID` of the track is set to a positive value after track initialization. For more information, see the `TrackInitializationFcn` property.

- d The tracker performs coasting, predicting unassigned tracks to the current time, and deletes tracks with more misses than allowed. For more information, see the `DeletionThreshold` property.

## References

- [1] Nuss, D., Reuter, S., Thom, M., Yuan, T., Krehl, G., Maile, M., Gern, A. and Dietmayer, K., 2018. A random finite set approach for dynamic occupancy grid maps with real-time application. *The International Journal of Robotics Research*, 37(8), pp.841-866.
- [2] Steyer, Sascha, Georg Tanzmeister, and Dirk Wollherr. "Object tracking based on evidential dynamic occupancy grids in urban environments." In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1064-1070. IEEE, 2017.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

[trackerPHD](#) | [trackingSensorConfiguration](#)

### Introduced in R2020b

# showDynamicMap

Plot dynamic occupancy grid map

## Syntax

```
showDynamicMap(tracker)
showDynamicMap(tracker,Name,Value)
```

## Description

`showDynamicMap(tracker)` plots the dynamic occupancy grid map in the local coordinates. The static cells are shown using grayscale images, in which the grayness represents the occupancy probability of the cell. The dynamic cells are shown using HSV (hue, saturation, and value) values on an RGB colormap:

- Hue — The orientation angle of the velocity vector divided by 360. As hue increases from 0 to 1, the color changes in the order of red to orange, yellow, green, cyan, blue, magenta, and back to red.
- Saturation — The Mahalanobis distance ( $d$ ) between the velocity distribution of the grid cell and the zero velocity. A cell with  $d > 4$  is drawn with full saturation.
- Value — The occupancy probability of the cell.

`showDynamicMap(tracker,Name,Value)` specifies options using one or more name-value pair arguments. Enclose each Name in quotes. For example, `showDynamicMap(myTracker,"PlotVelocity",true)` plots the dynamic map for `myTracker` with velocity plotting enabled.

## Examples

### Track Targets Using `trackerGridRFS` and Show Dynamic Map

Create a tracking scenario.

```
scene = trackingScenario('UpdateRate',5,'StopTime',5);
rng(2021); % For reproducible results
```

Add a platform with a mounted lidar sensor to the tracking scenario.

```
plat = platform(scene);
lidar = monostaticLidarSensor(1,'DetectionCoordinates','Body','HasOrganizedOutput',false);
```

Add two targets with random positions and velocities to the scenario. Also, define the trajectory, mesh, and dimension of each platform.

```
for i = 1:2
    target = platform(scene);
    x = 50*(2*rand - 1);
    y = 50*(2*rand - 1);
    vx = 5*(2*rand - 1);
```

```

vy = 5*(2*rand - 1);
target.Trajectory.Position = [x y 0];
target.Trajectory.Velocity = [vx vy 0];
% Align the orientation of the target with the direction of motion.
target.Trajectory.Orientation = quaternion([atan2d(vy,vx),0,0],'eulerd','ZYX','frame');
target.Mesh = extendedObjectMesh('sphere');
target.Dimensions = struct('Length',4,'Width',4,'Height',2,'OriginOffset',[0 0 0]);
end

```

Define the configuration of the lidar sensor.

```

config = trackingSensorConfiguration(1, ...
    'SensorLimits', [-180 180; 0 100], ...
    'SensorTransformParameters', struct, ...
    'IsValidTime', true);

```

Create a grid-based tracker.

```

tracker = trackerGridRFS('SensorConfigurations', config, ...
    'AssignmentThreshold', 5, ...
    'MinNumCellsPerCluster', 4, ...
    'ClusteringThreshold', 3);

```

Advance the scenario and run the tracker with the lidar data.

```

while advance(scene)

    time = scene.SimulationTime;

    % Generate point cloud.
    tgtMeshes = targetMeshes(plat);
    [ptCloud, config] = lidar(tgtMeshes, time);

    % Format the data for the tracker.
    sensorData = struct('Time', time, ...
        'SensorIndex', 1, ...
        'Measurement', ptCloud, ...
        'MeasurementParameters', struct...
    );
    % Update the tracker using the sensor data.
    tracks = tracker(sensorData, time);
end

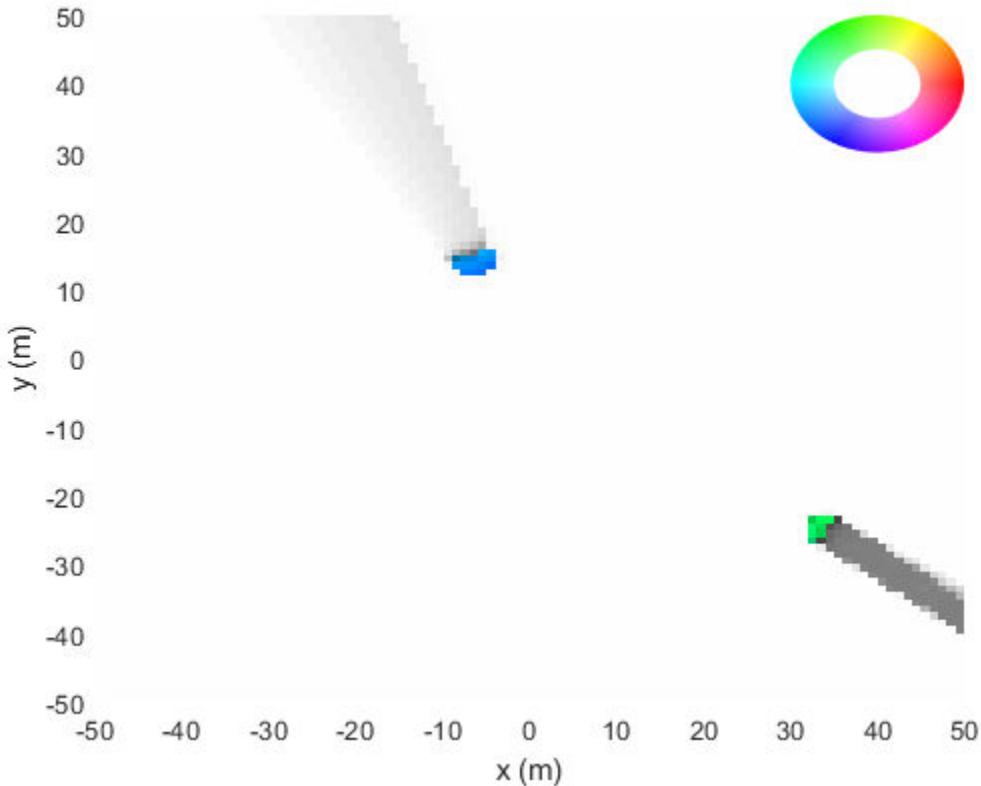
```

Show the dynamic map.

```

showDynamicMap(tracker)
xlabel('x (m)');
ylabel('y (m)');

```



## Input Arguments

**tracker — Grid-based RFS tracker**  
trackerGridRFS object

Grid-based RFS tracker, specified as a `trackerGridRFS` object.

## 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: `showDynamicMap(myTracker, "PlotVelocity", false)` plots the dynamic map for `myTracker` with velocity plotting enabled.

**PlotVelocity — Enable velocity plotting**  
`false` (default) | `true`

Enable velocity plotting, specified as `true` or `false`. When specified as `true`, the velocity vector for each dynamic cell is plotted at the center of the grid cell. The length of the plotted vector represents the magnitude of the velocity.

**Parent — Parent axes**  
`gca` | Axes handle

Parent axes on which to plot the map, specified as an `Axes` handle.

**FastUpdate — Enable updating from previous map**

`true` (default) | `false`

Enable updating from previous map, specified as `true` or `false`. When specified as `true`, the function plots the map via a lightweight update to the previous map in the figure. When specified as `false`, the function plots a new map on the figure every time.

**InvertColors — Enable inverted colors**

`false` (default) | `true`

Enable inverted colors on the map, specified as `true` or `false`. When specified as `false`, the function plots empty space in white and occupied space in black. When specified as `true`, the function plots empty space in black and occupied space in white.

**See Also**

`trackerGridRFS`

**Introduced in R2020b**

# trackerPHD

Multi-sensor, multi-object PHD tracker

## Description

The `trackerPHD` System object is a tracker capable of processing detections of multiple targets from multiple sensors. The tracker uses a multi-target probability hypothesis density (PHD) filter to estimate the states of point targets and extended objects. PHD is a function defined over the state-space of the tracking system, and its value at a state is defined as the expected number of targets per unit state-space volume. The PHD is represented by a weighted summation (mixture) of probability density functions, and peaks in the PHD correspond to possible targets. For an overview of how the tracker functions, see “Algorithms” on page 3-447.

By default, the `trackerPHD` can track extended objects using the `ggiwphd` filter, which models detections from an extended object as a parse points cloud. You can also use `trackerPHD` with the `gmphd` filters, which tracks point targets and extended objects with designated shapes. Inputs to the tracker are detection reports generated by `objectDetection`, `radarSensor`, `monostaticRadarSensor`, `irSensor`, or `sonarSensor` objects. The tracker outputs all maintained tracks and their analysis information.

To track targets using this object:

- 1 Create the `trackerPHD` 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 = trackerPHD  
tracker = trackerPHD(Name,Value)
```

## Description

`tracker = trackerPHD` creates a `trackerPHD` System object with default property values.

`tracker = trackerPHD(Name,Value)` sets properties for the tracker using one or more name-value pairs. For example, `trackerPHD('MaxNumTracks',100)` creates a PHD tracker that allows 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. You must specify this property as a positive integer to use the track outputs as inputs to a track fuser.

Example: 1

#### **SensorConfigurations — Configurations of tracking sensors**

cell array of `trackingSensorConfiguration` objects

Configuration of tracking sensors, specified as a cell array of `trackingSensorConfiguration` objects. This property provides the tracking sensor configuration information, such as sensor detection limits and sensor resolution, to the tracker. Note that there are no default values for the `SensorConfigurations` property, and you must specify the `SensorConfigurations` property before using the tracker. However, you can update the configuration by setting the `HasSensorConfigurationsInput` property to `true` and specifying the configuration input, `config`. If you set the `MaxDetsPerObject` property of the `trackingSensorConfiguration` object to 1, the tracker creates only one partition, such that at most one detection can be assigned to each target.

#### **PartitioningFcn — Function to partition detections into detection cells**

`@partitionDetections` (default) | function handle | character vector

Function to partition detections into detection cells, specified as a function handle or as a character vector. When each sensor can report more than one detection per object, a partition function is required. The partition function reports all possible partitions of the detections from a sensor. In each partition, the detections are separated into mutually exclusive detection cells, assuming that each detection cell belongs to one extended object.

You can also specify your own detections partition function. For guidance in writing this function, you can examine the details of the default partitioning function, `partitionDetections`, using the `type` command as:

```
type partitionDetections
```

Example: `@myfunction` or `'myfunction'`

Data Types: `function_handle` | `char`

#### **BirthRate — Birth rate of new targets in the density**

`1e-3` (default) | positive real scalar

Birth rate of new targets in the density, specified as a scalar. Birth rate indicates the expected number of targets added in the density per unit time. The birth density is created by using the `FilterInitializationFcn` of the `trackingSensorConfiguration` used with the tracker. In general, the tracker adds components to the density function in two ways:

- 1 Predictive birth density - density initialized by `FilterInitializationFcn` function when called with no inputs.
- 2 Adaptive birth density - density initialized by `FilterInitializationFcn` function when called with detections inputs. The detections are chosen by the tracker based on their log-likelihood of association with the current estimates of the targets.

Note that the value for the `BirthRate` property represents the summation of both predictive birth density and adaptive birth density for each time step.

Example: `0.01`

Data Types: `single | double`

**DeathRate — Death rate of components in the density**

`1e-6` (default) | positive real scalar

Death rate of components in the density, specified as a scalar. Death rate indicates the rate at which a component vanishes in the density after one time step. Death rate relates to the survival probability ( $P_s$ ) of a component between successive time steps by

$$P_s = (1 - \text{DeathRate})^{\Delta T}$$

where  $\Delta T$  is the time step.

Example: `1e-4`

Data Types: `single | double`

**AssignmentThreshold — Threshold of selecting detections for component initialization**

`25` (default) | real positive scalar

Threshold of selecting detections for component initialization, specified as a positive scalar. During correction, the tracker calculates the likelihood of association between existing tracks and detection cells. If the association likelihood (given by negative log-likelihood) of a detection cell to all existing tracks is higher than the threshold (which means the detection cell has low likelihood of association to existing tracks), the detection cell is used to initialize new components in the adaptive birth density.

Example: `18.1`

Data Types: `single | double`

**ExtractionThreshold — Threshold for initializing tentative track**

`0.5` (default) | real positive scalar

Threshold for initializing a tentative track, specified as a scalar. If the weight of a component is higher than the threshold specified by the `ExtractionThreshold` property, the component is labeled as a 'Tentative' track and given a `TrackID`.

Example: `0.45`

Data Types: `single | double`

**ConfirmationThreshold — Threshold for track confirmation**

`0.8` (default) | real positive scalar

Threshold for track confirmation, specified as a scalar. In a `trackerPHD` object, a track can have multiple components sharing the same `TrackID`. If the weight summation of a tentative track's components is higher than the threshold specified by the `ConfirmationThreshold` property, the track's status is marked as 'Confirmed'.

Example: 0.85

Data Types: `single` | `double`

#### **DeletionThreshold — Threshold for component deletion**

`1e-3` (default) | real positive scalar

Threshold for component deletion, specified as a scalar. In the PHD tracker, if the weight of a component is lower than the value specified by the `DeletionThreshold` property, the component is deleted.

Example: 0.01

Data Types: `single` | `double`

#### **MergingThreshold — Threshold for components merging**

`25` (default) | real positive scalar

Threshold for components merging, specified as a real positive scalar. In the PHD tracker, if the Kullback-Leibler distance between components with the same `TrackID` is smaller than the value specified by the `MergingThreshold` property, then these components are merged into one component. The merged weight of the new component is equal to the summation of the weights of the pre-merged components. Moreover, if the merged weight is higher than the first threshold specified in the `LabelingThresholds` property, the merged weight is truncated to the first threshold. Note that components with `TrackID` equal to 0 can also be merged with each other.

Example: 30

Data Types: `single` | `double`

#### **LabelingThresholds — Thresholds for label management**

`[1.1 1 0.8]` (default) | 1-by-3 vector of positive values

Labeling thresholds, specified as an 1-by-3 vector of decreasing positive values,  $[C_1, C_2, C_3]$ . Based on the `LabelingThresholds` property, the tracker manages components in the density using these rules:

- 1 The weight of any component that is higher than the first threshold  $C_1$  is reduced to  $C_1$ .
- 2 For all components with the same `TrackID`, if the largest weight among these components is greater than  $C_2$ , then the component with the largest weight is preserved to retain the `TrackID`, while all other components are deleted.
- 3 For all components with the same `TrackID`, if the ratio of the largest weight to the weight summation of all these components is greater than  $C_3$ , then the component with the largest weight is preserved to retain the `TrackID`, while all other components are deleted.
- 4 If neither condition 2 nor condition 3 is satisfied, then the component with the largest weight retains the `TrackID`, while the labels of all other components are set to 0. When this occurs, it essentially means that some components may represent other objects. This treatment keeps the possibility for these unreserved components to be extracted again in the future.

Data Types: `single` | `double`

**HasSensorConfigurationsInput** — **Enable updating sensor configurations with time**  
false (default) | true

Enable updating sensor configurations with time, specified as `false` or `true`. Set this property to `true` if you want the configurations of the sensor updated with time. Also, when this property is set to `true`, the tracker must be called with the configuration input, `config`, as shown in the usage syntax.

Data Types: logical

**StateParameters** — **Parameters of track state reference frame**  
`struct([ ])` (default) | `struct` array

Parameters of the track state reference frame, specified as a structure array. Use this property to define the track state reference frame and how to transform the track from the tracker coordinate system, the source system, to the fuser coordinate system.

**Tunable:** Yes

Data Types: `struct`

**NumTracks** — **Number of tracks maintained by tracker**  
nonnegative integer

This property is read-only.

Number of tracks maintained by the tracker, returned as a nonnegative integer.

Data Types: `double`

**NumConfirmedTracks** — **Number of confirmed tracks**  
nonnegative integer

This property is read-only.

Number of confirmed tracks, returned as a nonnegative integer. If the `IsConfirmed` field of an output track structure is `true`, the track is confirmed.

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. `MaxNumSensors` must be greater than or equal to the largest value of `SensorIndex` found in all the detections used to update the tracker. `SensorIndex` is a property of an `objectDetection` object.

Data Types: `single` | `double`

**MaxNumTracks** — **Maximum number of tracks**  
100 (default) | positive integer

Maximum number of tracks that the tracker can maintain, specified as a positive integer.

Data Types: `single` | `double`

## Usage

To process detections and update tracks, call the tracker with arguments, as if it were a function (described here).

## Syntax

```
confirmedTracks = tracker(detections, time)
confirmedTracks = tracker(detections, config, time)
[confirmedTracks, tentativeTracks, allTracks] = tracker(____)
[confirmedTracks, tentativeTracks, allTracks, analysisInformation] = tracker(____)
```

### Description

`confirmedTracks = tracker(detections, time)` returns a list of confirmed tracks that are updated from a list of detections, `detections`, at the update time, `time`. Confirmed tracks are corrected and predicted to the update time.

`confirmedTracks = tracker(detections, config, time)` also specifies a sensor configuration input, `config`. Use this syntax when the configurations of sensors are changing with time. To enable this syntax, set the `HasSensorConfigurationsInput` property to `true`.

`[confirmedTracks, tentativeTracks, allTracks] = tracker(____)` also returns a list of tentative tracks, `tentativeTracks`, and a list of all tracks, `allTracks`. You can use this output syntax with any of the previous input syntaxes.

`[confirmedTracks, tentativeTracks, allTracks, analysisInformation] = tracker(____)` also returns the analysis information, `analysisInformation`, which can be used for track analysis. You can use this output syntax with any of the previous input syntaxes.

### 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 update time, `time`, and greater than the previous time value used to update the tracker. Also, the `Time` differences between different `objectDetection` objects in the cell array do not need to be equal.

#### **time — Time of update**

scalar

Time of update, specified as a 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 tracker.

Data Types: `single` | `double`

#### **config — Sensor configurations**

array of structs | cell array of structs | cell array of `trackingSensorConfiguration` objects

Sensor configurations, specified as an array of structs, a cell array of structs, or a cell array of `trackingSensorConfiguration` objects. If you specify the value using an array of structs or a cell array of structs, you must include `SensorIndex` as a field for each struct. The other optional fields in each struct must have the same name as one of the properties of the `trackingSensorConfiguration` object. Note that you only need to specify sensor configurations that need to be updated. For example, if you only want to update the `IsValidTime` property for the fifth sensor, provide the value for `config` as `struct('SensorIndex',5,'IsValidTime',false)`.

#### Dependencies

To enable this argument, set the `HasSensorConfigurationsInput` property to `true`.

#### Output Arguments

##### **confirmedTracks — Confirmed tracks**

structure | array of structures

Confirmed tracks updated to the current time, returned as a structure or an array of structures. Each structure corresponds to a track. A track is confirmed if the weight summation of its components is above the threshold specified by the `ConfirmationThreshold` property. If a track is confirmed, the `IsConfirmed` field of the structure is `true`. The fields of the confirmed tracks structure are defined in "Track Structure" on page 3-446.

Data Types: `struct`

##### **tentativeTracks — Tentative tracks**

structure | array of structures

Tentative tracks, returned as a structure or an array of structures. Each structure corresponds to a track. A track is tentative if the weight summation of its components is above the threshold specified by the `ExtractionThreshold` property, but below the threshold specified by the `ConfirmationThreshold` property. In that case, the `IsConfirmed` field of the structure is `false`. The fields of the structure are defined in "Track Structure" on page 3-446.

Data Types: `struct`

##### **allTracks — All tracks**

structure | array of structures

All tracks, returned as a structure or an array of structures. Each structure corresponds to a track. The set of all tracks consists of confirmed and tentative tracks. The fields of the structure are defined in "Track Structure" on page 3-446.

Data Types: `struct`

##### **analysisInformation — Additional information for analyzing track updates**

structure

Additional information for analyzing track updates, returned as a structure. The fields of this structure are:

Field	Description
<code>CorrectionOrder</code>	The order in which sensors are used for state estimate correction, returned as a row vector of <code>SensorIndex</code> . For example, [1 3 2 4].

<b>TrackIDsAtStepBeginning</b>	Track IDs when step began.
<b>DeletedTrackIDs</b>	IDs of tracks deleted during the step.
<b>TrackIDsAtStepEnd</b>	Track IDs when the step ended.
<b>SensorAnalysisInfo</b>	Cell array of sensor analysis information.

The **SensorAnalysisInfo** field can include multiple sensor information reports. Each report is a structure containing:

<b>Field</b>	<b>Description</b>
<b>SensorIndex</b>	Sensor index.
<b>DetectionCells</b>	Detection cells, returned as a logical matrix. Each column of the matrix denotes a detection cell. In each column, if the $i$ th element is 1, then the $i$ th detection belongs to the detection cell denoted by that column.
<b>DetectionLikelihoods</b>	The association likelihoods between components in the density function and detection cells, returned as an $N$ -by- $P$ matrix. $N$ is the number of components in the density function, and $P$ is the number of detection cells.
<b>IsBirthCells</b>	Indicates if the detection cells listed in <b>DetectionCells</b> give birth to new tracks, returned as a 1-by- $P$ logical vector, where $P$ is the number of detection cells.
<b>NumPartitions</b>	Number of partitions.
<b>DetectionProbability</b>	Probability of existing tracks being detected by the sensor, specified as a 1-by- $N$ row vector, where $N$ is the number of components in the density function.
<b>LabelsBeforeCorrection</b>	Labels of components in the density function before correction, return as a 1-by- $M_b$ row vector. $M_b$ is the number of components maintained in the tracker before correction. Each element of the vector is a <b>TrackID</b> . For example, [1 1 2 0 0]. Note that multiple components can share the same <b>TrackID</b> .
<b>LabelsAfterCorrection</b>	Labels of components in the density function after correction, returned as a 1-by- $M_a$ row vector. $M_a$ is the number of components maintained in the tracker after correction. Each element of the vector is a <b>TrackID</b> . For example, [1 1 1 2 2 0 0]. Note that multiple components can share the same <b>TrackID</b> .

<code>WeightsBeforeCorrection</code>	Weights of components in the density function before correction, returned as a 1-by- $M_b$ row vector. $M_b$ is the number of components maintained in the tracker before correction. Each element of the vector is the weight of the corresponding component given in <code>LabelsBeforeCorrection</code> . For example, [0.1 0.5 0.7 0.3 0.2].
<code>WeightsAfterCorrection</code>	Weights of components in the density function after correction, returned as a 1-by- $M_a$ row vector. $M_a$ is the number of components maintained in the tracker after correction. Each element of the vector is the weight of the corresponding component given in <code>LabelsAfterCorrection</code> . For example, [0.1 0.4 0.2 0.6 0.3 0.2 0.2].

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 trackerPHD

<code>predictTracksToTime</code>	Predict track state
<code>deleteTrack</code>	Delete existing track
<code>initializeTrack</code>	Initialize new track

### 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
<code>isLocked</code>	Determine if System object is in use
<code>clone</code>	Create duplicate System object
<code>reset</code>	Reset internal states of System object

## Examples

### Track Two Objects Using trackerPHD

Set up the sensor configuration, create a PHD tracker, and feed the tracker with detections.

```
% Create sensor configuration. Specify clutter density of the sensor and
% set the IsValidTime property to true.
configuration = trackingSensorConfiguration(1);
configuration.ClutterDensity = 1e-7;
configuration.IsValidTime = true;
```

```
% Create a PHD tracker.
tracker = trackerPHD('SensorConfigurations',configuration);

% Create detections near points [5;-5;0] and [-5;5;0] at t=0, and
% update the tracker with these detections.
detections = cell(20,1);
for i = 1:10
    detections{i} = objectDetection(0,[5;-5;0] + 0.2*randn(3,1));
end
for j = 11:20
    detections{j} = objectDetection(0,[-5;5;0] + 0.2*randn(3,1));
end

tracker(detections,0);
```

Update the tracker again after 0.1 seconds by assuming that targets move at a constant velocity of [1;2;0] unit per second.

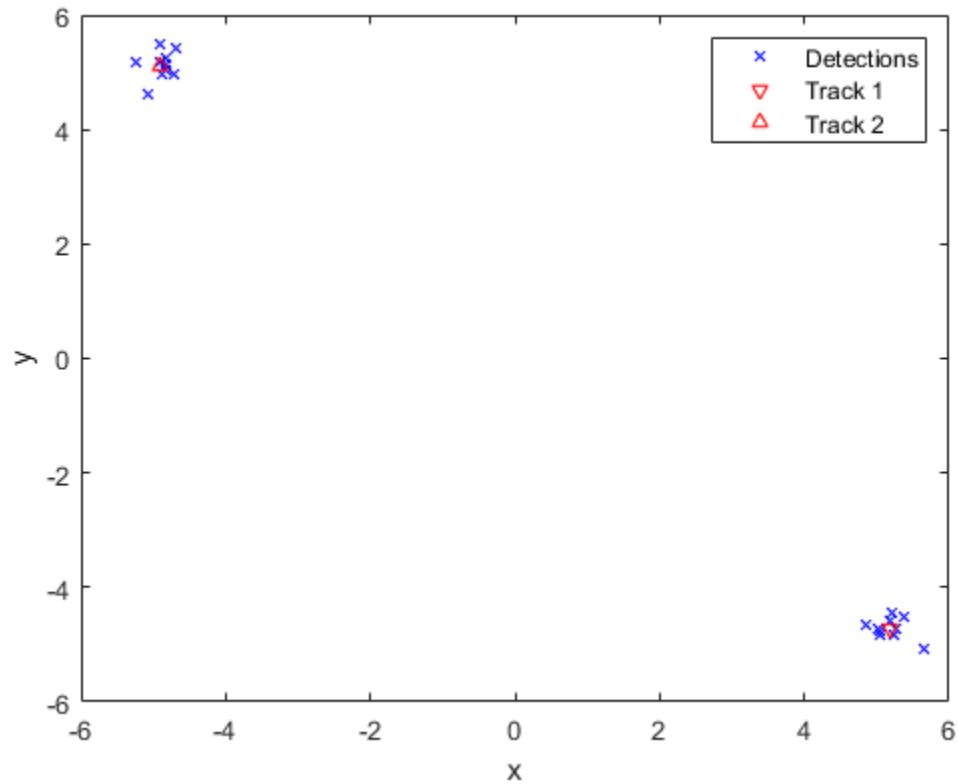
```
dT = 0.1;
for i = 1:20
    detections{i}.Time = detections{i}.Time + dT;
    detections{i}.Measurement = detections{i}.Measurement + [1;2;0]*dT;
end
[confTracks,tentTracks,allTracks] = tracker(detections,dT);
```

Visualize detections and confirmed tracks.

```
% Obtain measurements from detections.
d = [detections{:}];
measurements = [d.Measurement];

% Extract positions of confirmed tracking using getTrackPositions function.
% Note that we used the default sensor configuration
% FilterInitializationFcn, initcvrgiwpd, which uses a constant velocity
% model and defines the states as [x;vx;y;vy;z;vy].
positionSelector = [1 0 0 0 0 0;0 0 1 0 0 0;0 0 0 0 0 0];
positions = getTrackPositions(confTracks,positionSelector);

figure()
plot(measurements(1,:),measurements(2,:),'x','MarkerSize',5,'MarkerEdgeColor','b');
hold on;
plot(positions(1,1),positions(1,2),'v','MarkerSize',5,'MarkerEdgeColor','r' );
hold on;
plot(positions(2,1),positions(2,2),'^','MarkerSize',5,'MarkerEdgeColor','r' );
legend('Detections','Track 1','Track 2')
xlabel('x')
ylabel('y')
```



## More About

### Track Structure

Track information is returned as an array of structures having the following fields:

Field	Description
TrackID	Unique integer that identifies the track.
SourceIndex	Unique identifier of the tracker in a multiple tracker environment. The SourceIndex is exactly the same with the TrackerIndex.
UpdateTime	The time the track was updated.
Age	Number of times the track survived.
State	Value of state vector at the update time.
StateCovariance	Uncertainty covariance matrix.
Extent	Spatial extent estimate of the tracked object, returned as a $d$ -by- $d$ matrix, where $d$ is the dimension of the object. This field is only returned when the tracking filter is specified as a ggipwhd filter.

MeasurementRate	Expected number of detections from the tracked object. This field is only returned when the tracking filter is specified as a <code>ggiwphd</code> filter.
IsConfirmed	True if the track is assumed to be of a real target.
IsCoasted	trackerPHD does not support the <code>IsCoasted</code> field. The value is always 0.
ObjectClassID	trackerPHD does not support the <code>ObjectClassID</code> field. The value is always 0.
StateParamters	Parameters about the track state reference frame specified in the <code>StateParameters</code> property of the PHD tracker.
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.

## Algorithms

### Tracker Logic Flow

trackerPHD adopts an iterated-corrector approach to update the probability hypothesis density by processing detection information from multiple sensors sequentially. The workflow of trackerPHD follows these steps:

- 1 The tracker sorts sensors according to their detection reporting time and determines the order of correction accordingly.
- 2 The tracker considers two separate densities: current density and birth density. The current density is the density of targets propagated from the previous time step. The birth density is the density of targets expected to be born in the current time step.
- 3 For each sensor:
  - a The tracker predicts the current density to sensor time-stamp using the survival probability calculated from `DeathRate` and the elapsed time from the last prediction.
  - b The tracker adds new components to the birth density using the `FilterInitializationFcn` with no inputs. This corresponds to the predictive birth density.
  - c The tracker creates partitions of the detections from the current sensor using the function specified by the `PartitioningFcn` property. Each partition is a possible segmentation of detections into detection cells for each object. If the `SensorConfiguration` specifies the `MaxNumDetsPerObject` as 1, the tracker generates only 1 partition, in which each detection is a standalone cell.
  - d Each detection cell is evaluated against the current density, and a log-likelihood value is computed for each detection cell.
  - e Using the log-likelihood values, the tracker calculates the probability of each partition.
  - f The tracker corrects the current density using each detection cell.
  - g For detection cells with high negative log-likelihood (greater than `AssignmentThreshold`), the tracker adds new components to the birth density using `FilterInitializationFcn`. This corresponds to the adaptive birth density.

- 4 After correcting the current density with each sensor, the tracker adds the birth density to the current density. The tracker makes sure that number of possible targets in the birth density is equal to `BirthRate` $\times$  $dT$ , where  $dT$  is the time step.
- 5 The current density is then predicted to the current update time.

### Probability Hypothesis Density

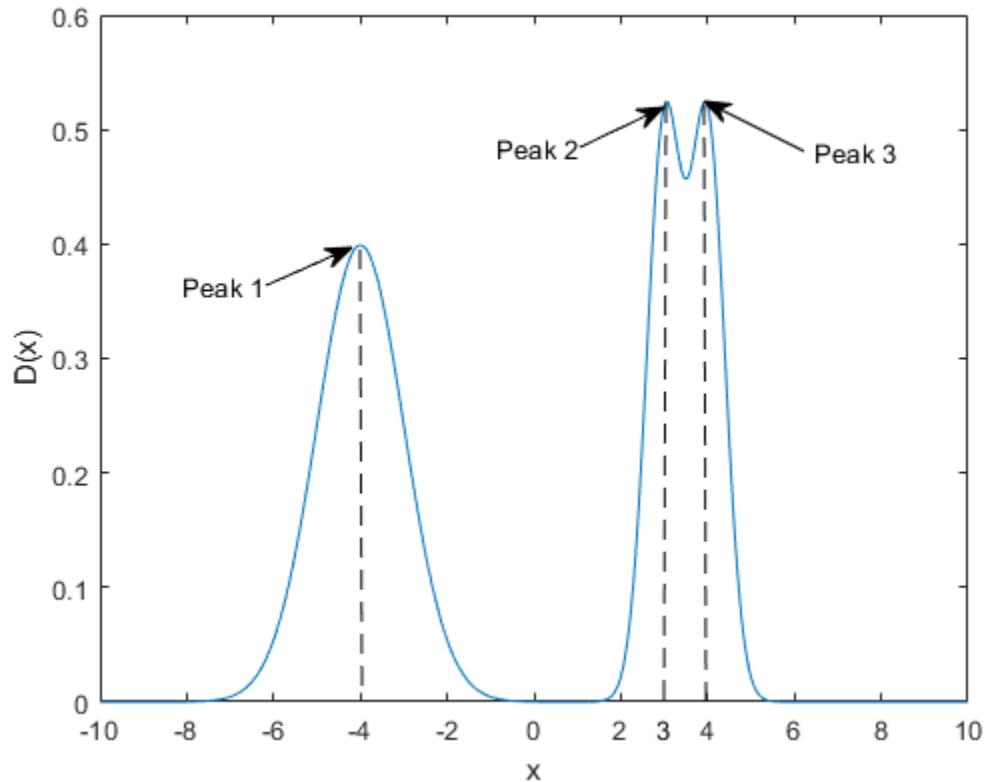
Probability hypothesis density (PHD) is a function defined over the state-space of the tracking system, and its value at a state is defined as the expected number of targets per unit state-space volume. The PHD is usually approximated by a mixture of components, and each component corresponds to an estimate of the state. The commonly used approximations of PHD are Gaussian mixture, SMC mixture, GGIW mixture, and GIW mixture. Currently, `trackerPHD` implements the GGIW mixture representation by `ggiwphd`, which can be used to track extended objects.

To understand PHD, take the Gaussian mixture as an example. The Gaussian mixture can be represented by

$$D(x) = \sum_{i=1}^M w_i N(x | m_i, P_i)$$

where  $M$  is the total number of components,  $N(x | m_i, P_i)$  is a normal distribution with mean  $m_i$  and covariance  $P_i$ , and  $w_i$  is the weight of the  $i$ th component. The weight  $w_i$  denotes the number (can be fractional) of targets represented by the  $i$ th component. Integration of  $D(x)$  over a state-space region results in the expected number of targets in that region. Integrating  $D(x)$  over the whole state space results in the total expected number of targets ( $\sum w_i$ ), since the integration of a normal distribution over the whole state space is 1. The  $x$  coordinates of the peaks (local maximums) of  $D(x)$  represent the most likely states of targets.

For example, the following figure illustrates a PHD function given by  $D(x) = N(x | -4, 2) + 0.5N(x | 3, 0.4) + 0.5N(x | 4, 0.4)$ . The weight summation of these components is 2, which means that 2 targets probably exist. From the peaks of  $D(x)$ , the possible positions of these targets are at  $x = -4$ ,  $x = 3$ , and  $x = 4$ . Notice that the last two components are very close to each other, which means that these two components can possibly be attributed to one object.



## References

- [1] Granstrom, K., C. Lundquist, and O. Orguner. " Extended target tracking using a Gaussian-mixture PHD filter." *IEEE Transactions on Aerospace and Electronic Systems*. Vol. 48, Number 4, 2012, pp. 3268-3286.
- [2] Granstrom, K., and O. Orguner." A PHD filter for tracking multiple extended targets using random matrices." *IEEE Transactions on Signal Processing*. Vol. 60, Number 11, 2012, pp. 5657-5671.
- [3] Granstrom, K., and A. Natale, P. Braca, G. Ludeno, and F. Serafino."Gamma Gaussian inverse Wishart probability hypothesis density for extended target tracking using X-band marine radar data." *IEEE Transactions on Geoscience and Remote Sensing*. Vol. 53, Number 12, 2015, pp. 6617-6631.

## 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 must have properties with the same sizes and types.

## See Also

### Functions

`getTrackPositions` | `getTrackVelocities` | `partitionDetections` | `predictTracksToTime`

### Objects

`objectDetection` | `trackingSensorConfiguration`

### System Objects

`staticDetectionFuser` | `trackerGNN` | `trackerJPDA` | `trackerTOMHT`

**Introduced in R2019a**

# initializeTrack

Initialize new track

## Syntax

```
trackID = initializeTrack(tracker,track)
trackID = initializeTrack(tracker,track,filter)
```

## Description

`trackID = initializeTrack(tracker,track)` initializes a new `track` in the PHD tracker. The tracker must be updated at least once before initializing a track. If the track is initialized successfully, the tracker or fuser assigns the output `trackID` to the track, set 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 or track fuser already maintains the maximum number of tracks specified by the `MaxNumTracks` property of the PHD tracker. In this case, the `trackID` is returned as `0`, which indicates a failure to initialize the track.

---

**Note** You can only use this syntax if the internal probability hypothesis density filter of the PHD tracker is `gmphd`. If the internal filter is `ggiwphd`, use the second syntax.

---

`trackID = initializeTrack(tracker,track,filter)` initializes a new `track` in the PHD tracker using a specified probability hypothesis density filter, `filter`.

---

## Note

- If the internal probability hypothesis density filter used in the tracker is a `ggiwphd` filter, you must use this syntax instead of the first syntax.
- 

## Examples

### Initialize a Track in `trackerPHD`

Create a PHD tracker after setting up the tracking sensor configuration. Update the tracker with ten detections. The tracker maintains one track.

```
configuration = trackingSensorConfiguration(1);
configuration.ClutterDensity = 1e-7;
configuration.IsValidTime = true;
tracker = trackerPHD('SensorConfigurations',configuration);

dt = 0.1;
for i = 1:10
    detections = objectDetection(i*dt,[5;-5;0] + 0.2*randn(3,1));
```

```
    tracker(detections,i*dt);
end
```

As seen from the `NumTracks` property, the tracker now maintains one track.

```
tracker
tracker =
  trackerPHD with properties:

    TrackerIndex: 0
    SensorConfigurations: {[1x1 trackingSensorConfiguration]}
    PartitioningFcn: 'partitionDetections'
    MaxNumSensors: 20
    MaxNumTracks: 1000

    AssignmentThreshold: 25
    BirthRate: 1.0000e-03
    DeathRate: 1.0000e-06

    ExtractionThreshold: 0.5000
    ConfirmationThreshold: 0.8000
    DeletionThreshold: 1.0000e-03
    MergingThreshold: 25
    LabelingThresholds: [1.1000 1 0.8000]

    StateParameters: [1x1 struct]
    HasSensorConfigurationsInput: false
      NumTracks: 1
      NumConfirmedTracks: 1
```

Create a new track using the `objectTrack` object.

```
newTrack = objectTrack();
```

Initialize a track in the PHD tracker using the newly created track.

```
trackID = initializeTrack(tracker,newTrack,ggiwphd)
```

```
trackID = uint32
2
```

As seen from the `NumTracks` property, the tracker now maintains two tracks.

```
tracker
tracker =
  trackerPHD with properties:

    TrackerIndex: 0
    SensorConfigurations: {[1x1 trackingSensorConfiguration]}
    PartitioningFcn: 'partitionDetections'
    MaxNumSensors: 20
    MaxNumTracks: 1000

    AssignmentThreshold: 25
    BirthRate: 1.0000e-03
    DeathRate: 1.0000e-06
```

```

ExtractionThreshold: 0.5000
ConfirmationThreshold: 0.8000
DeletionThreshold: 1.0000e-03
MergingThreshold: 25
LabelingThresholds: [1.1000 1 0.8000]

StateParameters: [1x1 struct]
HasSensorConfigurationsInput: false
NumTracks: 2
NumConfirmedTracks: 2

```

## Input Arguments

### **tracker — PHD tracker**

trackerPHD object

Probability hypothesis density tracker, specified as a `trackerPHD` 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 — Probability hypothesis density filter**

`gmpfd` | `ggiwphd`

Probability hypothesis density filter, specified as a `gmpfd` or `ggiwphd` 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

`trackerPHD`

**Introduced in R2020a**

# trackerJPDA

Joint probabilistic data association tracker

## Description

The `trackerJPDA` System object is a tracker capable of processing detections of multiple targets from multiple sensors. The tracker uses joint probabilistic data association to assign detections to each track. The tracker applies a soft assignment where multiple detections can contribute to each track. The tracker initializes, confirms, corrects, predicts (performs coasting), and deletes tracks. Inputs to the tracker are detection reports generated by `objectDetection`, `radarSensor`, `monostaticRadarSensor`, `irSensor`, or `sonarSensor` objects. The tracker estimates the state vector and state estimate error covariance matrix for each track. Each detection is assigned to at least one track. If the detection cannot be assigned to any existing track, the tracker creates a new track.

Any new track starts in a *tentative* state. If enough detections are assigned to a tentative track, its status changes to *confirmed* (see the `ConfirmationThreshold` property). If the detection already has a known classification (i.e., the `ObjectClassID` field of the returned track is nonzero), that corresponding track is confirmed immediately. When a track is confirmed, the tracker considers the track to represent a physical object. If detections are not assigned to the track within a specifiable number of updates, the track is deleted.

To track targets using this object:

- 1 Create the `trackerJPDA` 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 = trackerJPDA
tracker = trackerJPDA(Name,Value)
```

## Description

`tracker = trackerJPDA` creates a `trackerJPDA` System object with default property values.

`tracker = trackerJPDA(Name,Value)` sets properties for the tracker using one or more name-value pairs. For example,

`trackerJPDA('FilterInitializationFcn',@initcvukf,'MaxNumTracks',100)` creates a multi-object tracker that uses a constant-velocity, unscented Kalman filter and allows 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. You must specify this property as a positive integer to use the track outputs as inputs to a track fuser.

Example: 1

### **FilterInitializationFcn — Filter initialization function**

@initcvekf (default) | function handle | character vector

Filter initialization function, specified as a function handle or as a character vector containing the name of a valid filter initialization function. The tracker uses a filter initialization function when creating new tracks.

Sensor Fusion and Tracking Toolbox supplies many initialization functions that you can use to specify `FilterInitializationFcn` for a `trackerJPDA` object.

Initialization Function	Function Definition
<code>initcvkf</code>	Initialize constant-velocity linear Kalman filter.
<code>initcakf</code>	Initialize constant-acceleration linear Kalman filter.
<code>initcvabf</code>	Initialize constant-velocity alpha-beta filter
<code>initcaabf</code>	Initialize constant-acceleration alpha-beta filter
<code>initcvekf</code>	Initialize constant-velocity extended Kalman filter.
<code>initcaekf</code>	Initialize constant-acceleration extended Kalman filter.
<code>initrpekf</code>	Initialize constant-velocity range-parametrized extended Kalman filter.
<code>initapekf</code>	Initialize constant-velocity angle-parametrized extended Kalman filter.
<code>initctekf</code>	Initialize constant-turn-rate extended Kalman filter.
<code>initcackf</code>	Initialize constant-acceleration cubature filter.
<code>initctckf</code>	Initialize constant-turn-rate cubature filter.
<code>initcvckf</code>	Initialize constant-velocity cubature filter.

Initialization Function	Function Definition
<code>initcvukf</code>	Initialize constant-velocity unscented Kalman filter.
<code>initcaukf</code>	Initialize constant-acceleration unscented Kalman filter.
<code>initctukf</code>	Initialize constant-turn-rate unscented Kalman filter.
<code>initcvmscekf</code>	Initialize constant-velocity extended Kalman filter in modified spherical coordinates.
<code>initekfimm</code>	Initialize tracking IMM filter.

You can also write your own initialization function using the following syntax:

```
filter = filterInitializationFcn(detection)
```

The input to this function is a detection report like those created by `objectDetection`. The output of this function must be a filter object: `trackingKF`, `trackingEKF`, `trackingUKF`, `trackingCKF`, `trackingGSF`, `trackingIMM`, `trackingMSCEKF`, or `trackingABF`.

For guidance in writing this function, use the `type` command to examine the details of built-in MATLAB functions. For example:

```
type initcvekf
```

---

**Note** trackerJPDA does not accept all filter initialization functions in Sensor Fusion and Tracking Toolbox. The full list of filter initialization functions available in Sensor Fusion and Tracking Toolbox are given in the **Initialization** section of “Estimation Filters”.

---

Data Types: `function_handle` | `char`

**EventGenerationFcn — Feasible joint events generation function**  
`@jpdaEvents` (default) | `function handle` | `character vector`

Feasible joint events generation function, specified as a function handle or as a character vector containing the name of a feasible joint events generation function. A generation function generates feasible joint event matrices from admissible events (usually given by a validation matrix) of a tracking scenario. A validation matrix is a binary matrix listing all possible detections-to-track associations. For details, see `jpadEvents`.

You can also write your own generation function. The function must have the following syntax:

```
FJE = myfunction(ValidationMatrix)
```

The input and out of this function must exactly follow the formats used in `jpdaEvents`. For guidance in writing this function, use the `type` command to examine the details of `jpdaEvents`:

```
type jpdaEvents
```

Example: `@myfunction` or `'myfunction'`

Data Types: `function_handle` | `char`

**MaxNumTracks — Maximum number of tracks**

100 (default) | positive integer

Maximum number of tracks that the tracker can maintain, specified as a positive integer.

Data Types: single | 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.

MaxNumSensors must be greater than or equal to the largest value of SensorIndex found in all the detections used to update the tracker. SensorIndex is a property of an objectDetection object. The MaxNumSensors property determines how many sets of ObjectAttributes each track can have.

Data Types: single | double

**StateParameters — Parameters of track state reference frame**

struct([]) (default) | struct array

Parameters of the track state reference frame, specified as a structure array. Use this property to define the track state reference frame and how to transform the track from the tracker coordinate system, the source system, to the fuser coordinate system.

**Tunable:** Yes

Data Types: struct

**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 1-by-2 vector of  $[C_1, C_2]$ , where  $C_1 \leq C_2$ . If specified as a scalar, the specified value,  $val$ , is 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 the accurate normalized distance between them is less than  $C_1$ . See the `distance` function used with tracking filters (such as `trackingCKF` and `trackingEKF`) for explanation of the distance calculation.

Tips:

- Increase the value of  $C_2$  if there are track and detection combinations that should be calculated for assignment but are not. Decrease this value 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 this value if there are detections that are assigned to tracks they should not be assigned to (too far away).

**DetectionProbability — Probability of detection**

0.9 (default) | scalar in the range [0,1]

Probability of detection, specified as a scalar in the range [0,1]. This property is used in calculations of the marginal posterior probabilities of association and the probability of track existence when initializing and updating a track.

Example: 0.85

Data Types: single | double

#### **InitializationThreshold — Threshold to initialize a track**

0 (default) | scalar in the range [0,1]

The probability threshold to initialize a new track, specified as a scalar in the range [0,1]. If the probabilities of associating a detection with any of the existing tracks are all smaller than **InitializationThreshold**, the detection will be used to initialize a new track. This allows detections that are within the validation gate of a track but have an association probability lower than the initialization threshold to spawn a new track.

Example: 0.1

Data Types: single | double

#### **TrackLogic — Track confirmation and deletion logic type**

'History' (default) | 'Integrated'

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.
- 'Integrated' - Track confirmation and deletion is based on the probability of track existence, which is integrated in the assignment function.

#### **ConfirmationThreshold — Threshold for track confirmation**

scalar | 1-by-2 vector

Threshold for track confirmation, specified as a scalar or a 1-by-2 vector. The threshold depends on the type of track confirmation and deletion logic you set with the **TrackLogic** property:

- 'History' - Specify the confirmation threshold as 1-by-2 vector [M N]. A track is confirmed if it recorded at least M hits in the last N updates. The **trackerJPDA** registers a hit on a track's history logic according to the **HitMissThreshold**. The default value is [2 3].
- 'Integrated' - Specify the confirmation threshold as a scalar. A track is confirmed if its probability of existence is greater than or equal to the confirmation threshold. The default value is 0.95.

Data Types: single | double

#### **DeletionThreshold — Threshold for track deletion**

scalar | real-valued 1-by-2 vector

Threshold for track deletion, specified as a scalar or a real-valued 1-by-2 vector. The threshold depends on the type of track confirmation and deletion logic you set with the **TrackLogic** property:

- 'History' - Specify the confirmation threshold as [P R]. If, in P of the last R tracker updates, a confirmed track is not assigned to any detection that has a likelihood greater than the **HitMissThreshold** property, then that track is deleted. The default value is [5,5].
- 'Integrated' - Specify the deletion threshold as a scalar. A track is deleted if its probability of existence drops below the threshold. The default value is 0.1.

Example: 0.2 or [5,6]

Data Types: single | double

**HitMissThreshold — Threshold for registering hit or miss**

0.2 (default) | scalar in the range [0,1]

Threshold for registering a hit or miss, specified as a scalar in the range [0,1]. The track history logic will register a miss and the track will be coasted if the sum of the marginal probabilities of assignments is below the HitMissThreshold. Otherwise, the track history logic will register a hit.

Example: 0.3

**Dependencies**

To enable this argument, set the TrackLogic property to 'History'.

Data Types: single | double

**ClutterDensity — Spatial density of clutter measurements**

1e-6 (default) | positive scalar

Spatial density of clutter measurements, specified as a positive scalar. The clutter density describes the expected number of false positive detections per unit volume. It is used as the parameter of a Poisson clutter model. When TrackLogic is set to 'Integrated', ClutterDensity is also used in calculating the initial probability of track existence.

Example: 1e-5

Data Types: single | double

**NewTargetDensity — Spatial density of new targets**

1e-5 (default) | positive scalar

Spatial density of new targets, specified as a positive scalar. The new target density describes the expected number of new tracks per unit volume in the measurement space. It is used in calculating the probability of track existence during track initialization.

Example: 1e-3

**Dependencies**

To enable this argument, set the TrackLogic property to 'Integrated'.

Data Types: single | double

**DeathRate — Time rate of target deaths**

0.01 (default) | scalar in the range [0,1]

Time rate of target deaths, specified as a scalar in the range [0,1]. DeathRate describes the probability with which true targets disappear. It is related to the propagation of the probability of track existence (*PTE*) :

$$PTE(t + \delta t) = (1 - DeathRate)^{\delta t} PTE(t)$$

where  $\delta t$  is the time interval since the previous update time  $t$ .

**Dependencies**

To enable this argument, set the TrackLogic property to 'Integrated'.

Data Types: single | double

**InitialExistenceProbability — Initial probability of track existence**  
0.9 (default) | scalar in the range [0,1]

This property is read-only.

Initial probability of track existence, specified as a scalar in the range [0,1] and calculated as  $\text{InitialExistenceProbability} = \text{NewTargetDensity} * \text{DetectionProbability} / (\text{ClutterDensity} + \text{NewTargetDensity} * \text{DetectionProbability})$ .

#### Dependencies

To enable this property, set the `TrackLogic` property to '`Integrated`'. When the `TrackLogic` property is set to '`History`', this property is not available.

Data Types: single | double

**HasCostMatrixInput — Enable cost matrix input**  
false (default) | true

Enable a cost matrix, specified as `false` or `true`. If `true`, you can provide an assignment cost matrix as an input argument when calling the object.

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 informs the tracker of all tracks that the sensors are expected to detect and, optionally, the probability of detection for each track.

Data Types: logical

**NumTracks — Number of tracks maintained by tracker**  
nonnegative integer

This property is read-only.

Number of tracks maintained by the tracker, returned as a nonnegative integer.

Data Types: single | double

**NumConfirmedTracks — Number of confirmed tracks**  
nonnegative integer

This property is read-only.

Number of confirmed tracks, returned as a nonnegative integer. If the `IsConfirmed` field of an output track structure is `true`, the track is confirmed.

Data Types: single | double

**TimeTolerance — Absolute time tolerance between detections**  
1e-5 (default) | positive scalar

Absolute time tolerance between detections for the same sensor, specified as a positive scalar. Ideally, trackerJPDA expects detections from a sensor to have identical time stamps. However, if the time stamps differences between detections of a sensor are within the margin specified by TimeTolerance, these detections will be used to update the track estimate based on the average time of these detections.

Data Types: double

## Usage

To process detections and update tracks, call the tracker with arguments, as if it were a function (described here).

## Syntax

```
confirmedTracks = tracker(detections,time)
confirmedTracks = tracker(detections,time,costMatrix)
confirmedTracks = tracker(___,detectableTrackIDs)
[confirmedTracks,tentativeTracks,allTracks] = tracker(___)
[confirmedTracks,tentativeTracks,allTracks,analysisInformation] = tracker(___
)
```

### Description

`confirmedTracks = tracker(detections,time)` returns a list of confirmed tracks that are updated from a list of detections at the update time. Confirmed tracks are corrected and predicted to the update time, `time`.

`confirmedTracks = tracker(detections,time,costMatrix)` also specifies a cost matrix.

To enable this syntax, set the `HasCostMatrixInput` property to `true`.

`confirmedTracks = 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`.

`[confirmedTracks,tentativeTracks,allTracks] = tracker(___)` also returns a list of tentative tracks and a list of all tracks. You can use any of the input arguments in the previous syntaxes.

`[confirmedTracks,tentativeTracks,allTracks,analysisInformation] = tracker(___)` also returns analysis information that can be used for track analysis. You can use any of the input arguments in the previous syntaxes.

### 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 update time, `time`, and

greater than the previous time value used to update the tracker. Also, the Time differences between different `objectDetection` objects in the cell array do not need to be equal.

#### **time — Time of update**

scalar

Time of update, specified as a 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 tracker.

Data Types: `single` | `double`

#### **costMatrix — Cost matrix**

real-valued  $M$ -by- $N$  matrix

Cost matrix, specified as a real-valued  $M$ -by- $N$  matrix, where  $M$  is the number of existing tracks in the previous update, and  $N$  is the number of current detections. The cost matrix rows must be in the same order as the list of tracks, and the columns must be in the same order as the list of detections. Obtain the correct order of the list of tracks from the third output argument, `allTracks`, when the tracker is updated.

At the first update of the tracker or when the tracker has no previous tracks, specify the cost matrix to be empty with a size of `[0, numDetections]`. Note that the cost must be given so that lower costs indicate a higher likelihood of assigning a detection to a track. To prevent certain detections from being assigned to certain tracks, you can set the appropriate cost matrix entry to `Inf`.

#### **Dependencies**

To enable this argument, set the `HasCostMatrixInput` property to `true`.

Data Types: `double` | `single`

#### **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 allows you to add the detection probability for each track.

Tracks whose identifiers are not included in `detectableTrackIDs` are considered undetectable. In this case, the track deletion logic does not count the lack of detection for that track 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`

**analysisInformation — Additional information for analyzing track updates**

structure

Additional information for analyzing track updates, returned as a structure. The fields of this structure are:

Field	Description
<code>TrackIDsAtStepBeginning</code>	Track IDs when step began.
<code>CostMatrix</code>	Cost matrix for assignment.
<code>UnassignedTracks</code>	IDs of unassigned tracks.
<code>UnassignedDetections</code>	Indices of unassigned detections in the <code>detections</code> input.
<code>Clusters</code>	Cell array of cluster reports.
<code>InitiatedTrackIDs</code>	IDs of tracks initiated during the step.
<code>DeletedTrackIDs</code>	IDs of tracks deleted during the step.
<code>TrackIDsAtStepEnd</code>	Track IDs when the step ended.

The `Clusters` field can include multiple cluster reports. Each cluster report is a structure containing:

Field	Description
DetectionIndices	Indices of clustered detections.
TrackIDs	Track IDs of clustered tracks.
ValidationMatrix	Validation matrix of the cluster. See <code>jpadEvents</code> for more details.
SensorIndex	Index of the originating sensor of the clustered detections.
TimeStamp	Mean time stamp of clustered detections.
MarginalProbabilities	Matrix of marginal posterior joint association probabilities.

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 trackerJPDA

<code>predictTracksToTime</code>	Predict track state
<code>getTrackFilterProperties</code>	Obtain track filter properties
<code>setTrackFilterProperties</code>	Set track filter properties
<code>initializeTrack</code>	Initialize new track
<code>deleteTrack</code>	Delete existing track

### 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
<code>isLocked</code>	Determine if System object is in use
<code>clone</code>	Create duplicate System object
<code>reset</code>	Reset internal states of System object

## Examples

### Track Two Objects Using trackerJPDA

Construct a `trackerJPDA` object with a default constant velocity Extended Kalman Filter and 'History' track logic. Set `AssignmentThreshold` to 100 to allow tracks to be jointly associated.

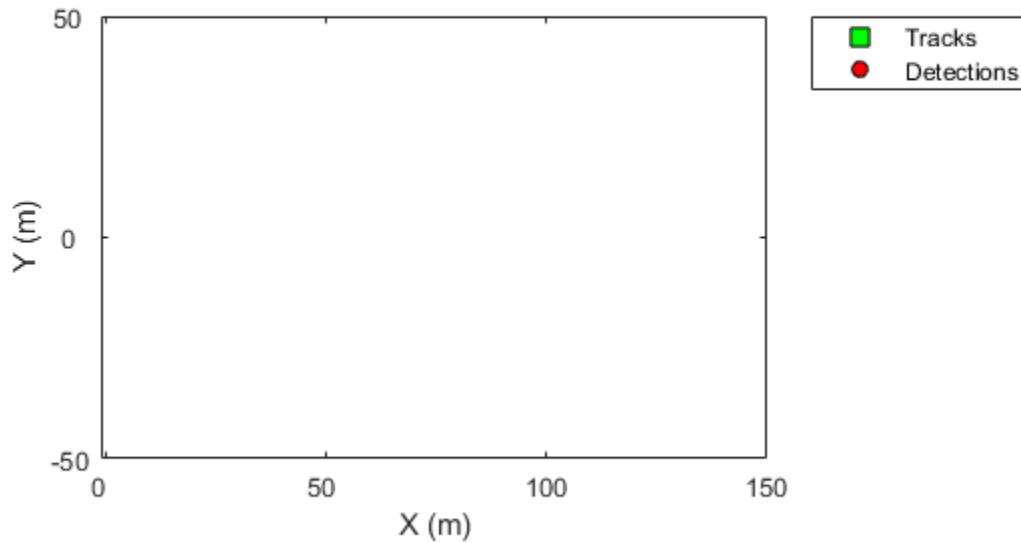
```
tracker = trackerJPDA('TrackLogic','History', 'AssignmentThreshold',100, ...
    'ConfirmationThreshold', [4 5], ...
    'DeletionThreshold', [10 10]);
```

Specify the true initial positions and velocities of the two objects.

```
pos_true = [0 0 ; 40 -40 ; 0 0];
V_true = 5*[cosd(-30) cosd(30) ; sind(-30) sind(30) ;0 0];
```

Create a theater plot to visualize tracks and detections.

```
tp = theaterPlot('XLimits',[ -1 150], 'YLimits', [-50 50]);
trackP = trackPlotter(tp,'DisplayName','Tracks','MarkerFaceColor','g','HistoryDepth',0);
detectionP = detectionPlotter(tp,'DisplayName','Detections','MarkerFaceColor','r');
```



To obtain the position and velocity, create position and velocity selectors.

```
positionSelector = [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 0 0]; % [x, y, 0]
velocitySelector = [0 1 0 0 0 0; 0 0 0 1 0 0; 0 0 0 0 0 0]; % [vx, vy, 0]
```

Update the tracker with detections, display cost and marginal probability of association information, and visualize tracks with detections.

```
dt = 0.2;
for time = 0:dt:30
    % Update the true positions of objects.
    pos_true = pos_true + V_true*dt;

    % Create detections of the two objects with noise.
    detection(1) = objectDetection(time,pos_true(:,1)+1*randn(3,1));
    detection(2) = objectDetection(time,pos_true(:,2)+1*randn(3,1));

    % Step the tracker through time with the detections.
    [confirmed,tentative,alltracks,info] = tracker(detection,time);
```

```
% Extract position, velocity and label info.
[pos,cov] = getTrackPositions(confirmed,positionSelector);
vel = getTrackVelocities(confirmed,velocitySelector);
meas = cat(2,detection.Measurement);
measCov = cat(3,detection.MeasurementNoise);

% Update the plot if there are any tracks.
if numel(confirmed)>0
    labels = arrayfun(@(x)num2str([x.TrackID]),confirmed,'UniformOutput',false);
    trackP.plotTrack(pos,vel,cov,labels);
end
detectionP.plotDetection(meas',measCov);
drawnow;

% Display the cost and marginal probability of distribution every eight
% seconds.
if time>0 && mod(time,8) == 0
    disp(['At time t = ' num2str(time) ' seconds,']);
    disp('The cost of assignment was: ')
    disp(info.CostMatrix);
    disp(['Number of clusters: ' num2str(numel(info.Clusters))]);
    if numel(info.Clusters) == 1

        disp('The two tracks were in the same cluster.')
        disp('Marginal probabilities of association:')
        disp(info.Clusters{1}.MarginalProbabilities)
    end
    disp('-----')
end
end
```

At time t = 8 seconds,

The cost of assignment was:

```
1.0e+03 *
0.0020    1.1523
1.2277    0.0053
```

Number of clusters: 2

-----

At time t = 16 seconds,

The cost of assignment was:

```
1.3968    4.5123
2.0747    1.9558
```

Number of clusters: 1

The two tracks were in the same cluster.

Marginal probabilities of association:

```
0.8344    0.1656
0.1656    0.8344
0.0000    0.0000
```

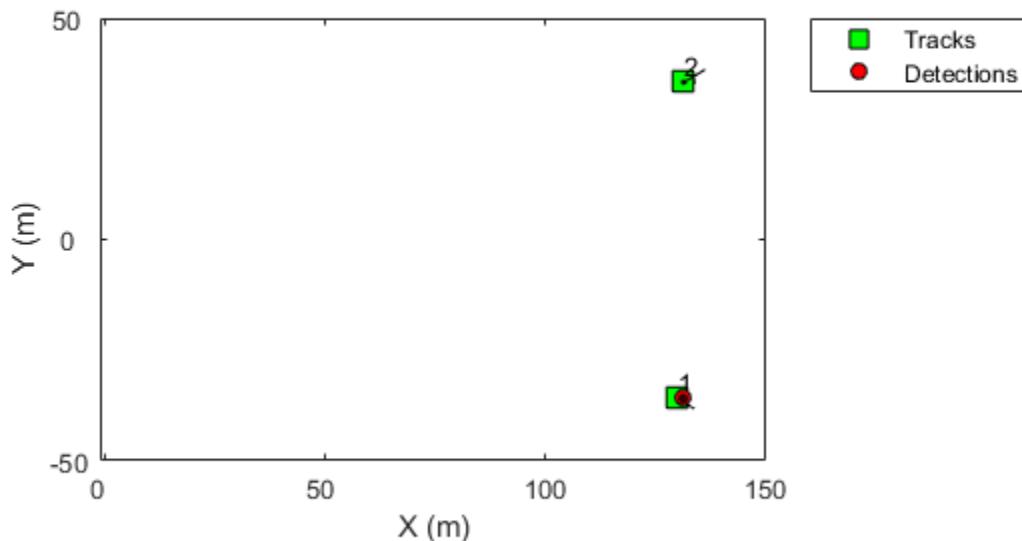
At time  $t = 24$  seconds,

The cost of assignment was:

```
1.0e+03 *  
0.0018    1.2962  
1.2664    0.0013
```

Number of clusters: 2

---



## Algorithms

### Tracker Logic Flow

When a JPDA tracker processes detections, track creation and management follow these steps.

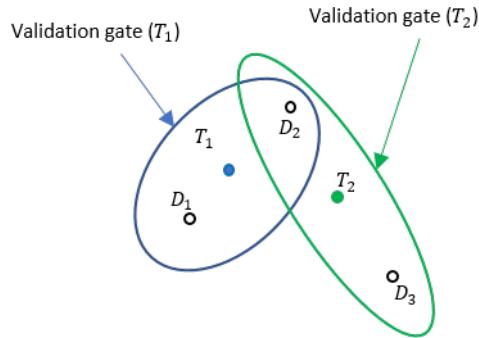
- 1 The tracker divides detections into multiple groups by originating sensor.
- 2 For each sensor:
  - a The tracker calculates the distances from detections to existing tracks and forms a `costMatrix`.

- b** The tracker creates a validation matrix based on the assignment threshold (or gate threshold) of the existing tracks. A validation matrix is a binary matrix listing all possible detections-to-track associations. For details, see “Feasible Joint Events” on page 3-469.
  - c** Tracks and detections are then separated into clusters. A cluster can contain one track or multiple tracks if these tracks share common detections within their validation gates. A validation gate is a spatial boundary, in which the predicted detection of the track has a high likelihood to fall. For details, see “Feasible Joint Events” on page 3-469.
- 3** Update all clusters following the order of the mean detection time stamp within the cluster. For each cluster, the tracker:
- a** Generates all feasible joint events. For details, see `jpdaEvents`.
  - b** Calculates the posterior probability of each joint event.
  - c** Calculates the marginal probability of each individual detection-track pair in the cluster.
  - d** Reports weak detections. Weak detections are the detections that are within the validation gate of at least one track, but have probability association to all tracks less than the `InitializationThreshold`.
  - e** Updates tracks in the cluster using `correctjpda`.
- 4** Unassigned detections (these are not in any cluster) and weak detections spawn new tracks.
- 5** The tracker checks all tracks for deletion. Tracks are deleted based on the number of scans without association using ‘History’ logic or based on their probability of existence using ‘Integrated’ track logic.
- 6** All tracks are predicted to the latest time value (either the time input if provided, or the latest mean cluster time stamp).

### Feasible Joint Events

In the typical workflow for a tracking system, the tracker needs to determine if a detection can be associated with any of the existing tracks. If the tracker only maintains one track, the assignment can be done by evaluating the validation gate around the predicted measurement and deciding if the measurement falls within the *validation gate*. In the measurement space, the validation gate is a spatial boundary, such as a 2-D ellipse or a 3-D ellipsoid, centered at the predicted measurement. The validation gate is defined using the probability information (state estimation and covariance, for example) of the existing track, such that the correct or ideal detections have high likelihood (97% probability, for example) of falling within this validation gate.

However, if a tracker maintains multiple tracks, the data association process becomes more complicated, because one detection can fall within the validation gates of multiple tracks. For example, in the following figure, tracks  $T_1$  and  $T_2$  are actively maintained in the tracker, and each of them has its own validation gate. Since the detection  $D_2$  is in the intersection of the validation gates of both  $T_1$  and  $T_2$ , the two tracks ( $T_1$  and  $T_2$ ) are connected and form a *cluster*. A cluster is a set of connected tracks and their associated detections.



To represent the association relationship in a cluster, the validation matrix is commonly used. Each row of the validation matrix corresponds to a detection while each column corresponds to a track. To account for the eventuality of each detection being clutter, a first column is added and usually referred to as "Track 0" or \$T\_0\$. If detection \$D\_i\$ is inside the validation gate of track \$D\_j\$, then the \$(j, i+1)\$ entry of the validation matrix is 1. Otherwise, it is zero. For the cluster shown in the figure, the validation matrix \$\Omega\$ is

$$\Omega = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Note that all the elements in the first column of \$\Omega\$ are 1, because any detection can be clutter or false alarm. One important step in the logic of joint probabilistic data association (JPDA) is to obtain all the feasible independent joint events in a cluster. Two assumptions for the feasible joint events are:

- A detection cannot be emitted by more than one track.
- A track cannot be detected more than once by the sensor during a single scan.

Based on these two assumptions, feasible joint events (FJEs) can be formulated. Each FJE is mapped to an FJE matrix \$\Omega\_p\$ from the initial validation matrix \$\Omega\$. For example, with the validation matrix \$\Omega\$, eight FJE matrices can be obtained:

$$\Omega_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \Omega_2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \Omega_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \Omega_4 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

$$\Omega_5 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \quad \Omega_6 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \Omega_7 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \Omega_8 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

As a direct consequence of the two assumptions, the \$\Omega\_p\$ matrices have exactly one "1" value per row. Also, except for the first column which maps to clutter, there can be at most one "1" per column. When the number of connected tracks grows in a cluster, the number of FJE increases rapidly. The `jpdaEvents` function uses an efficient depth-first search algorithm to generate all the feasible joint event matrices.

## 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.
- [2] Musicki, D., and R. Evans. "Joint Integrated Probabilistic Data Association: JIPDA." *IEEE transactions on Aerospace and Electronic Systems*. Vol. 40, Number 3, 2004, pp 1093-1099.

## 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 a multi-object tracker must have properties with the same sizes and types.
- If you use the `ObjectAttributes` field within an `objectDetection` object, you must specify this field as 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.
- If `ObjectAttributes` are contained in the detection, the `SensorIndex` value of the detection cannot be greater than 10.
- The first update to the multi-object tracker must contain at least one detection.

## See Also

### Functions

`correctjpda` | `getTrackPositions` | `getTrackVelocities` | `jpdaEvents` |  
`predictTracksToTime`

### Objects

`objectDetection` | `objectTrack` | `trackHistoryLogic` | `trackingABF` | `trackingCKF` |  
`trackingEKF` | `trackingIMM` | `trackingKF` | `trackingUKF`

### System Objects

`staticDetectionFuser` | `trackerGNN` | `trackerTOMHT`

### Introduced in R2019a

# poseTrajectory

Pose trajectory generator

## Description

`poseTrajectory` System object creates a trajectory starting from an initial pose. Execute the object to obtain the pose at each time step.

The object supports single and double data types for property values. If the value for a name-value pair is a single, then all of the properties and outputs from the object are converted to the single data type. Otherwise, the double data type is used. Data types cannot be changed after the object has been created.

To obtain trajectory points:

- 1 Create the `poseTrajectory` 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 = poseTrajectory()
trajectory = poseTrajectory(Name,Value)
```

### Description

`trajectory = poseTrajectory()` creates a pose trajectory object with default property values.

`trajectory = poseTrajectory(Name,Value)` sets properties using one or more name-value pairs. For example, `traj = poseTrajectory('SampleRate',2,'Position',[100 500 2000])` creates a pose trajectory that reports trajectory values every  $\frac{1}{2}$  second and has an initial position of (100,500,2000) meters in the scenario coordinate system. 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](#).

**SampleRate — Sampling frequency of trajectory**

100 (default) | positive scalar

Sample rate of trajectory, specified as a positive scalar. This property is tunable. Units are in hertz.

Data Types: single

**Position — Initial position of platform in scenario frame**

[0 0 0] (default) | 1-by-3 real-valued vector

Initial position of platform in the scenario frame, specified as a 1-by-3 real-valued vector. This property is tunable. Units are in meters.

Data Types: single | double

**Velocity — Initial velocity of platform in scenario frame**

[0 0 0] (default) | 1-by-3 real-valued vector

Initial velocity of platform in the scenario frame, specified as a 1-by-3 real-valued vector. This property is tunable. Units are in meters per second.

Example: [100 500 2000]

Data Types: single | double

**Acceleration — Acceleration of platform in body frame**

[0 0 0] (default) | 1-by-3 real-valued vector

Acceleration of platform in the body frame, specified as a 1-by-3 real-valued vector. This property is tunable. Units are in meters per second squared.

Data Types: single

**AngularVelocity — Angular velocity of platform in body frame**

[0 0 0] (default) | 1-by-3 real-valued vector

Angular velocity of platform in the body frame, specified as a 1-by-3 real-valued vector. This property is tunable. Units are in radians per second.

## Usage

### Syntax

```
[pos,orient,vel,acc,angvel] = trajectory()
```

### Description

[pos,orient,vel,acc,angvel] = trajectory() returns the pose of a platform at its current trajectory point.

- pos - current position
- orient - orientation
- vel - velocity
- acc - acceleration

- **angvel** – angular velocity

### Output Arguments

**pos — position of platform in scenario frame**

1-by-3 real-valued vector

Position of platform in scenario coordinates, returned as a 1-by-3 real-valued vector. Units are in meters.

**orient — Orientation of platform in scenario coordinates**

quaternion | 3-by-3 real-valued orthogonal matrix

Orientation of body frame, returned as a quaternion or 3-by-3 real-valued orthogonal matrix. The orientation rotates the scenario frame into the body frame. Units are dimensionless.

Data Types: double

**vel — Velocity of platform in scenario frame**

1-by-3 real-valued vector

Velocity of platform in scenario frame, returned as a 1-by-3 real-valued vector. Units are in meters per second.

**acc — Acceleration of platform in body frame**

1-by-3 real-valued vector

Acceleration of platform in body frame, returned as a 1-by-3 real-valued vector. Units are in meters per second squared.

Data Types: double

**angvel — Angular velocity in body frame**

[0 0 0] (default) | 1-by-3 real-valued vector

Angular velocity of body frame, returned as a 1-by-3 real-valued vector. Units are in radians per second.

## 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)
```

## Examples

### Create Circular Trajectory

Create a trajectory that follows a circle. Set the number of trajectory points to 500. To follow a circular trajectory, the platform must accelerate.

```
N = 500;  
fs = 1;
```

Set the initial conditions for the object motion. Place the object on the x-axis 100 meters from the origin in scenario coordinates. Set the velocity of the body to 2.5 m/s along the y-axis.

```
r = 100;
speed = 2.5;
initPos = [r,0,0];
velBody = [0,speed,0];
```

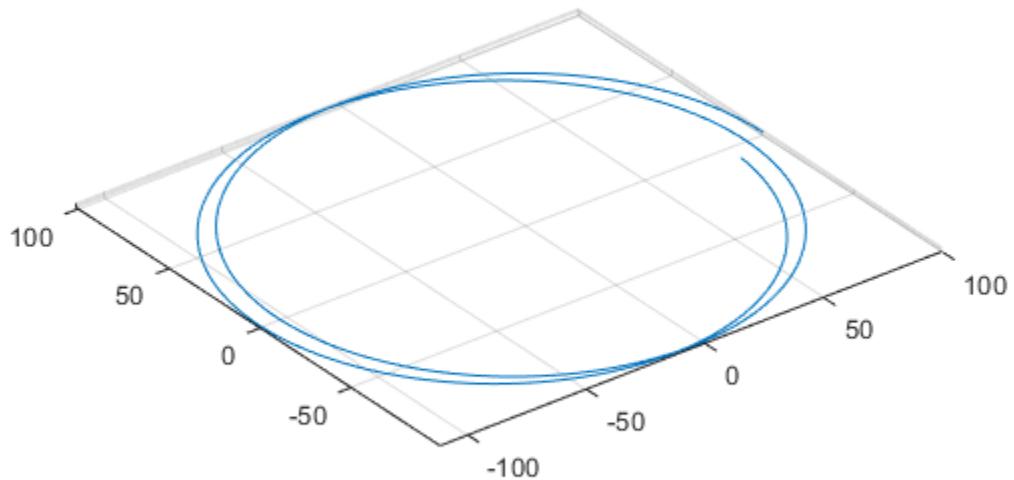
Orient the body along the direction of motion. Apply an acceleration orthogonal to the body in the xy-plane. Acceleration is always in the body frame. Rotate the body as it moves by an angular rotation rate equal to the rotation rate around the origin.

```
accmag = speed^2/r;
initialYaw = deg2rad(90);
initPos = [r,0,0];
velBody = [0,speed,0];
accBody = [0,accmag,0];
initAtt = quaternion([initialYaw, 0, 0], 'euler', 'ZYX', 'frame');
traj = kinematicTrajectory('SampleRate', fs, 'Position', initPos, ...
    'Velocity', velBody, 'Orientation', initAtt);

pos = zeros(N, 3);
for i = 1:N
    pos(i,:) = traj(accBody, [0 0 speed/r]);
end
```

Plot the trajectory.

```
plot3(pos(:,1), pos(:,2), pos(:,3))
grid
axis equal
```



**See Also**

[trackerGNN](#)

**Introduced in R2018b**

# imuSensor

IMU simulation model

## Description

The `imuSensor` System object models receiving data from an inertial measurement unit (IMU).

To model an IMU:

- 1 Create the `imuSensor` 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

```
IMU = imuSensor
IMU = imuSensor('accel-gyro')
IMU = imuSensor('accel-mag')
IMU = imuSensor('accel-gyro-mag')
IMU = imuSensor(___, 'ReferenceFrame', RF)
IMU = imuSensor(___, Name, Value)
```

### Description

`IMU = imuSensor` returns a System object, `IMU`, that computes an inertial measurement unit reading based on an inertial input signal. `IMU` has an ideal accelerometer and gyroscope.

`IMU = imuSensor('accel-gyro')` returns an `imuSensor` System object with an ideal accelerometer and gyroscope. `imuSensor` and `imuSensor('accel-gyro')` are equivalent creation syntaxes.

`IMU = imuSensor('accel-mag')` returns an `imuSensor` System object with an ideal accelerometer and magnetometer.

`IMU = imuSensor('accel-gyro-mag')` returns an `imuSensor` System object with an ideal accelerometer, gyroscope, and magnetometer.

`IMU = imuSensor(___, 'ReferenceFrame', RF)` returns an `imuSensor` System object that computes an inertial measurement unit reading relative to the reference frame `RF`. Specify `RF` as '`NED`' (North-East-Down) or '`ENU`' (East-North-Up). The default value is '`NED`'.

`IMU = imuSensor(___, Name, Value)` sets each property `Name` to the specified `Value`. Unspecified properties have default values. This syntax can be used in combination with any of the previous input arguments.

## 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](#).

### **IMUType — Type of inertial measurement unit**

`'accel-gyro'` (default) | `'accel-mag'` | `'accel-gyro-mag'`

Type of inertial measurement unit, specified as a `'accel-gyro'`, `'accel-mag'`, or `'accel-gyro-mag'`.

The type of inertial measurement unit specifies which sensor readings to model:

- `'accel-gyro'` -- Accelerometer and gyroscope
- `'accel-mag'` -- Accelerometer and magnetometer
- `'accel-gyro-mag'` -- Accelerometer, gyroscope, and magnetometer

You can specify `IMUType` as a value-only argument during creation or as a `Name`, `Value` pair.

Data Types: `char` | `string`

### **SampleRate — Sample rate of sensor (Hz)**

`100` (default) | positive scalar

Sample rate of the sensor model in Hz, specified as a positive scalar.

Data Types: `single` | `double`

### **Temperature — Temperature of IMU (°C)**

`25` (default) | real scalar

Operating temperature of the IMU in degrees Celsius, specified as a real scalar.

When the object calculates temperature scale factors and environmental drift noises, 25 °C is used as the nominal temperature.

**Tunable:** Yes

Data Types: `single` | `double`

### **MagneticField — Magnetic field vector in local navigation coordinate system (µT)**

`[27.5550 -2.4169 -16.0849]` (default) | real scalar

Magnetic field vector in microtesla, specified as a three-element row vector in the local navigation coordinate system.

The default magnetic field corresponds to the magnetic field at latitude zero, longitude zero, and altitude zero.

**Tunable:** Yes

Data Types: `single` | `double`

**Accelerometer — Accelerometer sensor parameters**`accelparams` object (default)Accelerometer sensor parameters, specified by an `accelparams` object.**Tunable:** Yes**Gyroscope — Gyroscope sensor parameters**`gyroparams` object (default)Gyroscope sensor parameters, specified by a `gyroparams` object.**Tunable:** Yes**Magnetometer — Magnetometer sensor parameters**`magparams` object (default)Magnetometer sensor parameters, specified by a `magparams` object.**Tunable:** Yes**RandomStream — Random number source**

'Global stream' (default) | 'mt19937ar with seed'

Random number source, specified as a character vector or string:

- 'Global stream' -- Random numbers are generated using the current global random number stream.
- 'mt19937ar with seed' -- Random numbers are generated using the mt19937ar algorithm with the seed specified by the `Seed` property.

Data Types: `char` | `string`**Seed — Initial seed**

67 (default) | nonnegative integer scalar

Initial seed of an mt19937ar random number generator algorithm, specified as a real, nonnegative integer scalar.

**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

```
[accelReadings,gyroReadings] = IMU(acc,angVel)
[accelReadings,gyroReadings] = IMU(acc,angVel,orientation)

[accelReadings,magReadings] = IMU(acc,angVel)
[accelReadings,magReadings] = IMU(acc,angVel,orientation)
```

```
[accelReadings,gyroReadings,magReadings] = IMU(acc,angVel)
[accelReadings,gyroReadings,magReadings] = IMU(acc,angVel,orientation)
```

### Description

[accelReadings,gyroReadings] = IMU(acc,angVel) generates accelerometer and gyroscope readings from the acceleration and angular velocity inputs.

This syntax is only valid if IMUType is set to 'accel-gyro' or 'accel-gyro-mag'.

[accelReadings,gyroReadings] = IMU(acc,angVel,orientation) generates accelerometer and gyroscope readings from the acceleration, angular velocity, and orientation inputs.

This syntax is only valid if IMUType is set to 'accel-gyro' or 'accel-gyro-mag'.

[accelReadings,magReadings] = IMU(acc,angVel) generates accelerometer and magnetometer readings from the acceleration and angular velocity inputs.

This syntax is only valid if IMUType is set to 'accel-mag'.

[accelReadings,magReadings] = IMU(acc,angVel,orientation) generates accelerometer and magnetometer readings from the acceleration, angular velocity, and orientation inputs.

This syntax is only valid if IMUType is set to 'accel-mag'.

[accelReadings,gyroReadings,magReadings] = IMU(acc,angVel) generates accelerometer, gyroscope, and magnetometer readings from the acceleration and angular velocity inputs.

This syntax is only valid if IMUType is set to 'accel-gyro-mag'.

[accelReadings,gyroReadings,magReadings] = IMU(acc,angVel,orientation) generates accelerometer, gyroscope, and magnetometer readings from the acceleration, angular velocity, and orientation inputs.

This syntax is only valid if IMUType is set to 'accel-gyro-mag'.

### Input Arguments

#### **acc — Acceleration of IMU in local navigation coordinate system (m/s<sup>2</sup>)**

N-by-3 matrix

Acceleration of the IMU in the local navigation coordinate system, specified as a real, finite N-by-3 array in meters per second squared. N is the number of samples in the current frame.

Data Types: single | double

#### **angVel — Angular velocity of IMU in local navigation coordinate system (rad/s)**

N-by-3 matrix

Angular velocity of the IMU in the local navigation coordinate system, specified as a real, finite N-by-3 array in radians per second. N is the number of samples in the current frame.

Data Types: single | double

#### **orientation — Orientation of IMU in local navigation coordinate system**

N-element quaternion column vector | 3-by-3-by-N-element rotation matrix

Orientation of the IMU with respect to the local navigation coordinate system, specified as a quaternion  $N$ -element column vector or a 3-by-3-by- $N$  rotation matrix. Each quaternion or rotation matrix represents a frame rotation from the local navigation coordinate system to the current IMU sensor body coordinate system.  $N$  is the number of samples in the current frame.

Data Types: `single` | `double` | `quaternion`

### **Output Arguments**

#### **accelReadings — Accelerometer measurement of IMU in sensor body coordinate system (m/s<sup>2</sup>)**

$N$ -by-3 matrix

Accelerometer measurement of the IMU in the sensor body coordinate system, specified as a real, finite  $N$ -by-3 array in meters per second squared.  $N$  is the number of samples in the current frame.

Data Types: `single` | `double`

#### **gyroReadings — Gyroscope measurement of IMU in sensor body coordinate system (rad/s)**

$N$ -by-3 matrix

Gyroscope measurement of the IMU in the sensor body coordinate system, specified as a real, finite  $N$ -by-3 array in radians per second.  $N$  is the number of samples in the current frame.

Data Types: `single` | `double`

#### **magReadings — Magnetometer measurement of IMU in sensor body coordinate system (μT)**

$N$ -by-3 matrix (default)

Magnetometer measurement of the IMU in the sensor body coordinate system, specified as a real, finite  $N$ -by-3 array in microtesla.  $N$  is the number of samples in the current frame.

Data Types: `single` | `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 imuSensor**

`loadparams` Load sensor parameters from JSON file

### **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 Default `imuSensor` System object

The `imuSensor` System object™ enables you to model the data received from an inertial measurement unit consisting of a combination of gyroscope, accelerometer, and magnetometer.

Create a default `imuSensor` object.

```
IMU = imuSensor  
  
IMU =  
    imuSensor with properties:  
  
        IMUType: 'accel-gyro'  
        SampleRate: 100  
        Temperature: 25  
        Accelerometer: [1x1 accelparams]  
        Gyroscope: [1x1 gyroparams]  
        RandomStream: 'Global stream'
```

The `imuSensor` object, `IMU`, contains an idealized gyroscope and accelerometer. Use dot notation to view properties of the gyroscope.

```
IMU.Gyroscope
```

```
ans =  
    gyroparams with properties:  
  
        MeasurementRange: Inf      rad/s  
        Resolution: 0            (rad/s)/LSB  
        ConstantBias: [0 0 0]     rad/s  
        AxesMisalignment: [0 0 0] %  
  
        NoiseDensity: [0 0 0]     (rad/s)/√Hz  
        BiasInstability: [0 0 0]   rad/s  
        RandomWalk: [0 0 0]       (rad/s)*√Hz  
  
        TemperatureBias: [0 0 0]   (rad/s)/°C  
        TemperatureScaleFactor: [0 0 0] %/°C  
        AccelerationBias: [0 0 0]  (rad/s)/(m/s²)
```

Sensor properties are defined by corresponding parameter objects. For example, the gyroscope model used by the `imuSensor` is defined by an instance of the `gyroparams` class. You can modify properties of the gyroscope model using dot notation. Set the gyroscope measurement range to 4.3 rad/s.

```
IMU.Gyroscope.MeasurementRange = 4.3;
```

You can also set sensor properties to preset parameter objects. Create an `accelparams` object to mimic specific hardware, and then set the IMU `Accelerometer` property to the `accelparams` object. Display the `Accelerometer` property to verify the properties are correctly set.

```
SpecSheet1 = accelparams( ...  
    'MeasurementRange', 19.62, ...  
    'Resolution', 0.00059875, ...  
    'ConstantBias', 0.4905, ...  
    'AxesMisalignment', 2, ...
```

```

'NoiseDensity',0.003924, ...
'BiasInstability',0, ...
'TemperatureBias',[0.34335 0.34335 0.5886], ...
'TemperatureScaleFactor', 0.02);

IMU.Accelerometer = SpecSheet1;

IMU.Accelerometer

ans =
    accelparams with properties:

    MeasurementRange: 19.62          m/s2
    Resolution: 0.00059875          (m/s2)/LSB
    ConstantBias: [0.4905 0.4905 0.4905] m/s2
    AxesMisalignment: [2 2 2]          %

    NoiseDensity: [0.003924 0.003924 0.003924]      (m/s2)/√Hz
    BiasInstability: [0 0 0]          m/s2
    RandomWalk: [0 0 0]              (m/s2)*/√Hz

    TemperatureBias: [0.34335 0.34335 0.5886]      (m/s2)/°C
    TemperatureScaleFactor: [0.02 0.02 0.02]          %/°C

```

## Generate Ideal IMU Data from Stationary Input

Use the `imuSensor` System object™ to model receiving data from a stationary ideal IMU containing an accelerometer, gyroscope, and magnetometer.

Create an ideal IMU sensor model that contains an accelerometer, gyroscope, and magnetometer.

```

IMU = imuSensor('accel-gyro-mag')

IMU =
    imuSensor with properties:

    IMUType: 'accel-gyro-mag'
    SampleRate: 100
    Temperature: 25
    MagneticField: [27.5550 -2.4169 -16.0849]
    Accelerometer: [1x1 accelparams]
    Gyroscope: [1x1 gyroparams]
    Magnetometer: [1x1 magparams]
    RandomStream: 'Global stream'

```

Define the ground-truth, underlying motion of the IMU you are modeling. The acceleration and angular velocity are defined relative to the local NED coordinate system.

```

numSamples = 1000;
acceleration = zeros(numSamples,3);
angularVelocity = zeros(numSamples,3);

```

Call `IMU` with the ground-truth acceleration and angular velocity. The object outputs accelerometer readings, gyroscope readings, and magnetometer readings, as modeled by the properties of the

imuSensor System object. The accelerometer readings, gyroscope readings, and magnetometer readings are relative to the IMU sensor body coordinate system.

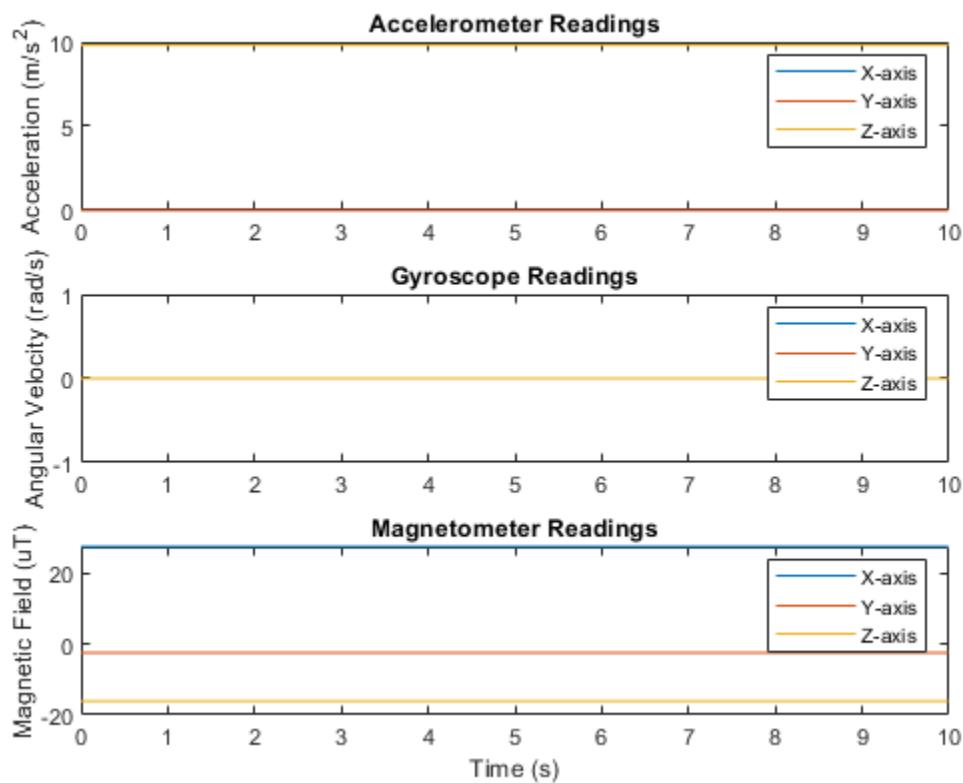
```
[accelReading,gyroReading,magReading] = IMU(acceleration,angularVelocity);
```

Plot the accelerometer readings, gyroscope readings, and magnetometer readings.

```
t = (0:(numSamples-1))/IMU.SampleRate;
subplot(3,1,1)
plot(t,accelReading)
legend('X-axis','Y-axis','Z-axis')
title('Accelerometer Readings')
ylabel('Acceleration (m/s^2)')

subplot(3,1,2)
plot(t,gyroReading)
legend('X-axis','Y-axis','Z-axis')
title('Gyroscope Readings')
ylabel('Angular Velocity (rad/s)')

subplot(3,1,3)
plot(t,magReading)
legend('X-axis','Y-axis','Z-axis')
title('Magnetometer Readings')
xlabel('Time (s)')
ylabel('Magnetic Field (uT)')
```



Orientation is not specified and the ground-truth motion is stationary, so the IMU sensor body coordinate system and the local NED coordinate system overlap for the entire simulation.

- Accelerometer readings: The z-axis of the sensor body corresponds to the Down-axis. The 9.8 m/s<sup>2</sup> acceleration along the z-axis is due to gravity.
- Gyroscope readings: The gyroscope readings are zero along each axis, as expected.
- Magnetometer readings: Because the sensor body coordinate system is aligned with the local NED coordinate system, the magnetometer readings correspond to the `MagneticField` property of `imuSensor`. The `MagneticField` property is defined in the local NED coordinate system.

## Model Rotating Six-Axis IMU Data

Use `imuSensor` to model data obtained from a rotating IMU containing an ideal accelerometer and an ideal magnetometer. Use `kinematicTrajectory` to define the ground-truth motion. Fuse the `imuSensor` model output using the `ecompass` function to determine orientation over time.

Define the ground-truth motion for a platform that rotates 360 degrees in four seconds, and then another 360 degrees in two seconds. Use `kinematicTrajectory` to output the orientation, acceleration, and angular velocity in the NED coordinate system.

```
fs = 100;
firstLoopNumSamples = fs*4;
secondLoopNumSamples = fs*2;
totalNumSamples = firstLoopNumSamples + secondLoopNumSamples;

traj = kinematicTrajectory('SampleRate',fs);

accBody = zeros(totalNumSamples,3);
angVelBody = zeros(totalNumSamples,3);
angVelBody(1:firstLoopNumSamples,3) = (2*pi)/4;
angVelBody(firstLoopNumSamples+1:end,3) = (2*pi)/2;

[~,orientationNED,~,accNED,angVelNED] = traj(accBody,angVelBody);
```

Create an `imuSensor` object with an ideal accelerometer and an ideal magnetometer. Call `IMU` with the ground-truth acceleration, angular velocity, and orientation to output accelerometer readings and magnetometer readings. Plot the results.

```
IMU = imuSensor('accel-mag','SampleRate',fs);

[accelReadings,magReadings] = IMU(accNED,angVelNED,orientationNED);

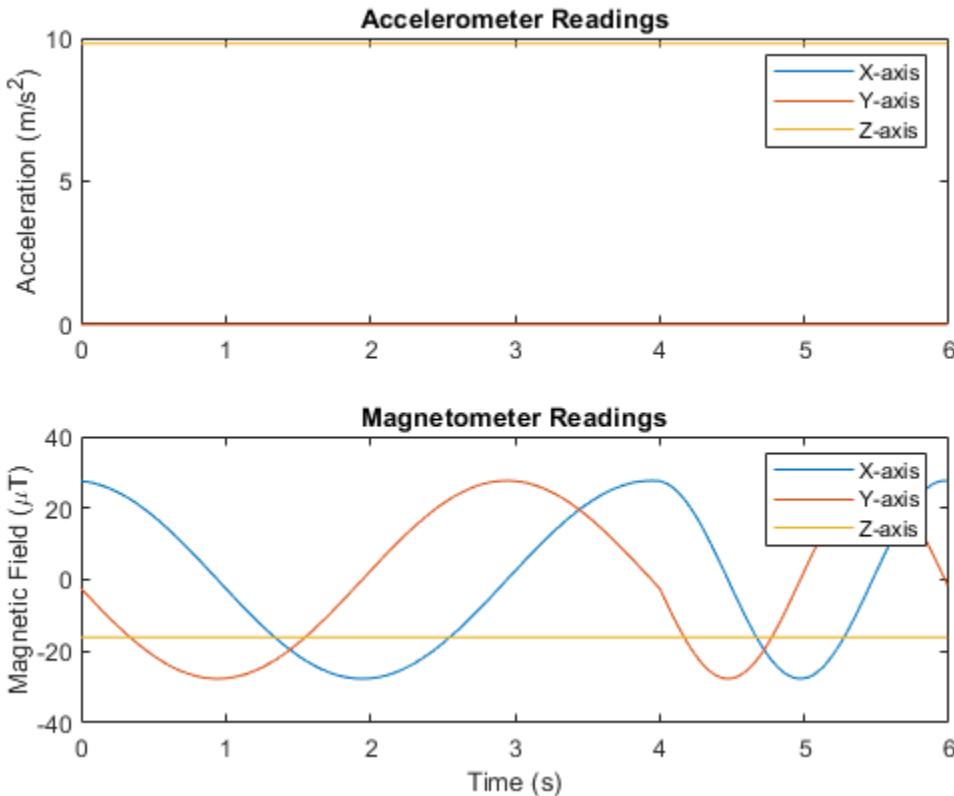
figure(1)
t = (0:(totalNumSamples-1))/fs;
subplot(2,1,1)
plot(t,accelReadings)
legend('X-axis','Y-axis','Z-axis')
ylabel('Acceleration (m/s^2)')
title('Accelerometer Readings')

subplot(2,1,2)
plot(t,magReadings)
legend('X-axis','Y-axis','Z-axis')
```

```

ylabel('Magnetic Field (\muT)')
xlabel('Time (s)')
title('Magnetometer Readings')

```



The accelerometer readings indicate that the platform has no translation. The magnetometer readings indicate that the platform is rotating around the  $z$ -axis.

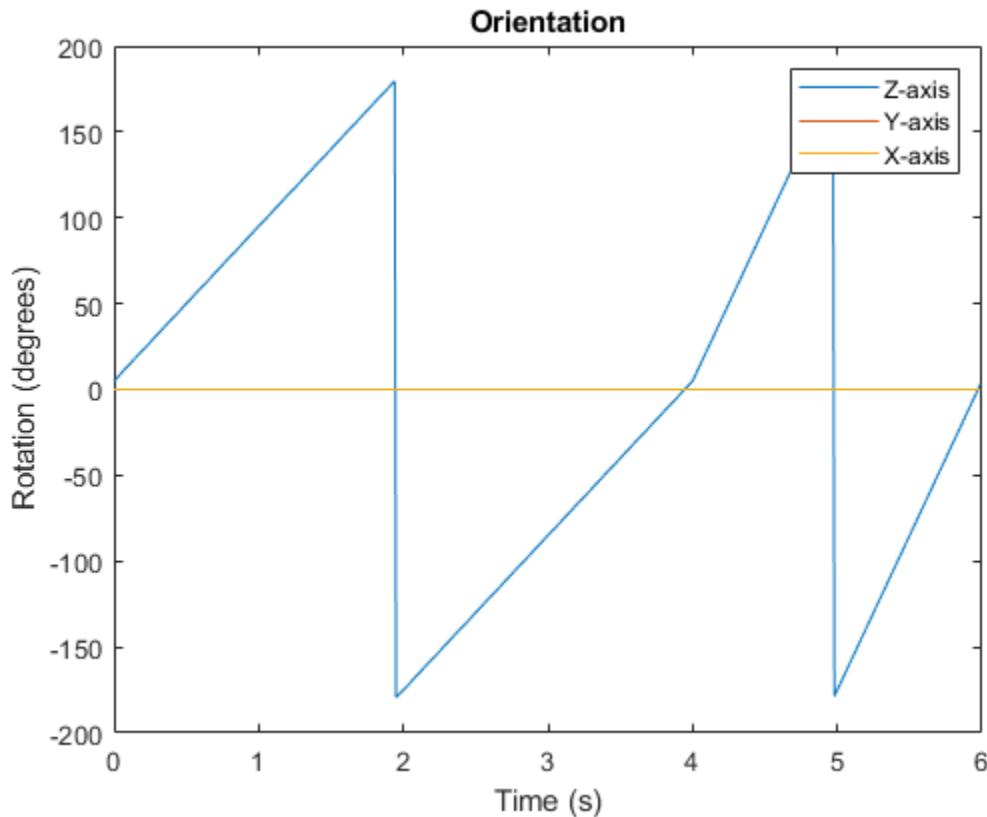
Feed the accelerometer and magnetometer readings into the `ecompass` function to estimate the orientation over time. The `ecompass` function returns orientation in quaternion format. Convert orientation to Euler angles and plot the results. The orientation plot indicates that the platform rotates about the  $z$ -axis only.

```

orientation = ecompass(accelReadings,magReadings);
orientationEuler = eulerd(orientation,'ZYX','frame');

figure(2)
plot(t,orientationEuler)
legend('Z-axis','Y-axis','X-axis')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation')

```



### Model Rotating Six-Axis IMU Data with Noise

Use `imuSensor` to model data obtained from a rotating IMU containing a realistic accelerometer and a realistic magnetometer. Use `kinematicTrajectory` to define the ground-truth motion. Fuse the `imuSensor` model output using the `ecompass` function to determine orientation over time.

Define the ground-truth motion for a platform that rotates 360 degrees in four seconds, and then another 360 degrees in two seconds. Use `kinematicTrajectory` to output the orientation, acceleration, and angular velocity in the NED coordinate system.

```
fs = 100;
firstLoopNumSamples = fs*4;
secondLoopNumSamples = fs*2;
totalNumSamples = firstLoopNumSamples + secondLoopNumSamples;

traj = kinematicTrajectory('SampleRate',fs);

accBody = zeros(totalNumSamples,3);
angVelBody = zeros(totalNumSamples,3);
angVelBody(1:firstLoopNumSamples,3) = (2*pi)/4;
angVelBody(firstLoopNumSamples+1:end,3) = (2*pi)/2;

[~,orientationNED,~,accNED,angVelNED] = traj(accBody,angVelBody);
```

Create an `imuSensor` object with a realistic accelerometer and a realistic magnetometer. Call `IMU` with the ground-truth acceleration, angular velocity, and orientation to output accelerometer readings and magnetometer readings. Plot the results.

```
IMU = imuSensor('accel-mag','SampleRate',fs);

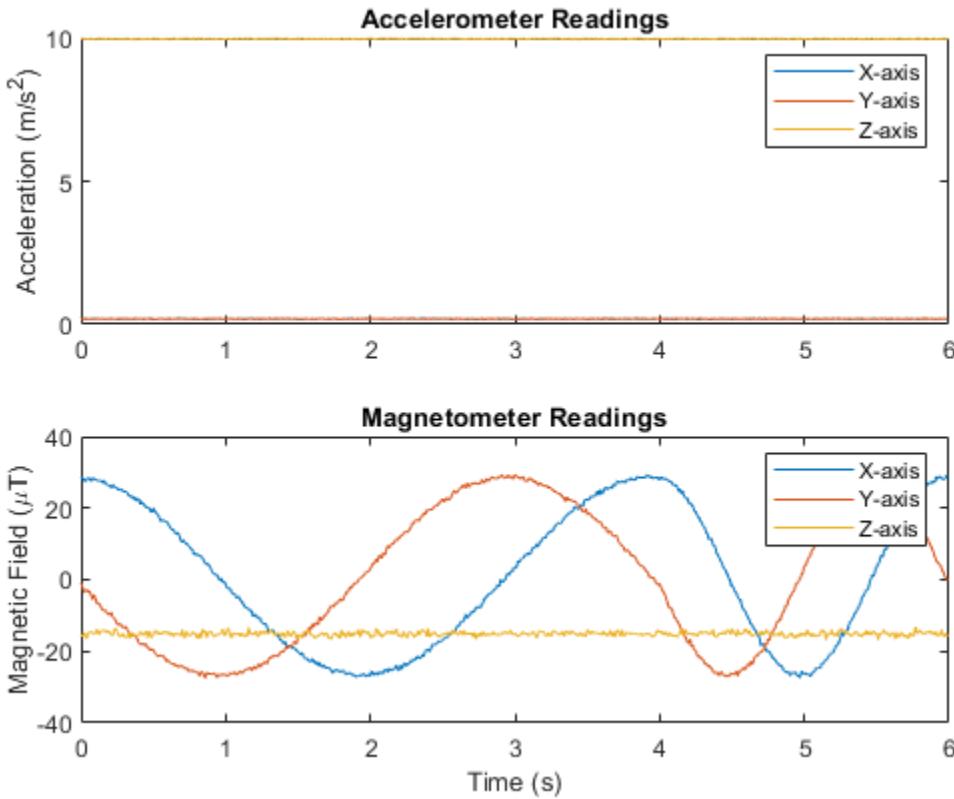
IMU.Accelerometer = accelparms( ...
    'MeasurementRange',19.62, ... % m/s^2
    'Resolution',0.0023936, ... % m/s^2 / LSB
    'TemperatureScaleFactor',0.008, ... % % / degree C
    'ConstantBias',0.1962, ... % m/s^2
    'TemperatureBias',0.0014715, ... % m/s^2 / degree C
    'NoiseDensity',0.0012361); % m/s^2 / Hz^(1/2)

IMU.Magnetometer = magparams( ...
    'MeasurementRange',1200, ... % uT
    'Resolution',0.1, ... % uT / LSB
    'TemperatureScaleFactor',0.1, ... % % / degree C
    'ConstantBias',1, ... % uT
    'TemperatureBias',[0.8 0.8 2.4], ... % uT / degree C
    'NoiseDensity',[0.6 0.6 0.9]/sqrt(100)); % uT / Hz^(1/2)

[accelReadings,magReadings] = IMU(accNED,angVelNED,orientationNED);

figure(1)
t = (0:(totalNumSamples-1))/fs;
subplot(2,1,1)
plot(t,accelReadings)
legend('X-axis','Y-axis','Z-axis')
ylabel('Acceleration (m/s^2)')
title('Accelerometer Readings')

subplot(2,1,2)
plot(t,magReadings)
legend('X-axis','Y-axis','Z-axis')
ylabel('Magnetic Field (\mu T)')
xlabel('Time (s)')
title('Magnetometer Readings')
```

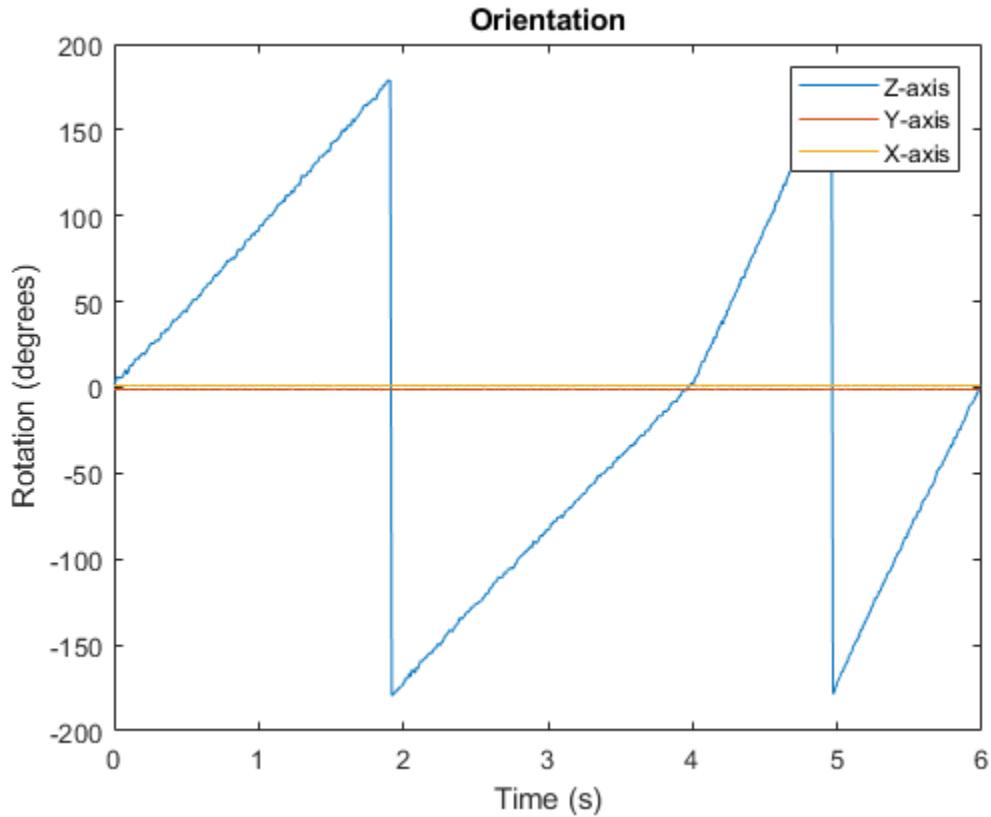


The accelerometer readings indicate that the platform has no translation. The magnetometer readings indicate that the platform is rotating around the  $z$ -axis.

Feed the accelerometer and magnetometer readings into the `ecompass` function to estimate the orientation over time. The `ecompass` function returns orientation in quaternion format. Convert orientation to Euler angles and plot the results. The orientation plot indicates that the platform rotates about the  $z$ -axis only.

```
orientation = ecompass(accelReadings,magReadings);
orientationEuler = eulerd(orientation,'ZYX','frame');

figure(2)
plot(t,orientationEuler)
legend('Z-axis','Y-axis','X-axis')
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation')
```



%

### Model Tilt Using Gyroscope and Accelerometer Readings

Model a tilting IMU that contains an accelerometer and gyroscope using the `imuSensor` System object™. Use ideal and realistic models to compare the results of orientation tracking using the `imufilter` System object.

Load a struct describing ground-truth motion and a sample rate. The motion struct describes sequential rotations:

- 1 yaw: 120 degrees over two seconds
- 2 pitch: 60 degrees over one second
- 3 roll: 30 degrees over one-half second
- 4 roll: -30 degrees over one-half second
- 5 pitch: -60 degrees over one second
- 6 yaw: -120 degrees over two seconds

In the last stage, the motion struct combines the 1st, 2nd, and 3rd rotations into a single-axis rotation. The acceleration, angular velocity, and orientation are defined in the local NED coordinate system.

```
load y120p60r30.mat motion fs
accNED = motion.Acceleration;
angVelNED = motion.AngularVelocity;
orientationNED = motion.Orientation;

numSamples = size(motion.Orientation,1);
t = (0:(numSamples-1)).'/fs;
```

Create an ideal IMU sensor object and a default IMU filter object.

```
IMU = imuSensor('accel-gyro','SampleRate',fs);
aFilter = imufilter('SampleRate',fs);
```

In a loop:

- 1 Simulate IMU output by feeding the ground-truth motion to the IMU sensor object.
- 2 Filter the IMU output using the default IMU filter object.

```
orientation = zeros(numSamples,1,'quaternion');
for i = 1:numSamples

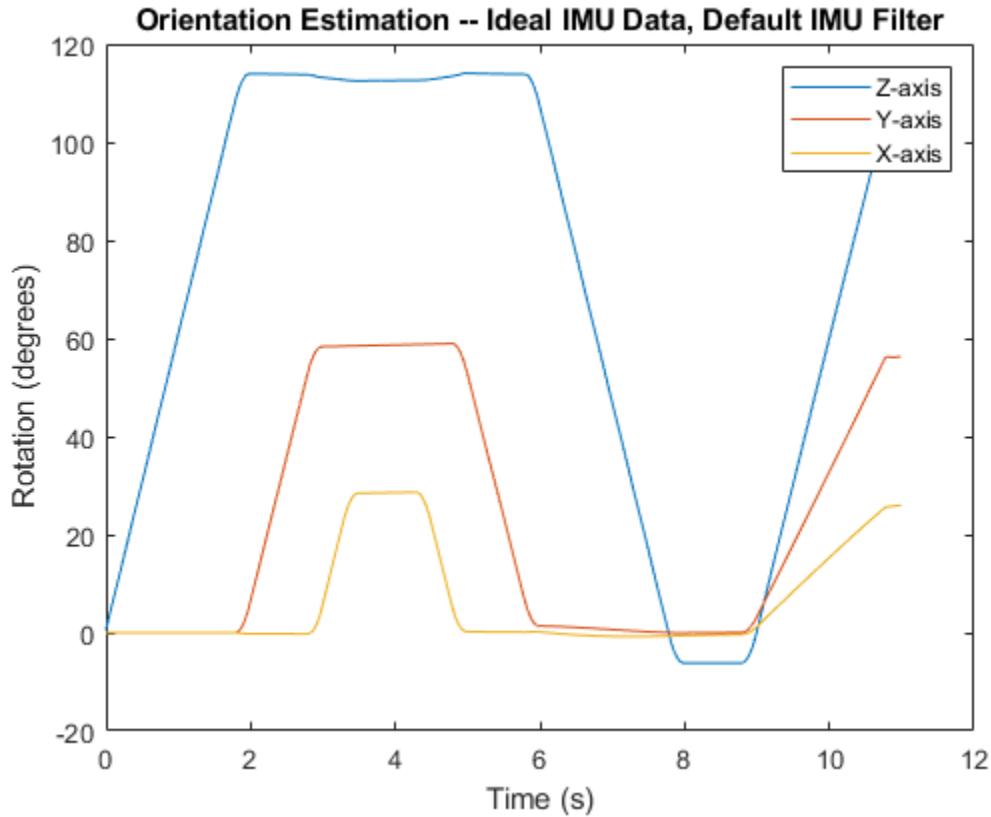
    [accelBody,gyroBody] = IMU(accNED(i,:),angVelNED(i,:),orientationNED(i,:));

    orientation(i) = aFilter(accelBody,gyroBody);

end
release(aFilter)
```

Plot the orientation over time.

```
figure(1)
plot(t,eulerd(orientation,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Ideal IMU Data, Default IMU Filter')
legend('Z-axis','Y-axis','X-axis')
```



Modify properties of your `imuSensor` to model real-world sensors. Run the loop again and plot the orientation estimate over time.

```
IMU.Accelerometer = accelparams( ...
    'MeasurementRange',19.62, ...
    'Resolution',0.00059875, ...
    'ConstantBias',0.4905, ...
    'AxesMisalignment',2, ...
    'NoiseDensity',0.003924, ...
    'BiasInstability',0, ...
    'TemperatureBias',[0.34335 0.34335 0.5886], ...
    'TemperatureScaleFactor',0.02);
IMU.Gyroscope = gyroparams( ...
    'MeasurementRange',4.3633, ...
    'Resolution',0.00013323, ...
    'AxesMisalignment',2, ...
    'NoiseDensity',8.7266e-05, ...
    'TemperatureBias',0.34907, ...
    'TemperatureScaleFactor',0.02, ...
    'AccelerationBias',0.00017809, ...
    'ConstantBias',[0.3491,0.5,0]);
orientationDefault = zeros(numSamples,1,'quaternion');
for i = 1:numSamples
    [accelBody,gyroBody] = IMU(accNED(i,:),angVelNED(i,:),orientationNED(i,:));
```

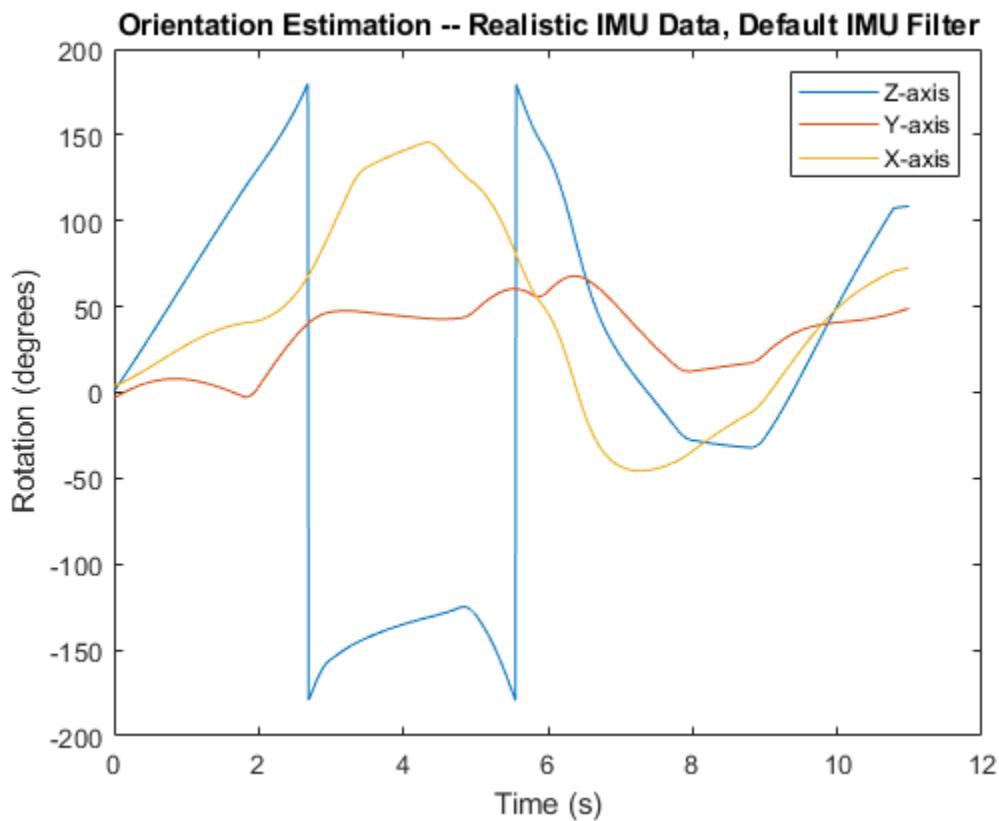
```

orientationDefault(i) = aFilter(accelBody,gyroBody);

end
release(aFilter)

figure(2)
plot(t,eulerd(orientationDefault,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Realistic IMU Data, Default IMU Filter')
legend('Z-axis','Y-axis','X-axis')

```



The ability of the `imufilter` to track the ground-truth data is significantly reduced when modeling a realistic IMU. To improve performance, modify properties of your `imufilter` object. These values were determined empirically. Run the loop again and plot the orientation estimate over time.

```

aFilter.GyroscopeNoise      = 7.6154e-7;
aFilter.AccelerometerNoise = 0.0015398;
aFilter.GyroscopeDriftNoise = 3.0462e-12;
aFilter.LinearAccelerationNoise = 0.00096236;
aFilter.InitialProcessNoise = aFilter.InitialProcessNoise*10;

orientationNondefault = zeros(numSamples,1,'quaternion');
for i = 1:numSamples
    [accelBody,gyroBody] = IMU(accNED(i,:),angVelNED(i,:),orientationNED(i,:));
    orientationNondefault(i) = aFilter(accelBody,gyroBody);

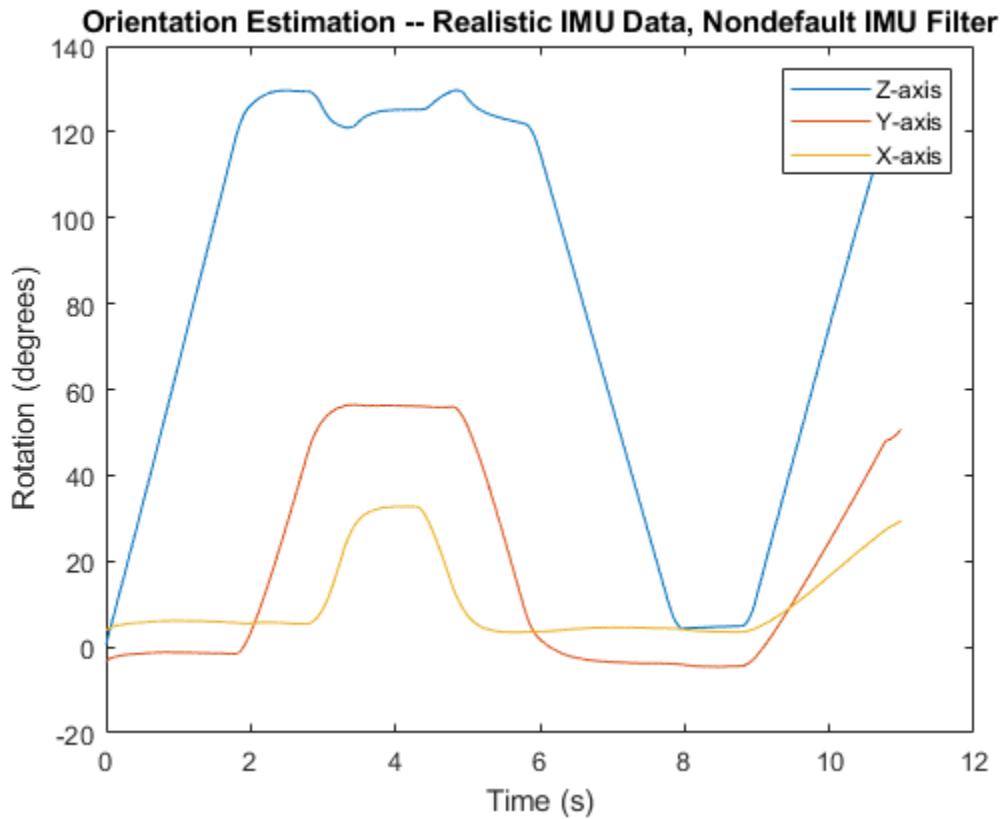
```

```

end
release(aFilter)

figure(3)
plot(t,eulerd(orientationNondefault,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Realistic IMU Data, Nondefault IMU Filter')
legend('Z-axis','Y-axis','X-axis')

```



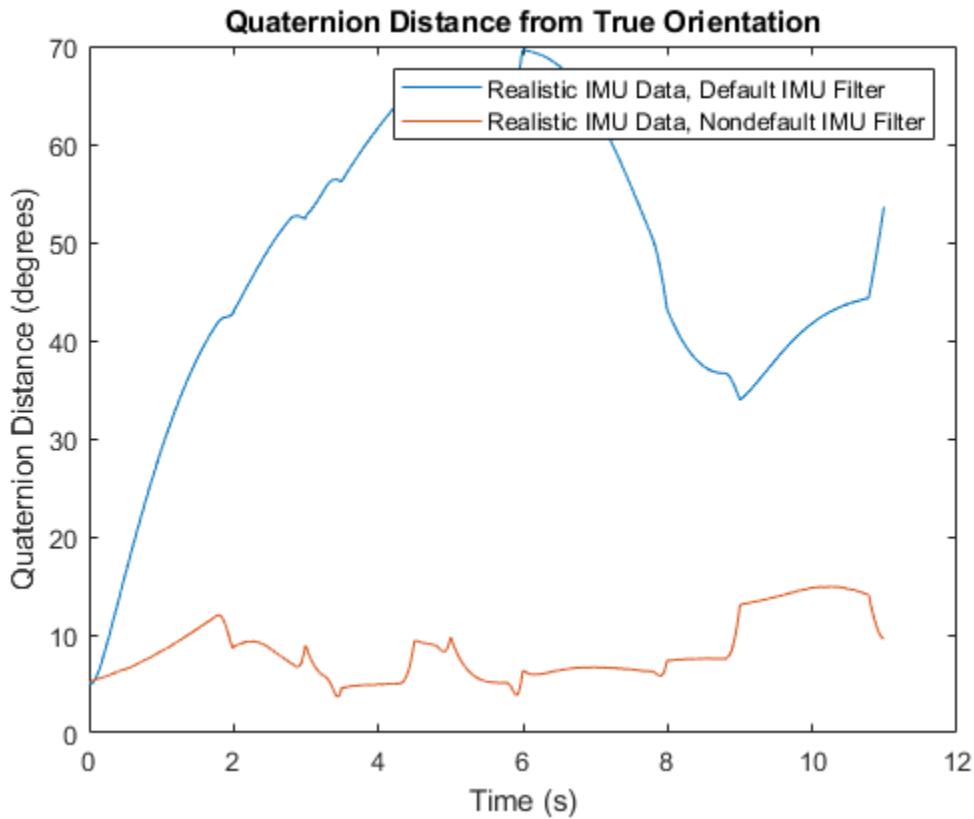
To quantify the improved performance of the modified `imufilter`, plot the quaternion distance between the ground-truth motion and the orientation as returned by the `imufilter` with default and nondefault properties.

```

qDistDefault = rad2deg(dist(orientationNED,orientationDefault));
qDistNondefault = rad2deg(dist(orientationNED,orientationNondefault));

figure(4)
plot(t,[qDistDefault,qDistNondefault])
title('Quaternion Distance from True Orientation')
legend('Realistic IMU Data, Default IMU Filter', ...
       'Realistic IMU Data, Nondefault IMU Filter')
xlabel('Time (s)')
ylabel('Quaternion Distance (degrees)')

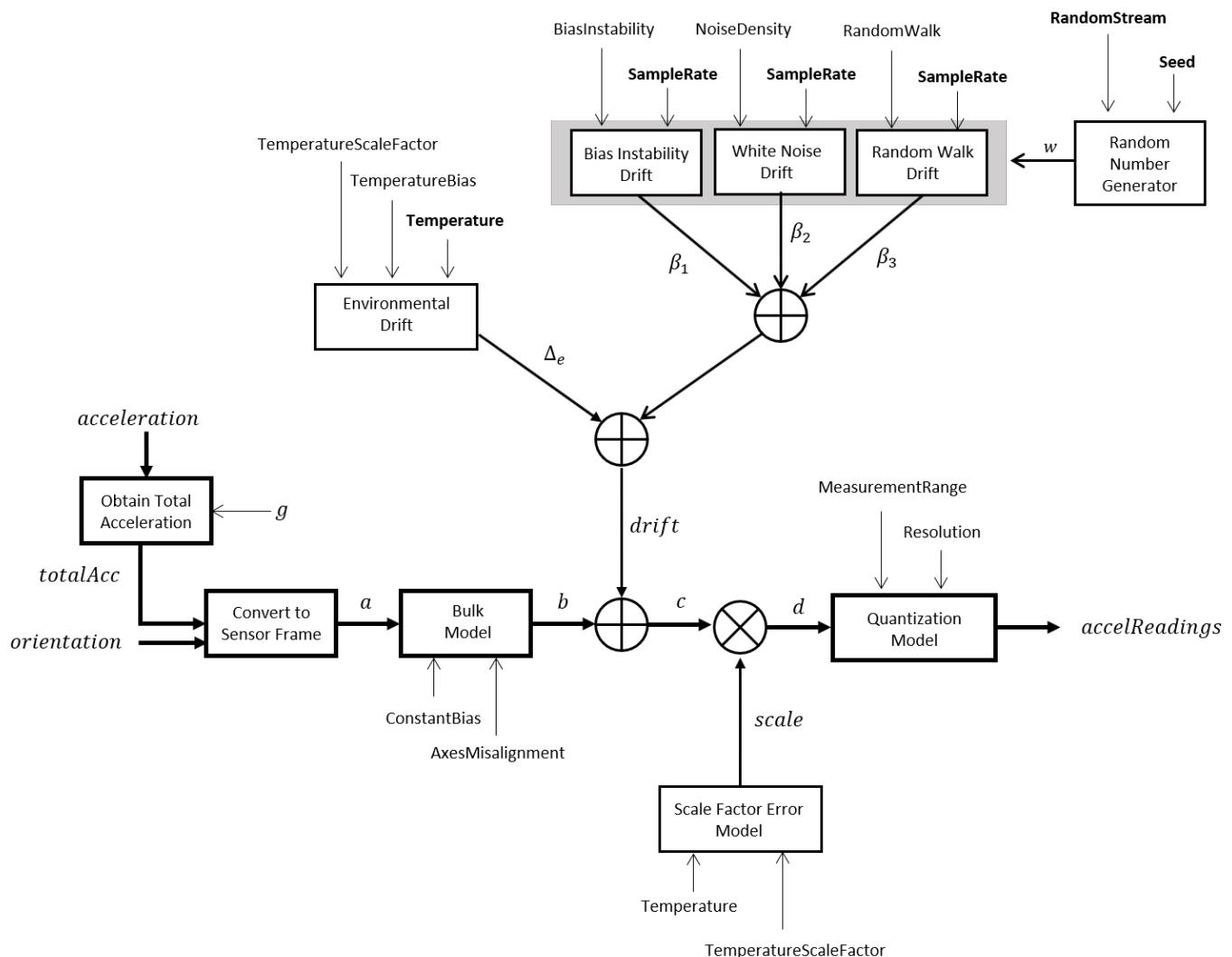
```



## Algorithms

### Accelerometer

The accelerometer model uses the ground-truth orientation and acceleration inputs and the `imuSensor` and `accelparams` properties to model accelerometer readings.



#### Obtain Total Acceleration

To obtain the total acceleration (*totalAcc*), the acceleration is preprocessed by negating and adding the gravity constant vector ( $g = [0; 0; 9.8] \text{ m/s}^2$ ) as:

$$totalAcc = -acceleration + g$$

#### Convert to Sensor Frame

Then the total acceleration is converted from the local navigation frame to the sensor frame using:

$$a = (orientation)(totalAcc)^T$$

If the orientation is input in quaternion form, it is converted to a rotation matrix before processing.

#### Bulk Model

The ground-truth acceleration in the sensor frame, *a*, passes through the bulk model, which adds axes misalignment and bias:

$$b = \begin{pmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{pmatrix} (a^T)^T + \text{ConstantBias}$$

where ConstantBias is a property of `accelparams`, and  $\alpha_1$ ,  $\alpha_2$ , and  $\alpha_3$  are given by the first, second, and third elements of the AxesMisalignment property of `accelparams`.

### Bias Instability Drift

The bias instability drift is modeled as white noise biased and then filtered:

$$\beta_1 = h_1 * (w)(\text{BiasInstability})$$

where BiasInstability is a property of `accelparams`, and  $h_1$  is a filter defined by the SampleRate property:

$$H_1(z) = \frac{1}{1 - \frac{1}{2}z^{-1}}$$

### White Noise Drift

White noise drift is modeled by multiplying elements of the white noise random stream by the standard deviation:

$$\beta_2 = (w)\left(\sqrt{\frac{\text{SampleRate}}{2}}\right)(\text{NoiseDensity})$$

where SampleRate is an `imuSensor` property, and NoiseDensity is an `accelparams` property. Elements of  $w$  are random numbers given by settings of the `imuSensor` random stream.

### Random Walk Drift

The random walk drift is modeled by biasing elements of the white noise random stream and then filtering:

$$\beta_3 = h_2 * (w)\left(\frac{\text{RandomWalk}}{\sqrt{\frac{\text{SampleRate}}{2}}}\right)$$

where RandomWalk is a property of `accelparams`, SampleRate is a property of `imuSensor`, and  $h_2$  is a filter defined as:

$$H_2(z) = \frac{1}{1 - z^{-1}}$$

### Environmental Drift Noise

The environmental drift noise is modeled by multiplying the temperature difference from a standard with the temperature bias:

$$\Delta_e = (\text{Temperature} - 25)(\text{TemperatureBias})$$

where Temperature is a property of `imuSensor`, and TemperatureBias is a property of `acelparams`. The constant 25 corresponds to a standard temperature.

### Scale Factor Error Model

The temperature scale factor error is modeled as:

$$scaleFactorError = 1 + \left( \frac{\text{Temperature} - 25}{100} \right) (\text{TemperatureScaleFactor})$$

where Temperature is a property of `imuSensor`, and TemperatureScaleFactor is a property of `acelparams`. The constant 25 corresponds to a standard temperature.

### Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

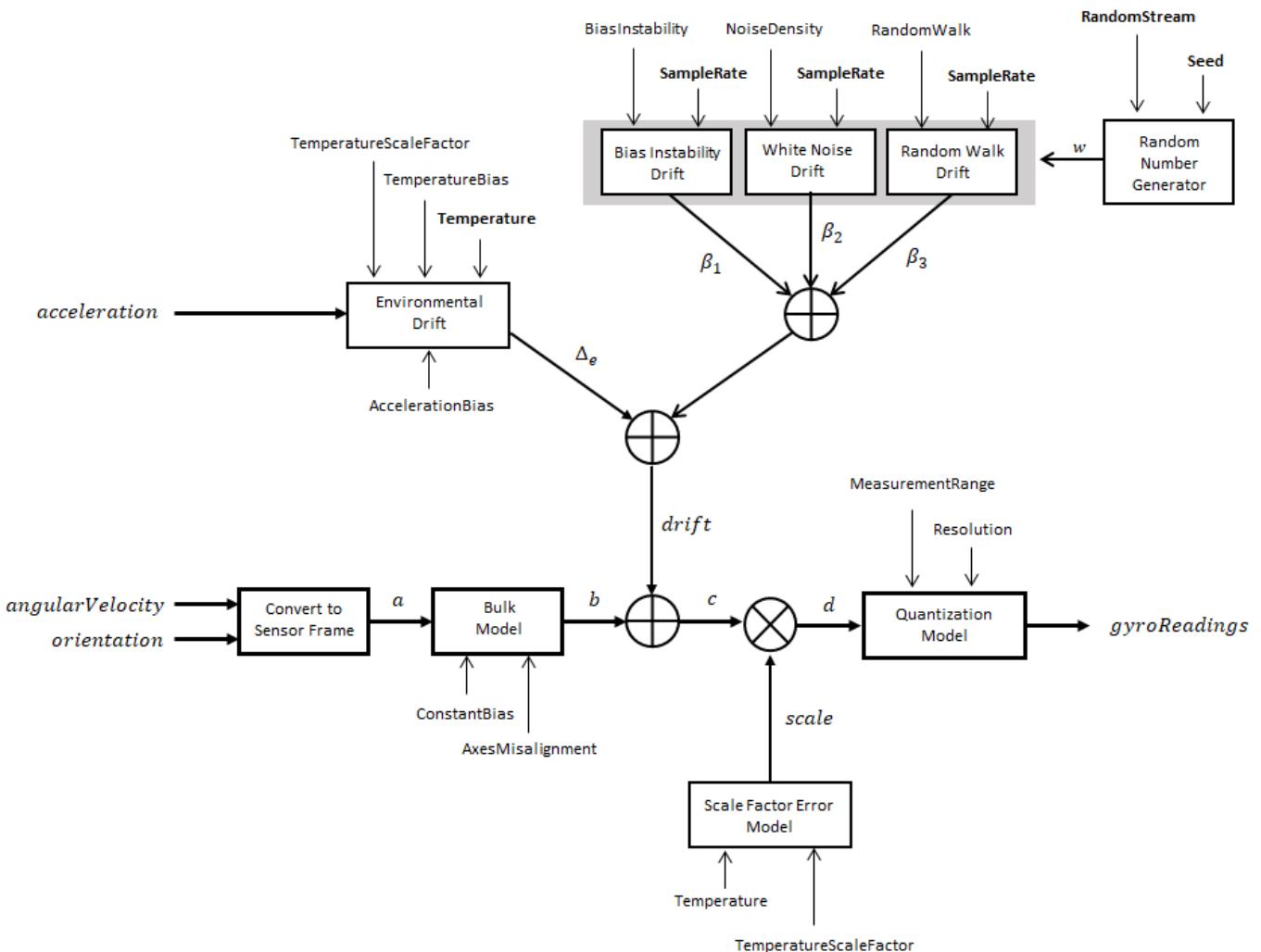
and then setting the resolution:

$$\text{accelReadings} = (\text{Resolution}) \left( \text{round} \left( \frac{e}{\text{Resolution}} \right) \right)$$

where MeasurementRange is a property of `acelparams`.

### Gyroscope

The gyroscope model uses the ground-truth orientation, acceleration, and angular velocity inputs, and the `imuSensor` and `gyroparms` properties to model accelerometer readings.



### Convert to Sensor Frame

The ground-truth angular velocity is converted from the local frame to the sensor frame using the ground-truth orientation:

$$a = (\text{orientation})(\text{angularVelocity})^T$$

If the orientation is input in quaternion form, it is converted to a rotation matrix before processing.

### Bulk Model

The ground-truth angular velocity in the sensor frame, *a*, passes through the bulk model, which adds axes misalignment and bias:

$$b = \begin{pmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{pmatrix} (a^T)^T + \text{ConstantBias}$$

where ConstantBias is a property of `gyroparams`, and  $\alpha_1$ ,  $\alpha_2$ , and  $\alpha_3$  are given by the first, second, and third elements of the AxesMisalignment property of `gyroparams`.

#### Bias Instability Drift

The bias instability drift is modeled as white noise biased and then filtered:

$$\beta_1 = h_1 * (w)(\text{BiasInstability})$$

where BiasInstability is a property of `gyroparams` and  $h_1$  is a filter defined by the SampleRate property:

$$H_1(z) = \frac{1}{1 - \frac{1}{2}z^{-1}}$$

#### White Noise Drift

White noise drift is modeled by multiplying elements of the white noise random stream by the standard deviation:

$$\beta_2 = (w)\left(\sqrt{\frac{\text{SampleRate}}{2}}\right)(\text{NoiseDensity})$$

where SampleRate is an `imuSensor` property, and NoiseDensity is an `gyroparams` property. The elements of  $w$  are random numbers given by settings of the `imuSensor` random stream.

#### Random Walk Drift

The random walk drift is modeled by biasing elements of the white noise random stream and then filtering:

$$\beta_3 = h_2 * (w)\left(\frac{\text{RandomWalk}}{\sqrt{\frac{\text{SampleRate}}{2}}}\right)$$

where RandomWalk is a property of `gyroparams`, SampleRate is a property of `imuSensor`, and  $h_2$  is a filter defined as:

$$H_2(z) = \frac{1}{1 - z^{-1}}$$

#### Environmental Drift Noise

The environmental drift noise is modeled by multiplying the temperature difference from a standard with the temperature bias:

$$\Delta_e = (\text{Temperature} - 25)(\text{TemperatureBias})$$

where Temperature is a property of `imuSensor`, and TemperatureBias is a property of `gyroparams`. The constant 25 corresponds to a standard temperature.

### Scale Factor Error Model

The temperature scale factor error is modeled as:

$$scaleFactorError = 1 + \left( \frac{\text{Temperature} - 25}{100} \right) (\text{TemperatureScaleFactor})$$

where Temperature is a property of `imuSensor`, and TemperatureScaleFactor is a property of `gyroparams`. The constant 25 corresponds to a standard temperature.

### Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

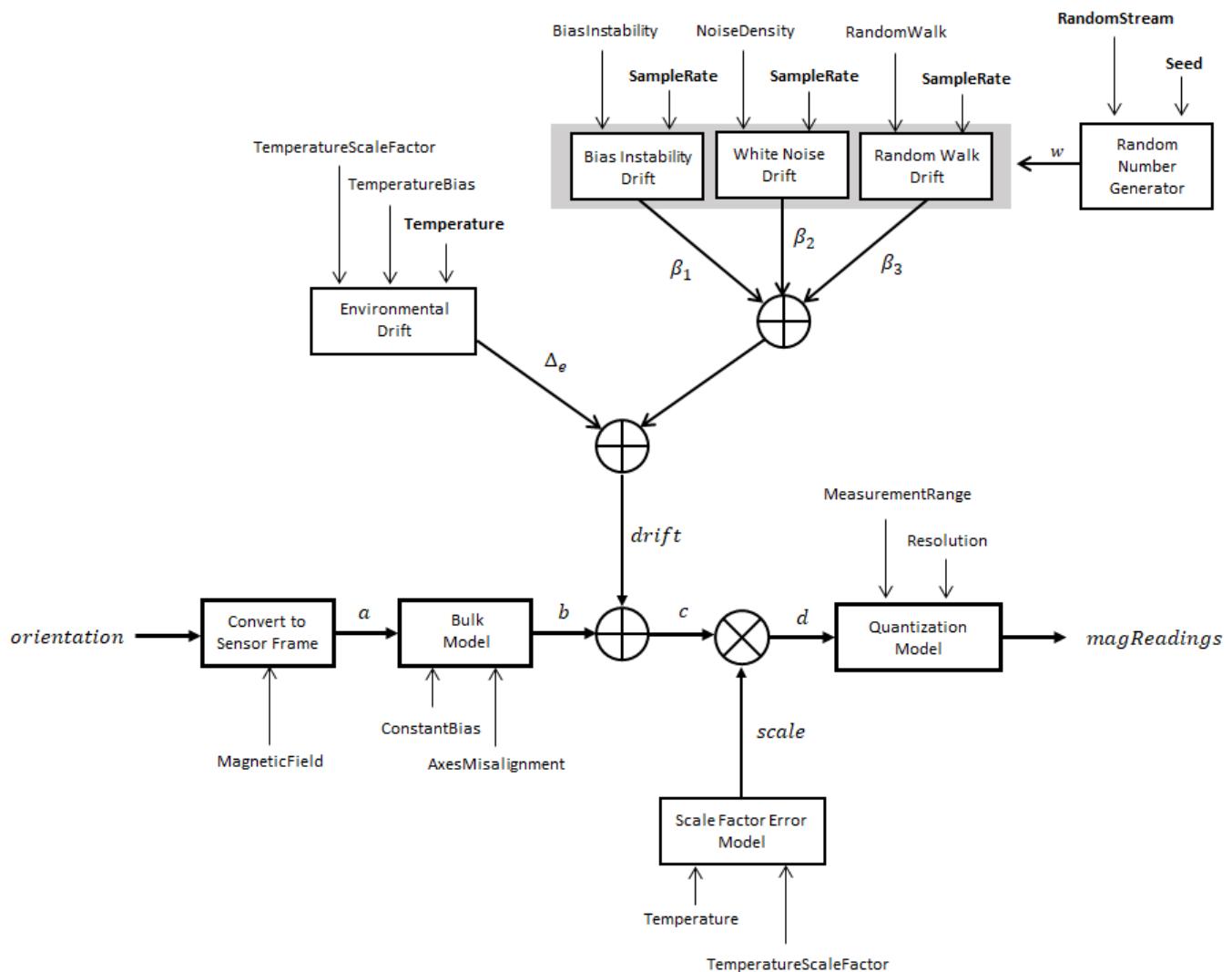
and then setting the resolution:

$$gyroReadings = (\text{Resolution}) \left( \text{round} \left( \frac{e}{\text{Resolution}} \right) \right)$$

where MeasurementRange is a property of `gyroparams`.

### Magnetometer

The magnetometer model uses the ground-truth orientation and acceleration inputs, and the `imuSensor` and `magparams` properties to model magnetometer readings.



#### Convert to Sensor Frame

The ground-truth acceleration is converted from the local frame to the sensor frame using the ground-truth orientation:

$$a = (\text{orientation})(\text{totalAcc})^T$$

If the orientation is input in quaternion form, it is converted to a rotation matrix before processing.

#### Bulk Model

The ground-truth acceleration in the sensor frame, *a*, passes through the bulk model, which adds axes misalignment and bias:

$$b = \begin{pmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{pmatrix} (a^T)^T + \text{ConstantBias}$$

where ConstantBias is a property of `magparams`, and  $\alpha_1$ ,  $\alpha_2$ , and  $\alpha_3$  are given by the first, second, and third elements of the AxesMisalignment property of `magparams`.

### Bias Instability Drift

The bias instability drift is modeled as white noise biased and then filtered:

$$\beta_1 = h_1 * (w)(\text{BiasInstability})$$

where BiasInstability is a property of `magparams` and  $h_1$  is a filter defined by the SampleRate property:

$$H_1(z) = \frac{1}{1 - \frac{1}{2}z^{-1}}$$

### White Noise Drift

White noise drift is modeled by multiplying elements of the white noise random stream by the standard deviation:

$$\beta_2 = (w)\left(\sqrt{\frac{\text{SampleRate}}{2}}\right)(\text{NoiseDensity})$$

where SampleRate is an `imuSensor` property, and NoiseDensity is an `magparams` property. The elements of  $w$  are random numbers given by settings of the `imuSensor` random stream.

### Random Walk Drift

The random walk drift is modeled by biasing elements of the white noise random stream and then filtering:

$$\beta_3 = h_2 * (w)\left(\frac{\text{RandomWalk}}{\sqrt{\frac{\text{SampleRate}}{2}}}\right)$$

where RandomWalk is a property of `magparams`, SampleRate is a property of `imuSensor`, and  $h_2$  is a filter defined as:

$$H_2(z) = \frac{1}{1 - z^{-1}}$$

### Environmental Drift Noise

The environmental drift noise is modeled by multiplying the temperature difference from a standard with the temperature bias:

$$\Delta_e = (\text{Temperature} - 25)(\text{TemperatureBias})$$

where Temperature is a property of `imuSensor`, and TemperatureBias is a property of `magparams`. The constant 25 corresponds to a standard temperature.

### Scale Factor Error Model

The temperature scale factor error is modeled as:

$$scaleFactorError = 1 + \left( \frac{\text{Temperature} - 25}{100} \right) (\text{TemperatureScaleFactor})$$

where Temperature is a property of `imuSensor`, and TemperatureScaleFactor is a property of `magparams`. The constant 25 corresponds to a standard temperature.

### Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

and then setting the resolution:

$$\text{magReadings} = (\text{Resolution}) \left( \text{round} \left( \frac{e}{\text{Resolution}} \right) \right)$$

where MeasurementRange is a property of `magparams`.

## 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

### Classes

`accelparams` | `gyroparams` | `magparams`

### Objects

`gpsSensor` | `insSensor`

### Topics

“Model IMU, GPS, and INS/GPS”

## Introduced in R2018b

# loadparams

Load sensor parameters from JSON file

## Syntax

```
loadparams(sensor,file,PN)
```

## Description

`loadparams(sensor,file,PN)` configures the `imuSensor` object, `sensor`, to match the parameters in the PN part of a JSON file, `File`.

## Examples

### Load Pre-defined Parameters in `imuSensor`

Create an `imuSensor` system object.

```
s = imuSensor;
```

Load a JSON file.

```
fn = fullfile(matlabroot,'toolbox','shared',...
    'positioning','positioningdata','generic.json');
```

Here is a screen shot of the JSON file with some parts collapsed.

```
{
  "GenericLowCost9Axis": {
    "Accelerometer": {
      "MeasurementRange": 19.6133,
      "Resolution": 0.0023928,
      "ConstantBias": [0.19, 0.19, 0.19],
      "AxesMisalignment": [0, 0, 0],
      "NoiseDensity": [0.0012356, 0.0012356, 0.0012356],
      "BiasInstability": [0, 0, 0],
      "RandomWalk": [0, 0, 0],
      "TemperatureBias": [0, 0, 0],
      "TemperatureScaleFactor": [0, 0, 0]
    },
    "Gyroscope": {
      [redacted],
      "Magnetometer": {
        [redacted]
      }
    },
    "GenericLowCost6Axis": {
      [redacted]
    }
}
```

Configure the object as a 6-axis sensor.

```
loadparams(s,fn,'GenericLowCost6Axis')
s

s =
    imuSensor with properties:

        IMUType: 'accel-gyro'
        SampleRate: 100
        Temperature: 25
        Accelerometer: [1x1 accelparams]
        Gyroscope: [1x1 gyroparams]
        RandomStream: 'Global stream'
```

Configure the object as a 9-axis sensor.

```
loadparams(s,fn,'GenericLowCost9Axis')
s

s =
    imuSensor with properties:

        IMUType: 'accel-gyro-mag'
        SampleRate: 100
        Temperature: 25
        MagneticField: [27.5550 -2.4169 -16.0849]
        Accelerometer: [1x1 accelparams]
        Gyroscope: [1x1 gyroparams]
        Magnetometer: [1x1 magparams]
        RandomStream: 'Global stream'
```

## Input Arguments

### **sensor — IMU sensor**

imuSensor object

IMU sensor, specified as an `imuSensor` system object.

### **file — JSON file**

.json file

JavaScript Object Notation (JSON) format file, specified as a `.json` file.

### **PN — Part name**

string

Part name in a JSON file, specified as a string.

## See Also

`imuSensor`

**Introduced in R2020a**

# trackBranchHistory

Track-oriented MHT branching and branch history

## Description

The `trackBranchHistory` System object is a track-oriented, multi-hypothesis tracking (MHT) branch history manager. The object maintains a history of track branches (hypotheses) that are based on the results of an assignment algorithm, such as the algorithm used by the `assignTOMHT` function. Given the most recent scan of a set of sensors, the assignment algorithm results include:

- The assignments of sensor detections to specific track branches
- The unassigned track branches
- The unassigned detections

The `trackBranchHistory` object creates, updates, and deletes track branches as needed and maintains the track branch history for a specified number of scans. Each track and branch stored in the object has a unique ID. To view a table of track branches for the current history, use the `getHistory` function. To compute branch clusters and incompatible branches, specify the track branch history as an input to the `clusterTrackBranches` function.

To create a branch history manager and update the branch history:

- 1 Create the `trackBranchHistory` 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

```
branchHistoryMgr = trackBranchHistory  
branchHistoryMgr = trackBranchHistory(Name,Value)
```

#### Description

`branchHistoryMgr = trackBranchHistory` creates a `trackBranchHistory` System object, `branchHistoryMgr`, with default property values.

`branchHistoryMgr = trackBranchHistory(Name,Value)` sets properties for the `trackBranchHistory` object by using one or more name-value pairs. For example, `branchHistoryMgr = trackBranchHistory('MaxNumTracks',250,'MaxNumTrackBranches',5)` creates a `trackBranchHistory` object that can maintain a maximum of 250 tracks and 5 track branches per track. Enclose property names in quotes. Specified property values can be any numeric data type, but they must all be of the same data 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](#).

### **MaxNumSensors — Maximum number of sensors**

20 (default) | positive integer

Maximum number of sensors, specified as a positive integer.

### **MaxNumHistoryScans — Maximum number of scans maintained in branch history**

4 (default) | positive integer

Maximum number of scans maintained in the branch history, specified as a positive integer. Typical values are from 2 to 6. Higher values increase the computational load.

### **MaxNumTracks — Maximum number of tracks**

200 (default) | positive integer

Maximum number of tracks that the branch history manager can maintain, specified as a positive integer.

### **MaxNumTrackBranches — Maximum number of branches per track**

3 (default) | positive integer

Maximum number of branches per track that the branch history manager can maintain, specified as a positive integer.

## Usage

### Syntax

```
history = branchHistoryMgr(assignments,unassignedTracks,unassignedDetections,  
originatingSensor)
```

### Description

`history = branchHistoryMgr(assignments,unassignedTracks,unassignedDetections,  
originatingSensor)` returns the branch history based on the results of an assignment algorithm. Specify the assignments of detections to branches, the lists of unassigned tracks and unassigned detections, and the IDs of the sensors from which the detections originated. The inputs can be of any numeric data type.

The `assignTOMHT` function returns assignment results as `uint32` values, but the inputs to `branchHistoryMgr` can be of any numeric data type.

## Input Arguments

### **assignments — Assignment of track branches to detections**

*P*-by-2 matrix of integers

Assignment of track branches to detections, specified as a *P*-by-2 matrix of integers, where *P* is the number of assignments. The first column lists the track branch indices. The second column lists the detection indices. The same branch can be assigned to multiple detections. The same detection can be assigned to multiple branches.

For example, if `assignments = [1 1; 1 2; 2 1; 2 2]`, the rows of `assignments` specify these assignments:

- `[1 1]` — Branch 1 was assigned to detection 1.
- `[1 2]` — Branch 1 was assigned to detection 2.
- `[2 1]` — Branch 2 was assigned to detection 1.
- `[2 2]` — Branch 2 was assigned to detection 2.

### **unassignedTracks — Indices of unassigned track branches**

*Q*-by-1 vector of integers

Indices of unassigned track branches, specified as a *Q*-by-1 vector of integers, where *Q* is the number of unassigned track branches. Each element of `unassignedTracks` must correspond to the indices of a track branch currently stored in the `trackBranchHistory` System object.

### **unassignedDetections — Indices of unassigned detections**

*R*-by-1 vector of integers

Indices of unassigned detections, specified as an *R*-by-1 vector of integers, where *R* is the number of unassigned detections. Each unassigned detection results in a new track branch.

### **originatingSensor — Indices of sensors from which each detection originated**

1-by-*L* vector of integers

Indices of sensors from which each detection originated, specified as a 1-by-*L* vector of integers, where *L* is the number of detections. The *i*th element of `originatingSensor` corresponds to the `SensorIndex` property value of `objectDetection` object *i*.

## Output Arguments

### **history — Branch history**

matrix of integers

Branch history, returned as a matrix of integers.

Each row of `history` represents a unique track branch. `history` has  $3+(D \times S)$  columns, where *D* is the number of maintained scans (the history depth) and *S* is the maximum number of maintained sensors. The first three columns represent the following information about each track branch:

- **TrackID** — ID of the track that is associated with the branch. Track branches that are assumed to have originated from the same target have the same track ID. If a branch originates from an unassigned detection, that branch gets a new track ID.
- **ParentID** — ID of the parent branch, that is, the branch from which the current branch originated. Branches that were created from the same parent have the same `ParentID`. A

ParentID of 0 indicates a new track. These tracks are created from hypotheses corresponding to unassigned detections.

- BranchID — Unique ID of track branch. Every branch created from an unassigned detection or assignment gets a new branch ID.

The remaining  $D \times S$  columns contain the IDs of the detections assigned to each branch. A branch can be assigned to at most one detection per scan and per sensor. The table shows the organization of these columns with sample detections.  $N$  is the number of scans. A value of 0 means that the sensor at that scan does not have a detection assigned to it.

Scan $N$				Scan $N - 1$				...	Scan $N - D$			
Senso r - 1	Senso r - 2	...	Senso r - S	Senso r - 1	Senso r - 2	...	Senso r - S	...	Senso r - 1	Senso r - 2	...	Senso r - S
1	0	...	0	1	2	...	0	...	0	0	...	0

## 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 trackBranchHistory

getHistory Get branch history of maintained tracks

### 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

### Branch Tracks Based on Assignment Results

Apply the results of an assignment algorithm to a track-oriented, multi-hypothesis tracking (MHT) branch history manager. View the resulting track branches (hypotheses).

Create the MHT branch history manager, which is a trackBranchHistory System object™. Set the object to maintain a history of four sensors and two scans.

```
branchHistoryMgr = trackBranchHistory('MaxNumSensors',4,'MaxNumHistoryScans',2)

branchHistoryMgr =
    trackBranchHistory with properties:

        MaxNumSensors: 4
        MaxNumHistoryScans: 2
        MaxNumTracks: 200
        MaxNumTrackBranches: 3
```

Update the branch history. Because the first update has no previous branches, the branch history manager contains only unassigned detections.

```
emptyAssignment = zeros(0,2,'uint32');
emptyUnassignment = zeros(0,1,'uint32');
unassignedDetections = uint32([1;2;3]);
originatingSensor = [1 1 2];
history = branchHistoryMgr(emptyAssignment,emptyUnassignment, ...
    unassignedDetections,originatingSensor);
```

View the current branch history by using the `getHistory` function. Each detection is assigned to a separate track.

```
getHistory(branchHistoryMgr)
```

TrackID	ParentID	BranchID	Scan2				Sensor4	Sensor1	Sensor2
			Sensor1	Sensor2	Sensor3	Sensor4			
1	0	1	1	0	0	0	0	0	0
2	0	2	2	0	0	0	0	0	0
3	0	3	0	3	0	0	0	0	0

Specify multiple branch assignments and multiple unassigned track branches and detections.

- Assign branch 1 to detections 1 and 2.
- Assign branch 2 to detections 1 and 2.
- Consider track branches 1 and 3 unassigned.
- Consider detections 1, 2, and 3 unassigned.

```
assignments = uint32([1 1; 1 2; 2 1; 2 2]);
unassignedTracks = uint32([1;3]);
unassignedDetections = uint32([1;2;3]);
```

Update the branch history manager with the assignments and unassigned tracks and detections.

```
history = branchHistoryMgr(assignments,unassignedTracks, ...
    unassignedDetections,originatingSensor);
```

View the updated branch history.

```
getHistory(branchHistoryMgr)
```

TrackID	ParentID	BranchID	Scan2				Sensor4	Sensor1	Sensor2
			Sensor1	Sensor2	Sensor3	Sensor4			
1	1	1	0	0	0	0	0	1	0
3	3	3	0	0	0	0	0	0	0
4	0	4	1	0	0	0	0	0	0
5	0	5	2	0	0	0	0	0	0
6	0	6	0	3	0	0	0	0	0
1	1	7	1	0	0	0	0	1	0
1	1	8	2	0	0	0	0	1	0

2	2	9	1	0	0	0	2
2	2	10	2	0	0	0	2

Inspect the branch history.

- The most recent scan is Scan 2. The previous scan is Scan 1, which was Scan 2 in the previous assignment update. The history has shifted one scan to the right.
- Branches 1 and 3 are the branches for the unassigned tracks.
- Branch 2 is no longer in the history because it was not considered to be unassigned. Its assignment to detections 1 and 2 created branches 9 and 10.
- Branches 4–6 are branches created for the unassigned detections.
- Branches 7–10 are branches created for the track assignments.

## References

[1] Werthmann, John R. "A Step-by-Step Description of a Computationally Efficient Version of Multiple Hypothesis Tracking." In *Proceedings of SPIE Vol. 1698, Signal and Processing of Small Targets*. 1992, pp. 288–300. doi: 10.1117/12.139379.

## See Also

### Functions

`assignTOMHT` | `clusterTrackBranches`

### System Objects

`trackerTOMHT`

### Introduced in R2018b

# getHistory

Get branch history of maintained tracks

## Syntax

```
history = getHistory(branchHistoryMgr)
history = getHistory(branchHistoryMgr,format)
```

## Description

`history = getHistory(branchHistoryMgr)` returns a table containing the track branch history maintained by the input `trackBranchHistory` System object, `branchHistoryMgr`.

`history = getHistory(branchHistoryMgr,format)` returns the branch history in the specified format: '`table`' or '`matrix`'.

## Examples

### Branch Tracks Based on Assignment Results

Apply the results of an assignment algorithm to a track-oriented, multi-hypothesis tracking (MHT) branch history manager. View the resulting track branches (hypotheses).

Create the MHT branch history manager, which is a `trackBranchHistory` System object™. Set the object to maintain a history of four sensors and two scans.

```
branchHistoryMgr = trackBranchHistory('MaxNumSensors',4,'MaxNumHistoryScans',2)

branchHistoryMgr =
  trackBranchHistory with properties:

    MaxNumSensors: 4
    MaxNumHistoryScans: 2
    MaxNumTracks: 200
    MaxNumTrackBranches: 3
```

Update the branch history. Because the first update has no previous branches, the branch history manager contains only unassigned detections.

```
emptyAssignment = zeros(0,2,'uint32');
emptyUnassignment = zeros(0,1,'uint32');
unassignedDetections = uint32([1;2;3]);
originatingSensor = [1 1 2];
history = branchHistoryMgr(emptyAssignment,emptyUnassignment, ...
    unassignedDetections,originatingSensor);
```

View the current branch history by using the `getHistory` function. Each detection is assigned to a separate track.

```
getHistory(branchHistoryMgr)
```

TrackID	ParentID	BranchID	Scan2				Sensor4	Sensor1	Sensor2
			Sensor1	Sensor2	Sensor3	Sensor4			
1	0	1	1	0	0	0	0	0	0
2	0	2	2	0	0	0	0	0	0
3	0	3	0	3	0	0	0	0	0

Specify multiple branch assignments and multiple unassigned track branches and detections.

- Assign branch 1 to detections 1 and 2.
- Assign branch 2 to detections 1 and 2.
- Consider track branches 1 and 3 unassigned.
- Consider detections 1, 2, and 3 unassigned.

```
assignments = uint32([1 1; 1 2; 2 1; 2 2]);
unassignedTracks = uint32([1;3]);
unassignedDetections = uint32([1;2;3]);
```

Update the branch history manager with the assignments and unassigned tracks and detections.

```
history = branchHistoryMgr(assignments,unassignedTracks, ...
    unassignedDetections,originatingSensor);
```

View the updated branch history.

```
getHistory(branchHistoryMgr)
```

TrackID	ParentID	BranchID	Scan2				Sensor4	Sensor1	Sensor2
			Sensor1	Sensor2	Sensor3	Sensor4			
1	1	1	0	0	0	0	0	1	0
3	3	3	0	0	0	0	0	0	0
4	0	4	1	0	0	0	0	0	0
5	0	5	2	0	0	0	0	0	0
6	0	6	0	3	0	0	0	0	0
1	1	7	1	0	0	0	0	1	0
1	1	8	2	0	0	0	0	1	0
2	2	9	1	0	0	0	0	2	0
2	2	10	2	0	0	0	0	2	0

Inspect the branch history.

- The most recent scan is Scan 2. The previous scan is Scan 1, which was Scan 2 in the previous assignment update. The history has shifted one scan to the right.
- Branches 1 and 3 are the branches for the unassigned tracks.
- Branch 2 is no longer in the history because it was not considered to be unassigned. Its assignment to detections 1 and 2 created branches 9 and 10.
- Branches 4–6 are branches created for the unassigned detections.
- Branches 7–10 are branches created for the track assignments.

## Input Arguments

### **branchHistoryMgr — Input branch history manager**

trackBranchHistory System object

Input branch history manager, specified as a trackBranchHistory System object.

### **format — Format of output branch history**

'table' (default) | 'matrix'

Format of the output branch history, specified as one of the following:

- 'table' (default) — Return branch history in a table.
- 'matrix' — Return branch history in a matrix. This output is equivalent to the output returned when calling the trackBranchHistory System object.

## Output Arguments

### **history — Branch history**

table of integers | matrix of integers

Branch history, returned as a table of integers or as a matrix of integers.

Each row of **history** represents a unique track branch. **history** has  $3+(D \times S)$  columns, where  $D$  is the number of maintained scans (the history depth) and  $S$  is the maximum number of maintained sensors. The first three columns represent the following information about each track branch:

- **TrackID** — ID of the track that is associated with the branch. Track branches that are assumed to have originated from the same target have the same track ID. If a branch originates from an unassigned detection, that branch gets a new track ID.
- **ParentID** — ID of the parent branch, that is, the branch from which the current branch originated. Branches that were created from the same parent have the same ParentID. A ParentID of 0 indicates a new track. These tracks are created from hypotheses corresponding to unassigned detections.
- **BranchID** — Unique ID of track branch. Every branch created from an unassigned detection or assignment gets a new branch ID.

The remaining  $D \times S$  columns contain the IDs of the detections assigned to each branch. A branch can be assigned to at most one detection per scan and per sensor. The table shows the organization of these columns with sample detections.  $N$  is the number of scans. A value of 0 means that the sensor at that scan does not have a detection assigned to it.

Scan $N$				Scan $N - 1$				...	Scan $N - D$			
Senso r - 1	Senso r - 2	...	Senso r - S	Senso r - 1	Senso r - 2	...	Senso r - S	...	Senso r - 1	Senso r - 2	...	Senso r - S
1	0	...	0	1	2	...	0	...	0	0	...	0

## See Also

trackBranchHistory

**Introduced in R2018b**

# staticDetectionFuser

Static fusion of synchronous sensor detections

## Description

`staticDetectionFuser` System object creates a static detection fuser object to fuse angle-only sensor detections.

To obtain the fuser:

- 1 Create the `staticDetectionFuser` 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

```
fuser = staticDetectionFuser()
fuser = staticDetectionFuser(Name,Value)
```

### Description

`fuser = staticDetectionFuser()` creates a default three-sensor static detection fuser object to fuse angle-only sensor detections.

`fuser = staticDetectionFuser(Name,Value)` sets properties using one or more name-value pairs. For example, `fuser = staticDetectionFuser('FalseAlarmRate',1e-6,'MaxNumSensors',12)` creates a fuser that has a maximum of 12 sensors and a false alarm rate of `1e-6`. 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](#).

### FuserSensorIndex — Sensor index of composite detections

1 (default) | positive integer

Sensor index of the composite detections reported by the fuser, specified as a positive integer. This index becomes the `SensorIndex` of `objectDetection` objects returned by the fuser.

Example: 5

Data Types: double

### **MeasurementFusionFcn — Function for fusing multiple sensor detections**

'triangulateLOS' (default) | char | string | function handle

Function for fusing multiple sensor detections, specified as a character vector, string, or function handle. The function fuses multiple detections into one and returns the fused measurement and measurement noise. Any fusing function combines at most one detection from each sensor. The syntax of the measurement fuser function is:

```
[fusedMeasurement, fusedMeasurementNoise] = MeasurementFusionFcn(detections)
```

where the input and output functions arguments are

- `detections` – cell array of `objectDetection` measurements.
- `fusedMeasurement` – an  $N$ -by-1 vector of fused measurements.
- `fusedMeasurementNoise` – an  $N$ -by- $N$  matrix of fused measurements noise.

The value of  $N$  depends on the `MeasurementFormat` property.

MeasurementFormat Property	N
'Position'	1, 2, and 3
'Velocity'	1, 2, and 3
'PositionAndVelocity'	2, 4, and 6
'Custom'	Any

Data Types: char | string | function\_handle

### **MeasurementFormat — Format of the fused measurement**

'Position' (default) | 'Velocity' | 'PositionAndVelocity' | 'Custom'

Format of the fused measurement, specified as 'Position', 'Velocity', 'PositionAndVelocity', or 'Custom'. The formats are

- 'Position' – the fused measurement is the position of the target in the global coordinate frame.
- 'Velocity' – the fused measurement is the velocity of the target in the global coordinate frame.
- 'PositionAndVelocity' – the fused measurement is the position and velocity of the target in the global coordinate frame defined according to the format  $[x;vx;y;vy;z;vz]$ .
- 'Custom' – custom fused measurement. To enable this format, specify a function using the `MeasurementFcn`.

Example: 'PositionAndVelocity'

### **MeasurementFcn — Custom measurement function**

char | string | function handle

Custom measurement function, specified as a character vector, string, or function handle. Specify the function that transforms fused measurements into sensor measurements. The function must have the following signature:

```
sensorMeas = MeasurementFcn(fusedMeas,measParameters)
```

**Dependencies**

To enable this property, set the **MeasurementFormat** property to 'Custom'.

Data Types: char | string | function\_handle

**MaxNumSensors — Maximum number of sensors in surveillance region**

3 (default) | positive integer greater than one

Maximum number of sensors in surveillance region, specified as a positive integer greater than one.

Data Types: double

**Volume — Volume of sensor detection bin**

1e-2 (default) | positive scalar |  $N$ -length vector of positive scalars

Volume of sensors detection bins, specified as a positive scalar or  $N$ -length vector of positive scalars.  $N$  is the number of sensors. If specified as a scalar, each sensor is assigned the same volume. If a sensor produces an angle-only measurement, for example, azimuth and elevation, the volume is defined as the solid angle subtended by one bin.

Data Types: double

**DetectionProbability — Probabilities of a target detection**

0.9 (default) | positive scalar |  $N$ -length vector of positive scalars

Probability of detection of a target by each sensor, specified as a scalar or  $N$ -length vector of positive scalars in the range (0,1).  $N$  is the number of sensors. If specified as a scalar, each sensor is assigned the same detection probability. The probability of detection is used in calculating the cost of fusing a "one" (target was detected) or "zero" (target was not detected) detections from each sensor.

Example: 0.99

Data Types: double

**FalseAlarmRate — Rate of false positives generated by sensors**

1e-6 (default) | positive scalar |  $N$ -length vector of positive scalars

Rate at which false positives are reported by sensor in each bin, specified as a scalar or  $N$ -length vector of positive scalars.  $N$  is the number of sensors. If specified as a scalar, each sensor is assigned the same false alarm rate. The false alarm rate is used to calculate the likelihood of clutter in the detections reported by each sensor.

Example: 1e-5

Data Types: double

**UseParallel — Option to use parallel computing resources**

false (default) | true

Option to use parallel computing resources, specified as `false` or `true`. The `staticDetectionFuser` calculates the cost of fusing detections from each sensor as an n-D assignment problem. The fuser spends most of the time in computing the cost matrix for the assignment problem. If Parallel Computing Toolbox is installed, this option lets the fuser use the parallel pool of workers to compute the cost matrix.

Data Types: logical

**TimeTolerance — Absolute tolerance between timestamps of detections**

1e-6 (default) | nonnegative scalar

Absolute tolerance between timestamps of detections, specified as a nonnegative scalar. The `staticDetectionFuser` assumes that sensors are synchronous. This property defines the allowed tolerance value between detection time-stamps to still be considered synchronous.

Example: 1e-3

Data Types: double

**Usage****Syntax**

```
compositeDets = fuser(dets)
[compositeDets,analysisInfo] = fuser(dets)
```

**Description**

`compositeDets = fuser(dets)` returns the fused detections, `compositeDets`, of input detections, `dets`.

`[compositeDets,analysisInfo] = fuser(dets)` also returns analysis information, `analysisInfo`.

**Input Arguments****dets — Pre-fused detections**cell array of `objectDetection` objects

Pre-fused detections, specified as a cell array of `objectDetection` objects.

**Output Arguments****compositeDets — Fused detections**cell array of `objectDetection` objects

Pre-fused detections, returned as a cell array of `objectDetection` objects.

**analysisInfo — Analysis information**

structure

Analysis information, returned as a structure. The fields of the structure are:

- **CostMatrix** -  $N$ -dimensional cost matrix providing the cost of association of detections, where  $N$  is the number of sensors. The cost is the negative log-likelihood of the association and can be interpreted as the negative score of the track that will be generated by the fused measurement.
- **Assignments** - A  $P$ -by- $N$  list of assignments, where  $P$  is the number of composite detections.
- **FalseAlarms** - A  $Q$ -by-1 list of indices of detections declared as false alarms by association.

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

release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object
isLocked	Determine if System object is in use
clone	Create duplicate System object

## Examples

### Fuse Detections from ESM Sensors

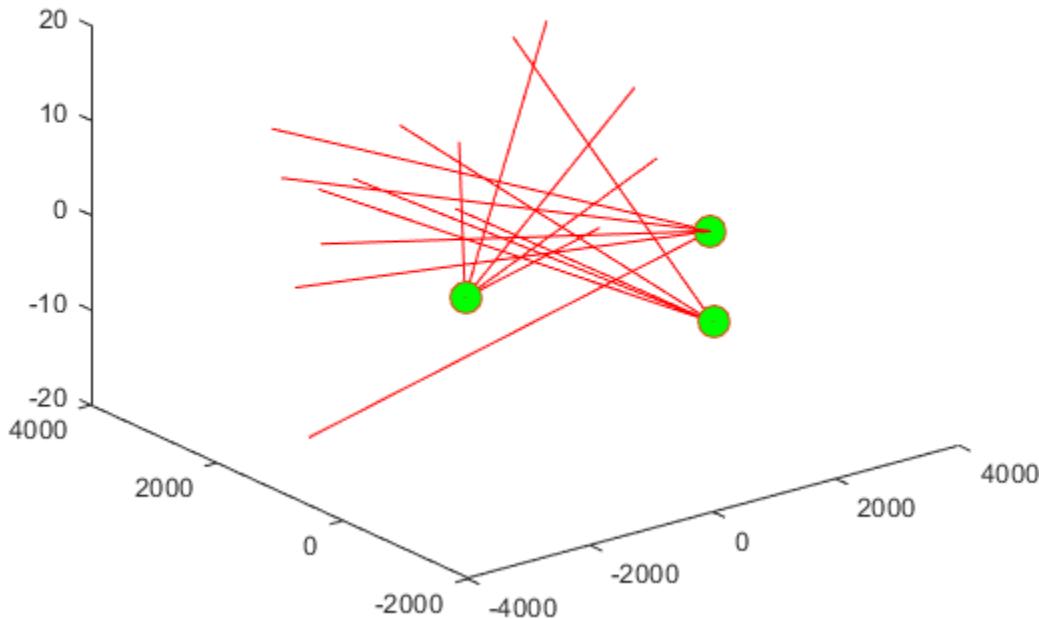
Fuse angle-only detections from three ESM sensors.

Load stored detections from the sensors.

```
load('angleOnlyDetectionFusion.mat','detections');
```

Visualize angle-only detections for plotting the direction vector.

```
rPlot = 5000;
plotData = zeros(3,numel(detections)*3);
for i = 1:numel(detections)
    az = detections{i}.Measurement(1);
    el = detections{i}.Measurement(2);
    [xt,yt,zt] = sph2cart(deg2rad(az),deg2rad(el),rPlot);
    % The sensor is co-located at platform center, therefore use
    % the position from the second measurement parameter
    originPos = detections{i}.MeasurementParameters(2).OriginPosition;
    positionData(:,i) = originPos(:,1);
    plotData(:,3*i-2) = [xt;yt;zt] + originPos(:,1);
    plotData(:,3*i-1) = originPos(:,1);
    plotData(:,3*i) = [NaN;NaN;NaN];
end
plot3(plotData(1,:),plotData(2,:),plotData(3,:),'r-')
hold on
plot3(positionData(1,:),positionData(2,:),positionData(3,:),'o','MarkerSize',12,'MarkerFaceColor','r')
```



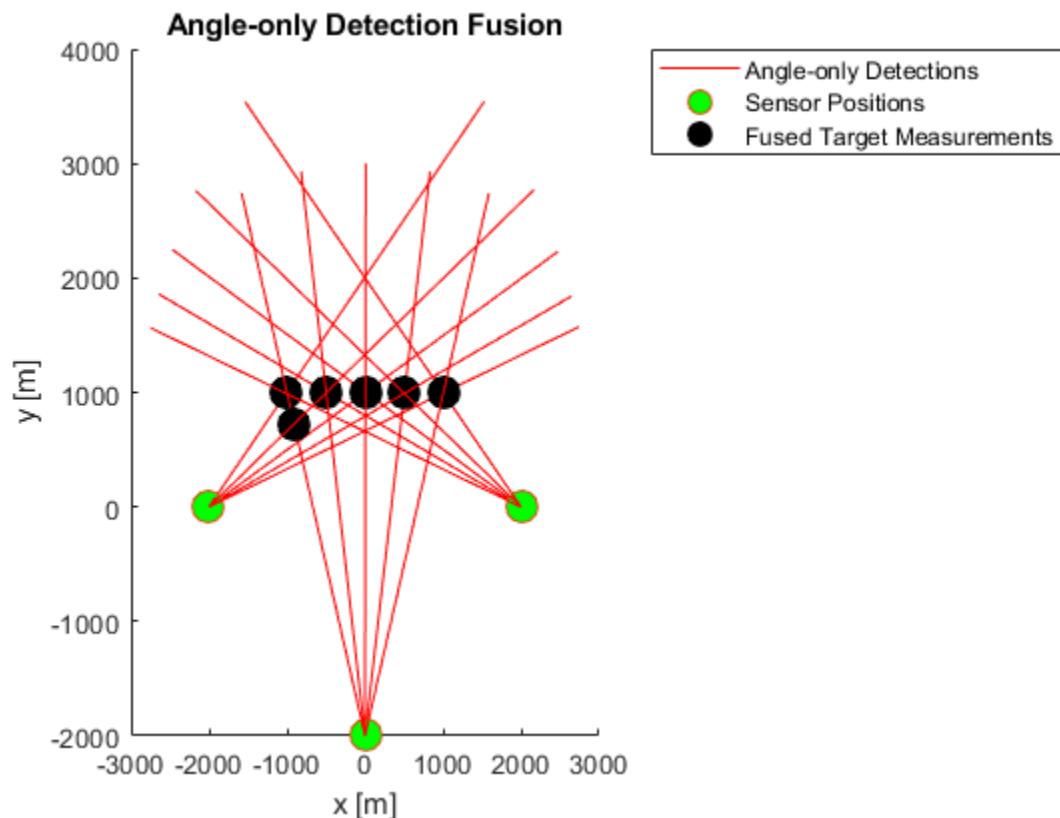
Create a `staticDetectionFuser` to fuse angle-only detections using the measurement fusion function `triangulateLOS`.

```
fuser = staticDetectionFuser('MeasurementFusionFcn', 'triangulateLOS', 'MaxNumSensors', 3)
fuser =
  staticDetectionFuser with properties:
    FusedSensorIndex: 1
    MeasurementFusionFcn: 'triangulateLOS'
    MeasurementFormat: 'Position'
    MaxNumSensors: 3
    Volume: [3x1 double]
    DetectionProbability: [3x1 double]
    FalseAlarmRate: [3x1 double]
    TimeTolerance: 1.0000e-06
    UseParallel: false
```

Create the fused detections and obtain the analysis information.

```
[fusedDetections, analysisInfo] = fuser(detections);
fusedPositions = zeros(3,numel(fusedDetections));
for i = 1:numel(fusedDetections)
    fusedPositions(:,i) = fusedDetections{i}.Measurement;
end
```

```
plot3(fusedPositions(1,:),fusedPositions(2,:),fusedPositions(3,:),'ko', ...
    'MarkerSize',12, 'MarkerFaceColor','k')
legend('Angle-only Detections','Sensor Positions','Fused Target Measurements')
title('Angle-only Detection Fusion')
xlabel('x [m]')
ylabel('y [m]')
view(2)
```



Use the analysisInfo output to check the assignments.

```
analysisInfo.Assignments
```

```
ans = 6x3 uint32 matrix
```

0	10	14
1	6	11
2	7	12
3	8	13
4	9	0
5	0	15

## Algorithms

### Detection Fusion Workflow

The static detection fuser:

- Calculates the cost of fusing or matching detections from each sensor to one another.
- Solves a 2-D or S-D assignment problem, where  $S$  is the number of sensors, to associate or match detections from one sensor to others.
- Fuses the measurement and measurement covariance of the associated detection n-tuples to generate a list of composite or fused detections.
- Declares unassigned detections from each sensor as false alarms.

The `staticDetectionFuser` assumes that all sensors are synchronous and generate detections simultaneously. The `staticDetectionFuser` also assumes that the sensors share a common surveillance region. Associating  $n$  detections from  $m$  sensors indicates  $m - n$  missed detections or false alarms.

## References

[1] Bar-Shalom, Yaakov, Peter K. Willett, and Xin Tian. *Tracking and data fusion*. Storrs, CT, USA:: YBS publishing, 2011.

## 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

`triangulateLOS`

### Objects

`objectDetection`

### System Objects

`irSensor` | `monostaticRadarSensor` | `radarSensor` | `sonarSensor`

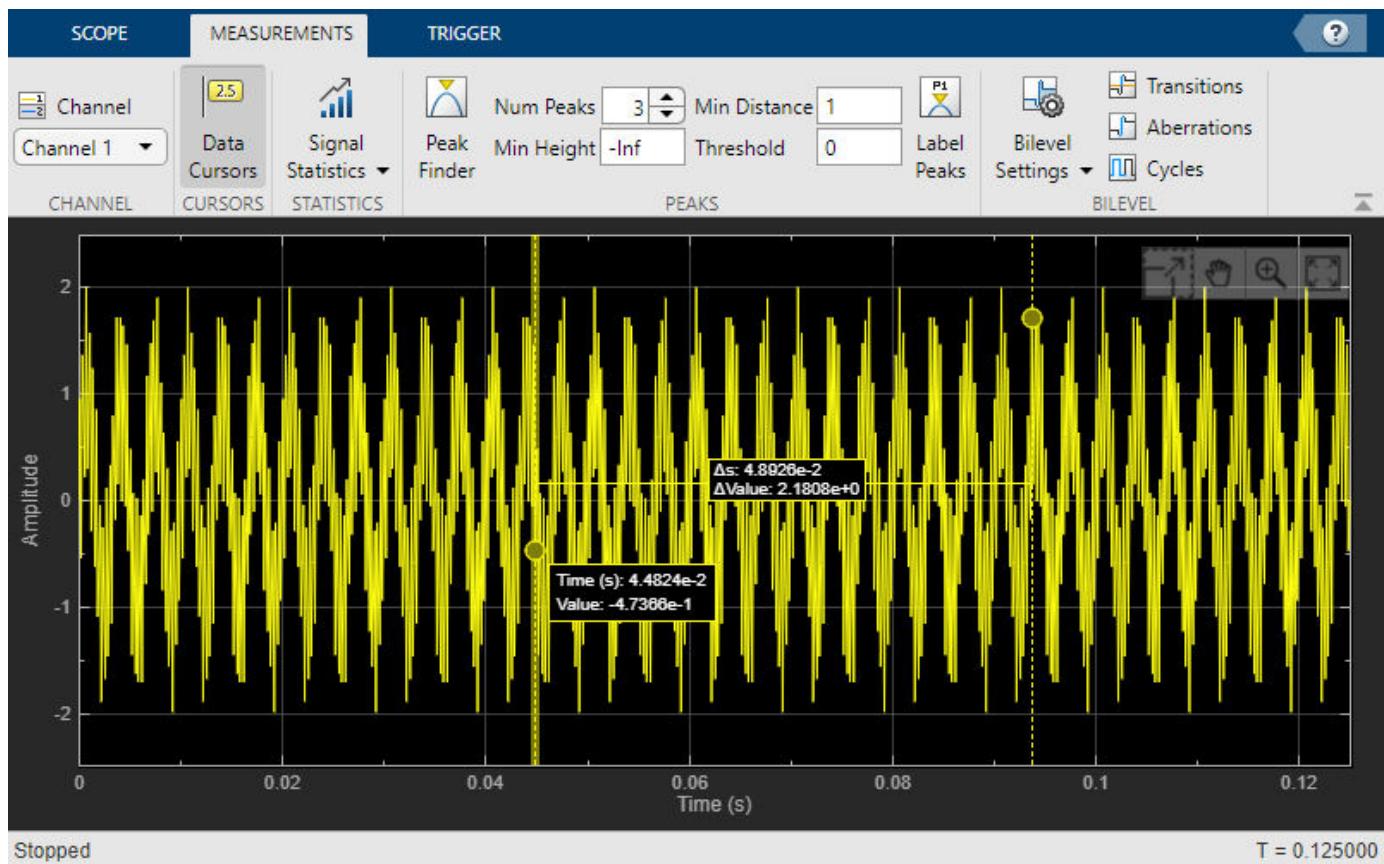
## Introduced in R2018b

## timescope

Display time-domain signals

### Description

The `timescope` object displays signals in the time domain.



Scope features:

- “Data Cursors” — Measure signal values using vertical and horizontal cursors.
- “Signal Statistics” — Display the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal.
- “Peak Finder” — Find maxima, showing the x-axis values at which they occur.
- “Bilevel Measurements” — Measure transitions, overshoots, undershoots, and cycles.
- “Triggers” — Set triggers to sync repeating signals and pause the display when events occur.

Use “Object Functions” on page 3-533 to show, hide, and determine visibility of the scope window.

# Creation

## Syntax

```
scope = timescope
scope = timescope(Name,Value)
```

## Description

`scope = timescope` returns a `timescope` object, `scope`. This object displays real- and complex-valued floating and fixed-point signals in the time domain.

`scope = timescope(Name,Value)` returns a `timescope` object with properties set to the specified value. Specify properties and their values in quotes, separated by commas. You can specify name-value pair arguments in any order.

## Properties

Most properties can be changed from the `timescope` UI.

### Frequently Used

#### **SampleRate — Sample rate of inputs**

1 (default) | finite numeric scalar | vector

Sampling rate of the input signal, in hertz, specified as a finite numeric scalar or vector of scalars.

The inverse of the sample rate determines the x-axis (time axis) spacing between points in the displayed signal. When the value of `NumInputPorts` is greater than 1 and the sample rate is scalar, the object uses the same sample rate for all inputs. To specify different sample rates for each input, use a vector.

You can only set this property when creating the object or after calling `release`.

### UI Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Sample Rate**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

#### **TimeSpanSource — Source of time span**

`'auto'` (default) | `'property'`

Source of the time span for frame-based input signals, specified as one of the following:

- `'property'` - The object derives the x-axis limits from the `TimeDisplayOffset` and `TimeSpan` properties.
- `'auto'` - The x-axis limits are derived from the `TimeDisplayOffset` property, `SampleRate` property, and the number of rows in each input signal (`FrameSize` in the equations below). The limits are calculated as:
  - Minimum time-axis limit = `TimeDisplayOffset`
  - Maximum time-axis limit = `TimeDisplayOffset + max(1/SampleRate.*FrameSize)`

#### UI Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Time Span**.

Data Types: char | string

#### TimeSpan — Time span

10 (default) | positive scalar

Time span, in seconds, specified as a positive, numeric scalar value. The time-axis limits are calculated as:

- Minimum time-axis limit = `TimeDisplayOffset`
- Maximum time-axis limit = `TimeDisplayOffset + TimeSpan`

#### Dependencies

To enable this property, set `TimeSpanSource` to 'property'.

#### UI Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, edit **Time Span**.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

#### TimeSpanOverrunAction — Data overrun behavior

'scroll' (default) | 'wrap'

Specify how the scope displays new data beyond the visible time span as either:

- 'scroll' — In this mode, the scope scrolls old data to the left to make room for new data on the right of the scope display. This mode is beneficial for debugging and monitoring time-varying signals.
- 'wrap' — In this mode, the scope adds data to the left of the plot after overrunning the right of the plot.

#### UI Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Overrun Action**.

Data Types: char | string

#### PlotType — Type of plot

'line' (default) | 'stairs'

Type of plot, specified as either:

- 'line' — Line graph, similar to the `line` or `plot` function.
- 'stairs' — Stair-step graph, similar to the `stairs` function. Stair-step graphs are useful for drawing time history graphs of digitally sampled data.

#### UI Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Plot Type**.

Data Types: char | string

**AxesScaling — Axes scaling mode**

'onceatstop' (default) | 'auto' | 'manual' | 'updates'

When this property is set to:

- 'onceatstop' -- The limits are updated once at the end of the simulation (when `release` is called).
- 'auto' -- The scope attempts to always keep the data in the display while minimizing the number of updates to the axes limits.
- 'manual' -- The scope takes no action unless specified by the user.
- 'updates' -- The scope scales the axes once and only once after 100 updates to the visualization.

You can set this property only when creating the object.

Data Types: char | string

**Advanced****LayoutDimensions — Display layout grid dimensions**

[1,1] (default) | [numberOfRows, numberOfColumns]

Specify the layout grid dimensions as a two-element vector: [numberOfRows,numberOfColumns]. The grid can have a maximum of 4 rows and 4 columns.

Example: `scope.LayoutDimensions = [2,4]`

**UI Use**

On the **Scope** tab, click **Display Grid** (grid icon) and select a specific number of rows and columns from the grid.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**TimeUnits — Units of x-axis**

'seconds' (default) | 'none'

Specify the units used to describe the x-axis (time axis). You can select one of the following options:

- 'seconds' — In this mode, the scope always displays the units on the x-axis as seconds. The scope shows the word **Time(s)** on the x-axis.
- 'none' — In this mode, the scope does not display any units on the x-axis. The scope only shows the word **Time** on the x-axis.

**UI Use**

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Time Units**.

Data Types: char | string

**TimeDisplayOffset — Offset x-axis limits**

0 (default) | scalar | vector

Specify, in seconds, how far to move the data on the x-axis. The signal value does not change, only the limits displayed on the x-axis change.

If you specify this property as a scalar, then that value is the time display offset for all channels. If you specify this property as a vector, each input channel can be a different time display offset

#### UI Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Time Offset**.

#### TimeAxisLabels — Time-axis labels

'all' (default) | 'bottom' | 'none'

Time-axis labels, specified as:

- 'all' — Time-axis labels appear in all displays.
- 'bottom' — Time-axis labels appear in the bottom display of each column.
- 'none' — No labels appear in any display.

#### UI Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Time Labels**.

Data Types: char | string

#### MaximizeAxes — Maximize axes control

'auto' (default) | 'on' | 'off'

Specify whether to display the scope in the maximized-axes mode. In this mode, the axes are expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, the tick-marks and their values appear on top of the plotted data. You can select one of the following options:

- 'auto' — The axes appear maximized in all displays only if the **Title** and **YLabel** properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.
- 'on' — The axes appear maximized in all displays. Any values entered into the **Title** and **YLabel** properties are hidden.
- 'off' — None of the axes appear maximized.

#### UI Use



On the scope window, click on to maximize axes, hiding all labels and insetting the axes values.

Data Types: char | string

#### BufferLength — Buffer length

50000 (default) | positive integer

Specify the length of the buffer used for each input signal as a positive integer.

You can set this property only when creating the object.

#### UI Use

On the **Scope** tab, click **Settings**. Under **Data and Axes**, set **Buffer Length**.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Visualization

### Name — Window name

'Time Scope' (default) | character vector | string scalar

Specify the name of the scope as a character vector or string scalar. This name appears as the title of the scope's figure window. To specify a title of a scope plot, use the **Title** property.

Data Types: char | string

### Position — Window position

screen center (default) | [left bottom width height]

Scope window position in pixels, specified by the size and location of the scope window as a four-element vector of the form [left bottom width height]. You can place the scope window in a specific position on your screen by modifying the values of this property.

By default, the window appears in the center of your screen with a width of 800 pixels and height of 500 pixels. The exact values of the position depend on your screen resolution.

### ChannelNames — Channel names

{''} (default) | cell array of character vectors

Specify the input channel names as a cell array of character vectors. The channel names appear in the legend, and on the **Measurements** tab under **Select Channel**. If you do not specify names, the channels are labeled as Channel 1, Channel 2, etc.

### Dependency

To enable this property, set **ShowLegend** to true.

Data Types: char

### ActiveDisplay — Active display for setting properties

1 (default) | integer

Active display used to set properties, specified by the integer display number. The number of a display corresponds to the display's row-wise placement index. Setting this property controls which display is used for the following properties: **YLimits**, **YLabel**, **ShowLegend**, **ShowGrid**, **Title**, and **PlotAsMagnitudePhase**.

### UI Use

On the **Scope** tab, click **Settings**. Under **Display and Labels**, set **Active Display**.

### Title — Display title

'' (default) | character vector | string scalar

Specify the display title as a character vector or a string scalar.

### Dependency

When you set this property, **ActiveDisplay** controls the display that is updated.

### UI Use

On the **Scope** tab, click **Settings**. Under **Display and Labels**, set **Title**.

Data Types: char | string

**YLabel — y-axis label**

'Amplitude' (default) | character vector | string scalar

Specify the text for the scope to display to the left of the y-axis.

**Dependencies**

This property applies only when `PlotAsMagnitudePhase` is `false`. When `PlotAsMagnitudePhase` is `true`, the two y-axis labels are read-only values "Magnitude" and "Phase", for the magnitude plot and the phase plot, respectively.

When you set this property, `ActiveDisplay` controls the display that is updated.

**UI Use**

On the **Scope** tab, click **Settings**. Under **Display and Labels**, set **YLabel**.

Data Types: char | string

**YLimits — y-axis limits**

[-10,10] (default) | [ymin, ymax]

Specify the y-axis limits as a two-element numeric vector, [ymin, ymax].

- If `PlotAsMagnitudePhase` is `false`, the default is [-10,10].
- If `PlotAsMagnitudePhase` is `true`, the default is [0,10]. This property specifies the y-axis limits of only the magnitude plot. The y-axis limits of the phase plot are always [-180,180]

**Dependency**

When you set this property, `ActiveDisplay` controls the display that is updated.

**UI Use**

On the **Scope** tab, click **Settings**. Under **Display and Labels**, set **Y-Axis Limits**.

**ShowLegend — Show legend**

false (default) | true

To show a legend with the input names, set this property to `true`.

From the legend, you can control which signals are visible. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again.

**UI Use**

On the **Scope** tab, click **Settings**. Under **Display and Labels**, select **Show Legend**.

Data Types: logical

**ShowGrid — Grid visibility**

true (default) | false

Set this property to `true` to show grid lines on the plot.

## UI Use

On the **Scope** tab, click **Settings**. Under **Display and Labels**, select **Show Grid**.

### **PlotAsMagnitudePhase — Plot signal as magnitude and phase**

false (default) | true

Plot signal as magnitude and phased, specified as either:

- **true** – The scope plots the magnitude and phase of the input signal on two separate axes within the same active display.
- **false** – The scope plots the real and imaginary parts of the input signal on two separate axes within the same active display.

This property is useful for complex-valued input signals. Turning on this property affects the phase for real-valued input signals. When the amplitude of the input signal is nonnegative, the phase is 0 degrees. When the amplitude of the input signal is negative, the phase is 180 degrees.

## UI Use

On the **Scope** tab, click **Settings**. Under **Display and Labels**, select **Magnitude Phase Plot**.

## Object Functions

To use an object function, specify the object as the first input argument.

hide	Hide scope window
show	Display scope window
isVisible	Determine visibility of scope
generateScript	Generate MATLAB script to create scope with current settings
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

### View Sine Wave on Time Scope

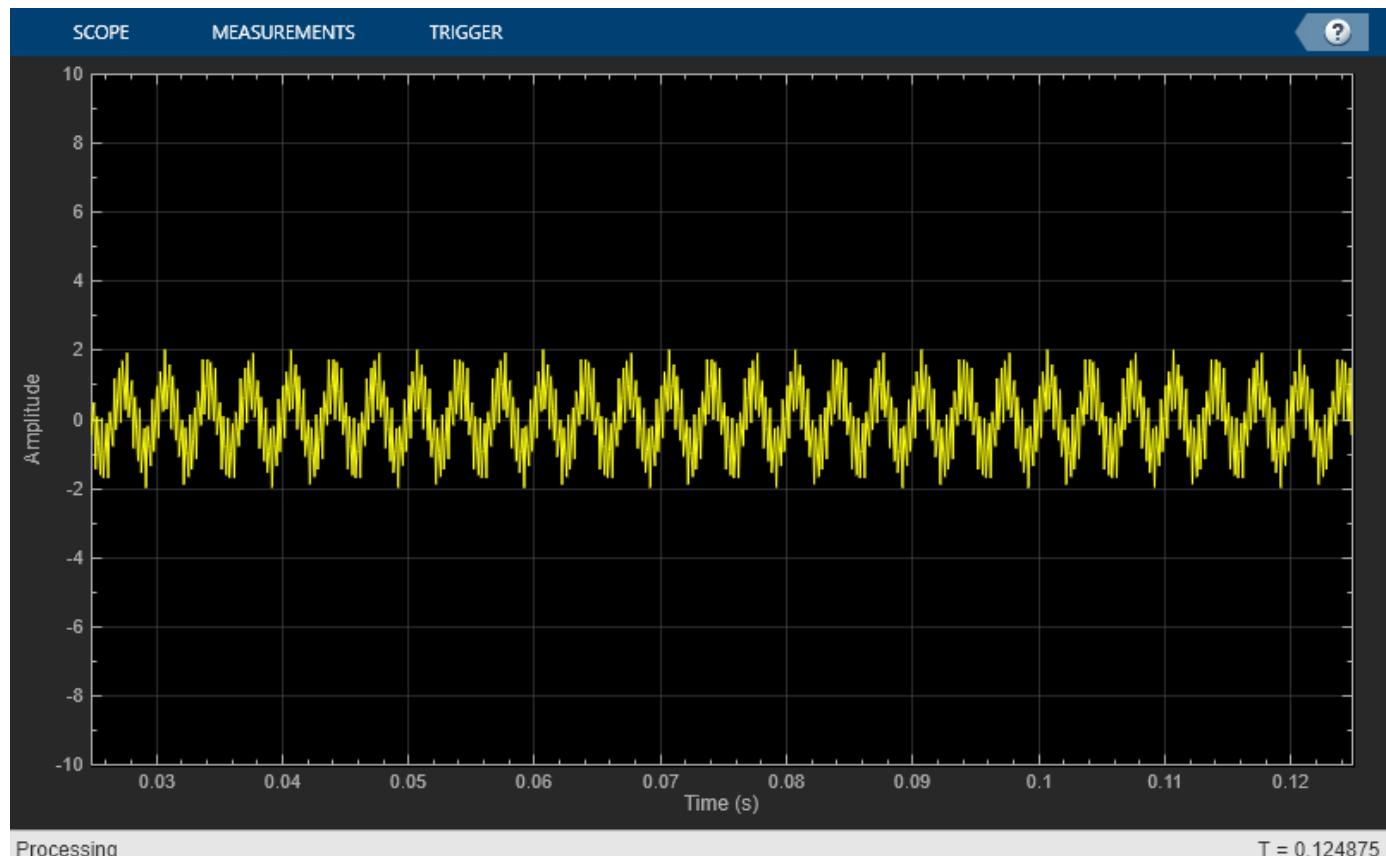
Create a time-domain sinusoidal signal. Display the signal by calling the time scope object.

Create a sinusoidal signal with two tones, one at 0.3 kHz and the other at 3 kHz.

```
t = (0:1000)'/8e3;
xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t);
```

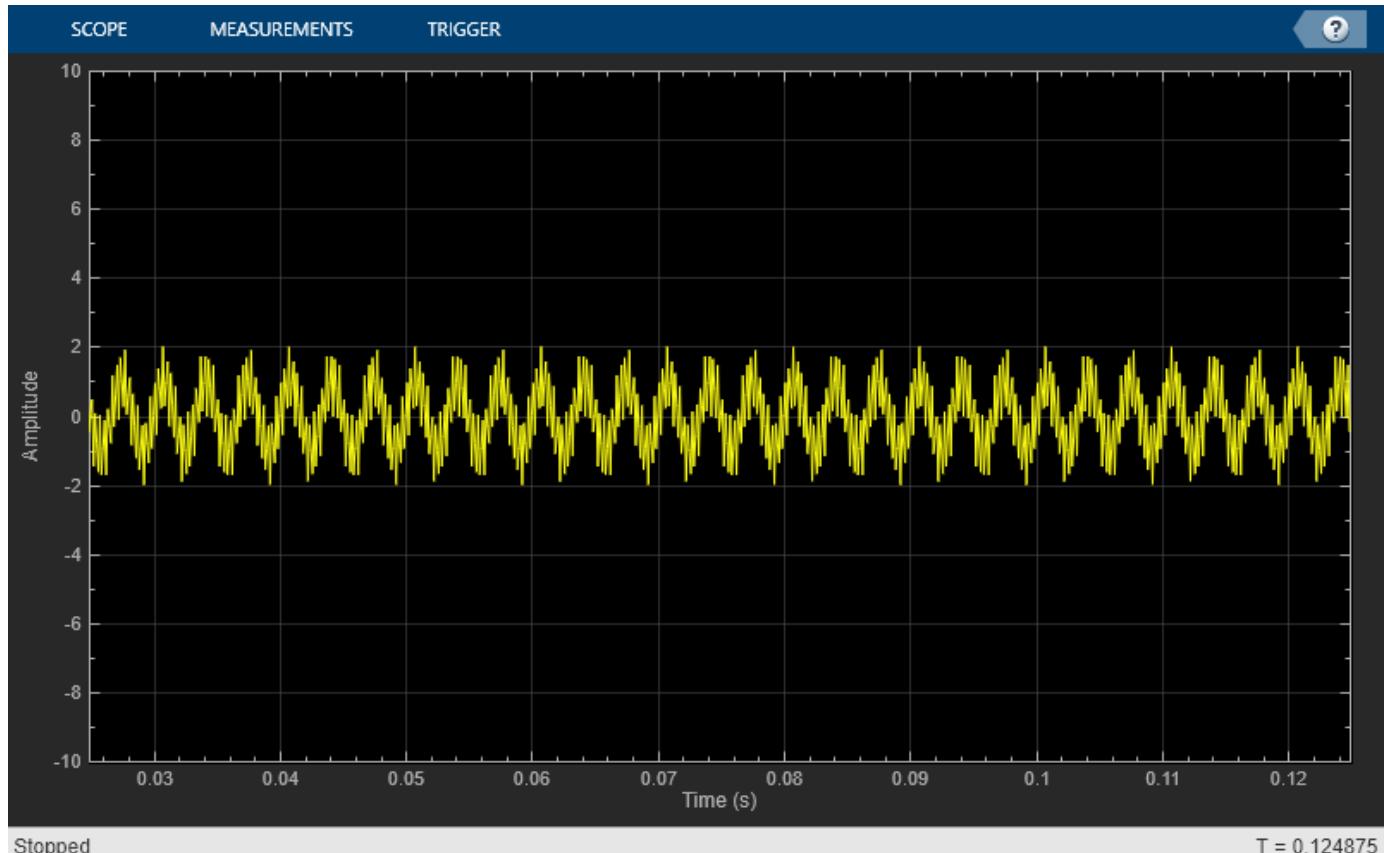
Create a **timescope** object and view the sinusoidal signal by calling the time scope object **scope**.

```
scope = timescope('SampleRate', 8e3, ...
    'TimeSpanSource', 'property', ...
    'TimeSpan', 0.1);
scope(xin)
```



Run `release` to allow changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope);
```

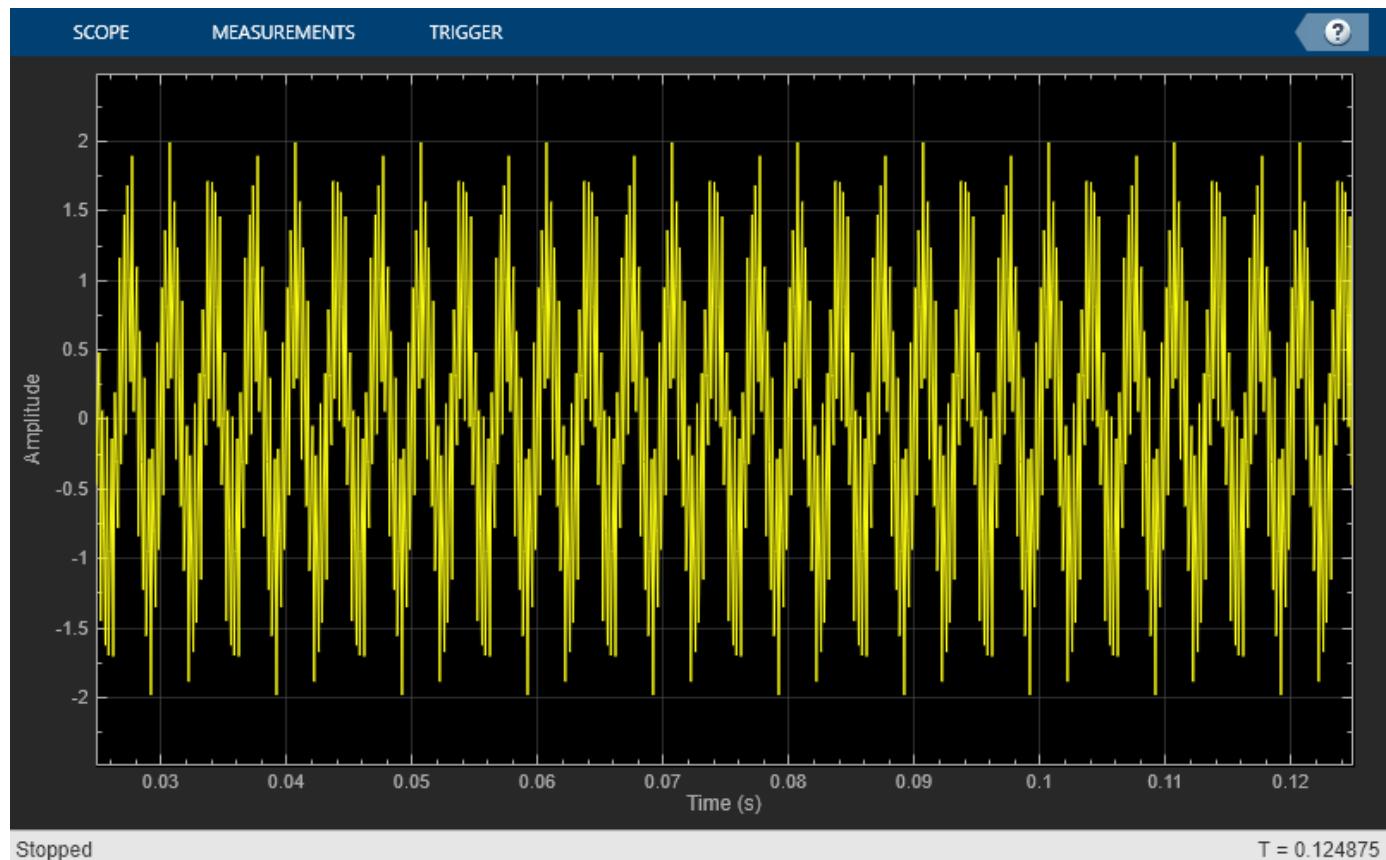


Hide the scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the scope window.

```
if(~isVisible(scope))
    show(scope)
end
```



## See Also

### Topics

"Configure Time Scope MATLAB Object"

**Introduced in R2020a**

# generateScript

Generate MATLAB script to create scope with current settings

## Syntax

```
generateScript(scope)
```

## Description

`generateScript(scope)` generates a MATLAB script that can re-create a `timescope` object with the current settings in the scope.

## Examples

### Generate Script from `timescope`

Generate MATLAB script after making changes to the `timescope` object in the scope window.

---

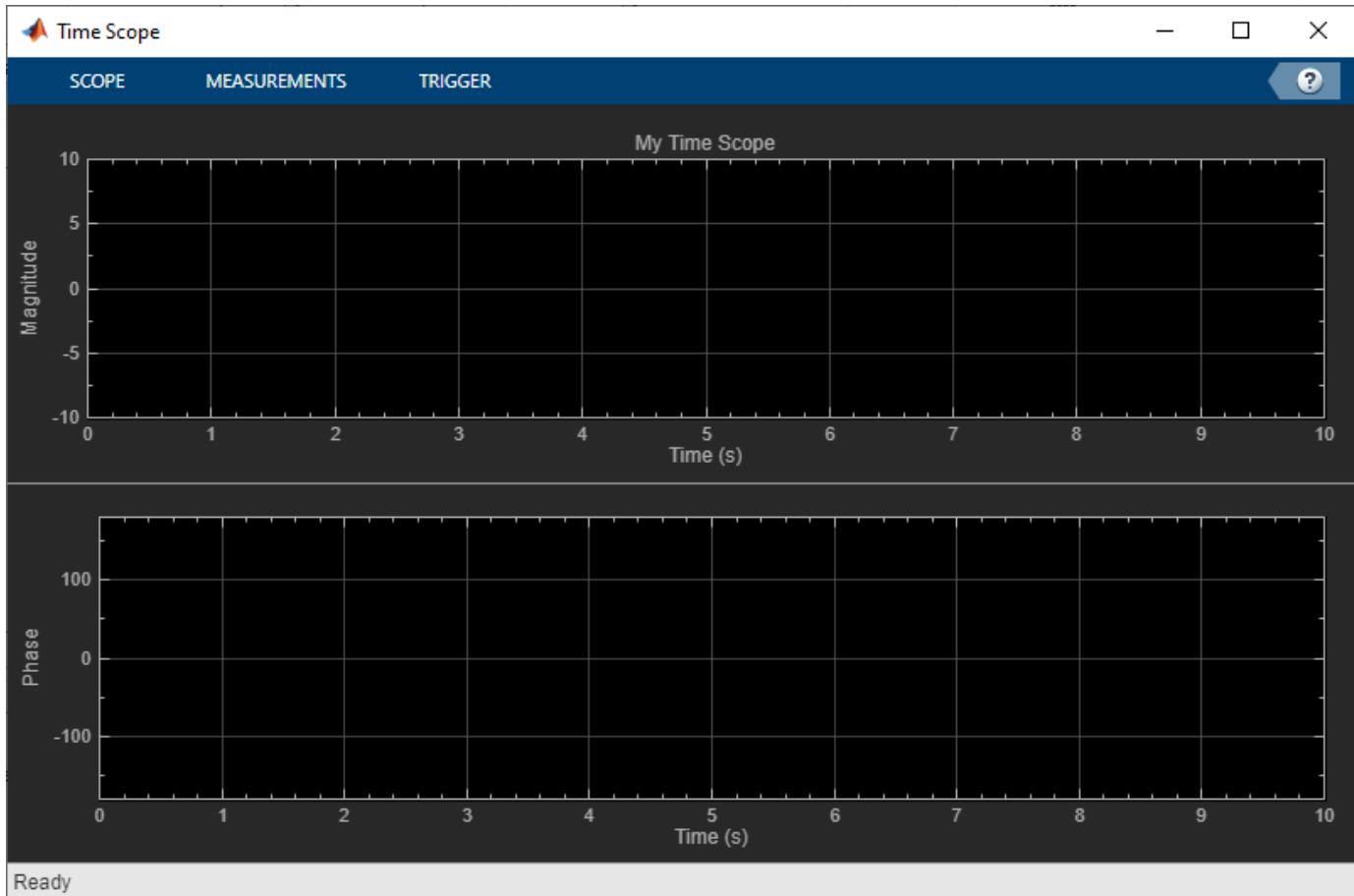
**Note** The script only generates commands for settings that are available from the command line, applicable to the current visualization, and changed from the default value.

---

- 1 Create a `timescope` object.

```
scope = timescope;  
show(scope)
```

- 2 Set options in the Time Scope. For this example, on the **Scope** tab, click **Settings**. Under **Display and Labels**, select **Show Legend** and **Magnitude Phase Plot**. Set the **Title** as well.



- 3 Generate a script to recreate the timescope with the same modified settings. Either select **Generate Script** from the **Scope** tab, or enter:

```
generateScript(scope);
```

A new editor window opens with code to regenerate the same scope.

```
% Creation Code for 'timescope'.
% Generated by Time Scope on 8-Nov-2019 13:51:54 -0500.

timeScope = timescope('Position',[2286 355 800 500], ...
    'Title','My Time Scope',...
    'ShowLegend',true,...  
    'PlotAsMagnitudePhase',true);
```

## Input Arguments

### scope — object

timescope object

Object whose settings you want to recreate with a script.

## See Also

### Functions

hide | isVisible | show

**Objects**  
timescope

**Introduced in R2020a**

## hide

Hide scope window

### Syntax

```
hide(scope)
```

### Description

`hide(scope)` hides the scope window.

### Examples

#### View Sine Wave on Time Scope

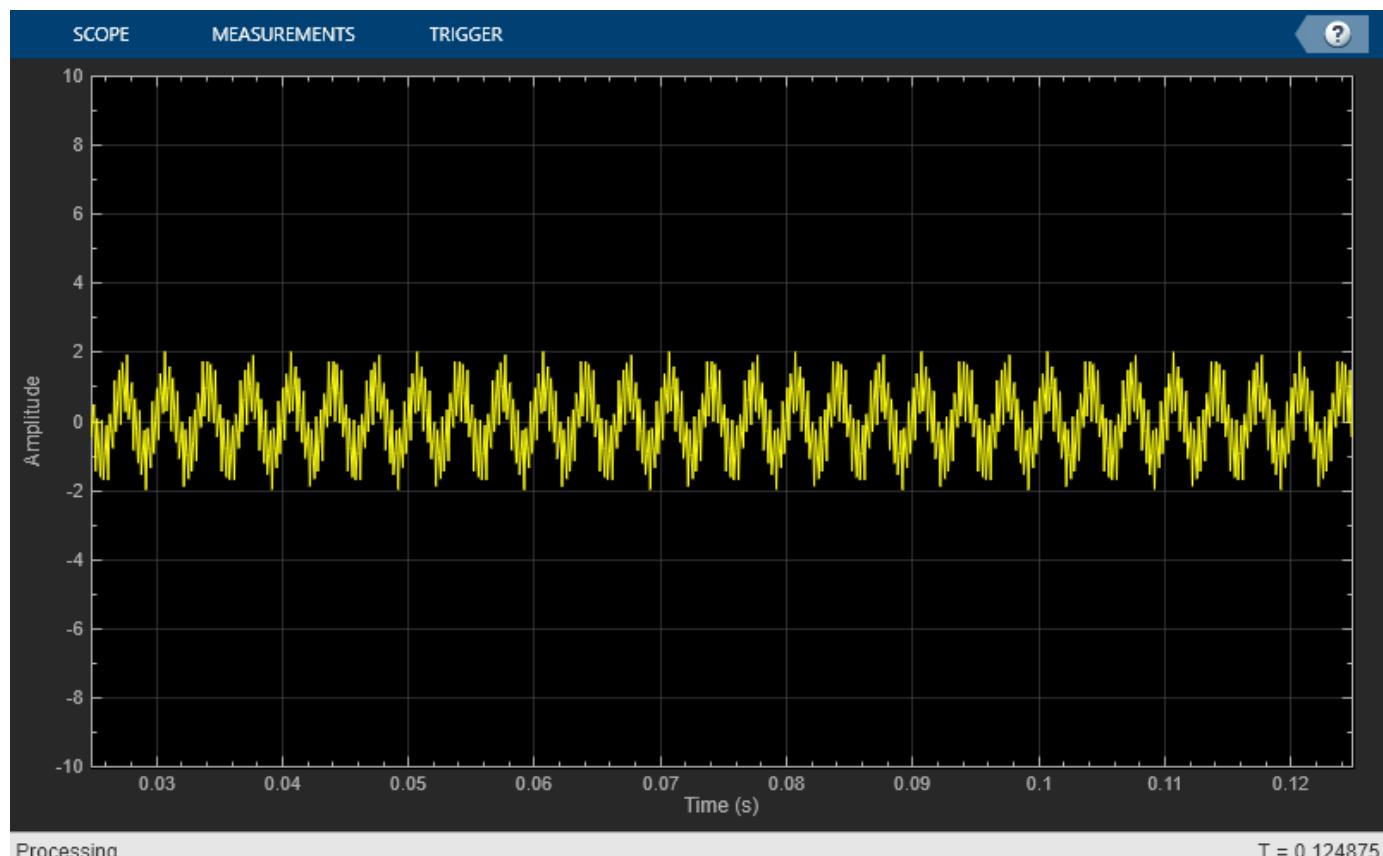
Create a time-domain sinusoidal signal. Display the signal by calling the time scope object.

Create a sinusoidal signal with two tones, one at 0.3 kHz and the other at 3 kHz.

```
t = (0:1000)'/8e3;
xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t);
```

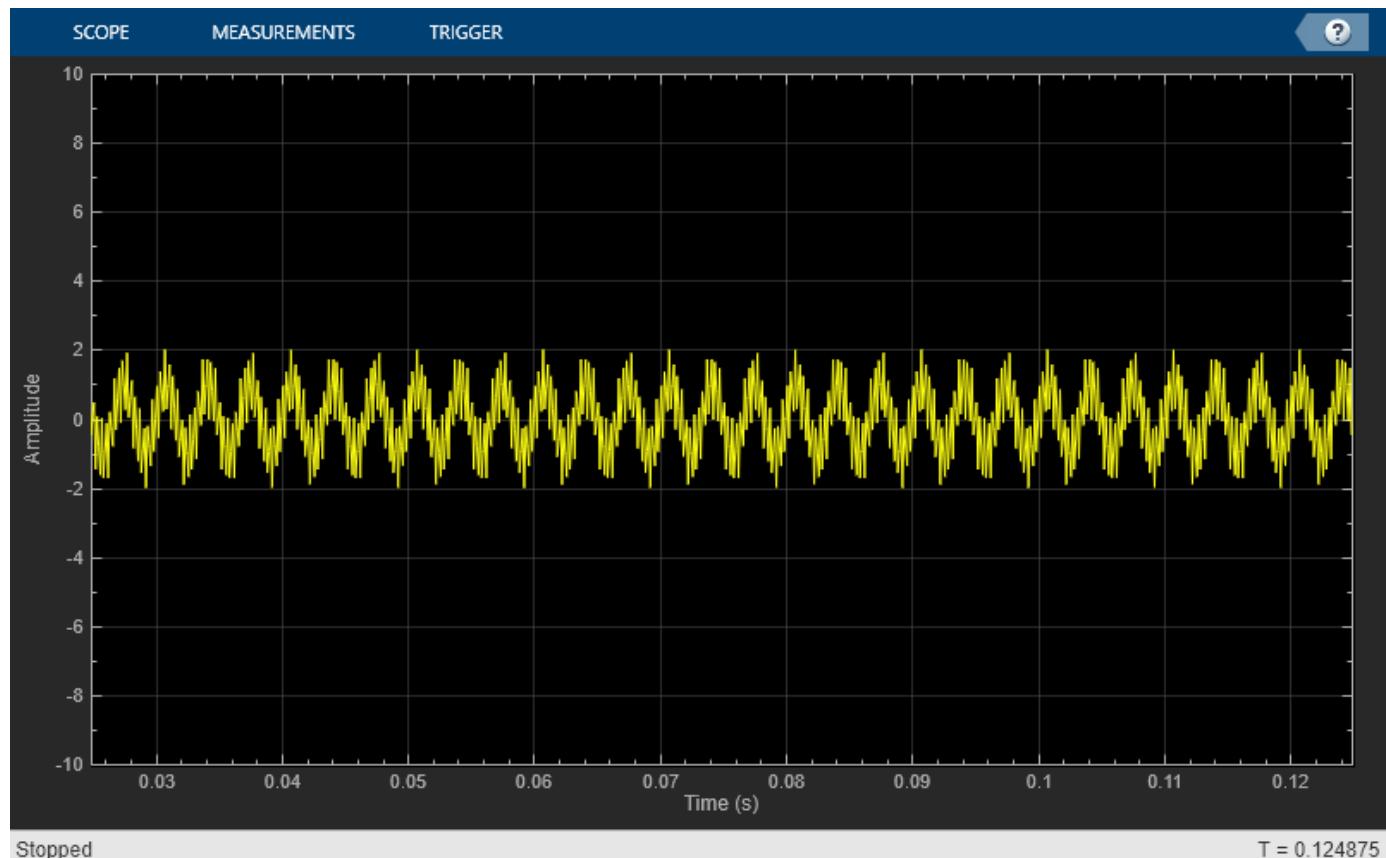
Create a `timescope` object and view the sinusoidal signal by calling the time scope object `scope`.

```
scope = timescope('SampleRate', 8e3, ...
    'TimeSpanSource', 'property', ...
    'TimeSpan', 0.1);
scope(xin)
```



Run `release` to allow changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope);
```

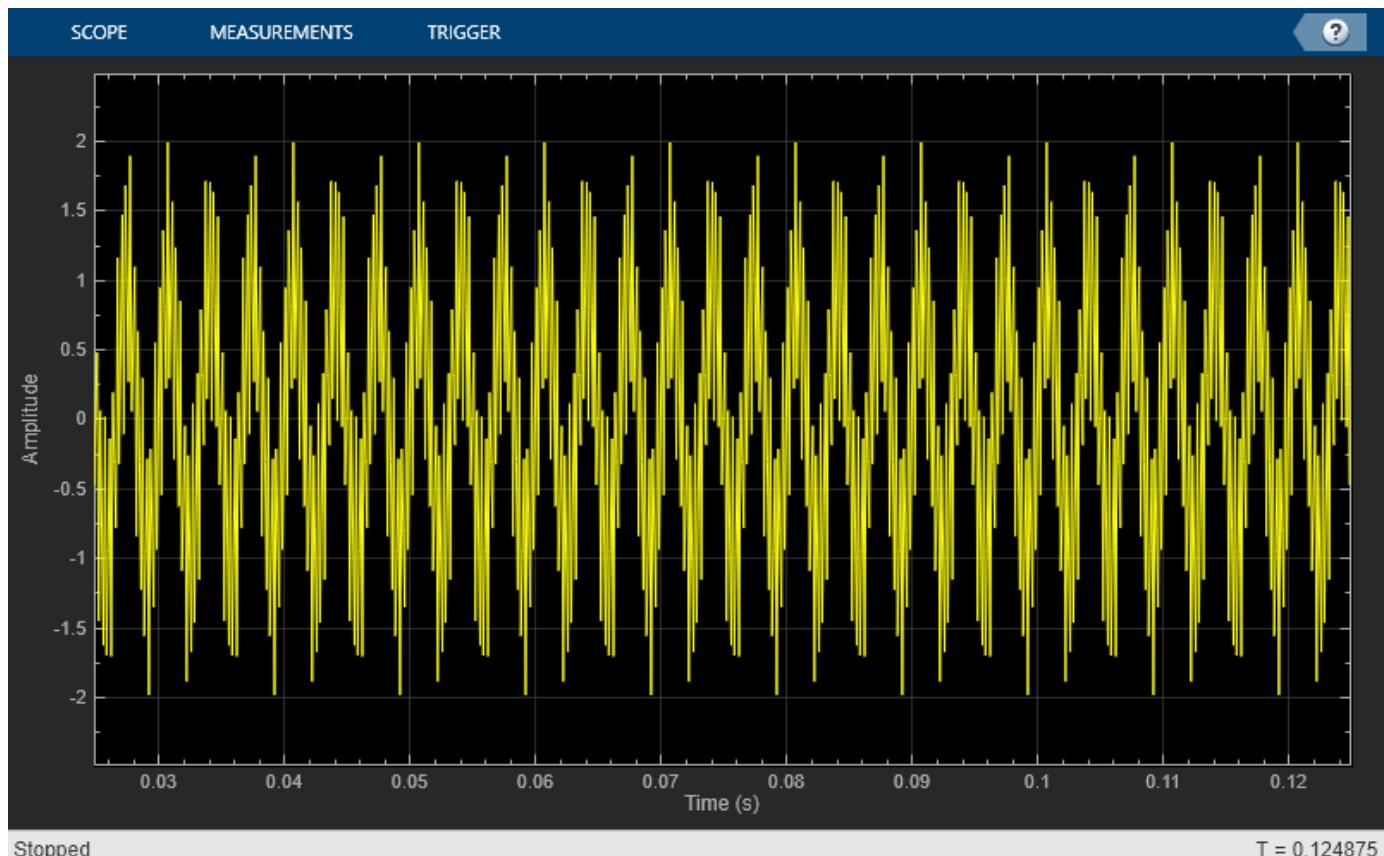


Hide the scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the scope window.

```
if(~isVisible(scope))
    show(scope)
end
```



## Input Arguments

### scope — Scope object

timescope object

Scope object whose window you want to hide, specified as a timescope object.

Example: `myScope = timescope; hide(myScope)`

## See Also

### Functions

`generateScript` | `isVisible` | `show`

### Objects

`timescope`

**Introduced in R2020a**

## isVisible

Determine visibility of scope

### Syntax

```
visibility = isVisible(scope)
```

### Description

`visibility = isVisible(scope)` returns the visibility of the scope as logical, with 1 (`true`) for visible.

### Examples

#### View Sine Wave on Time Scope

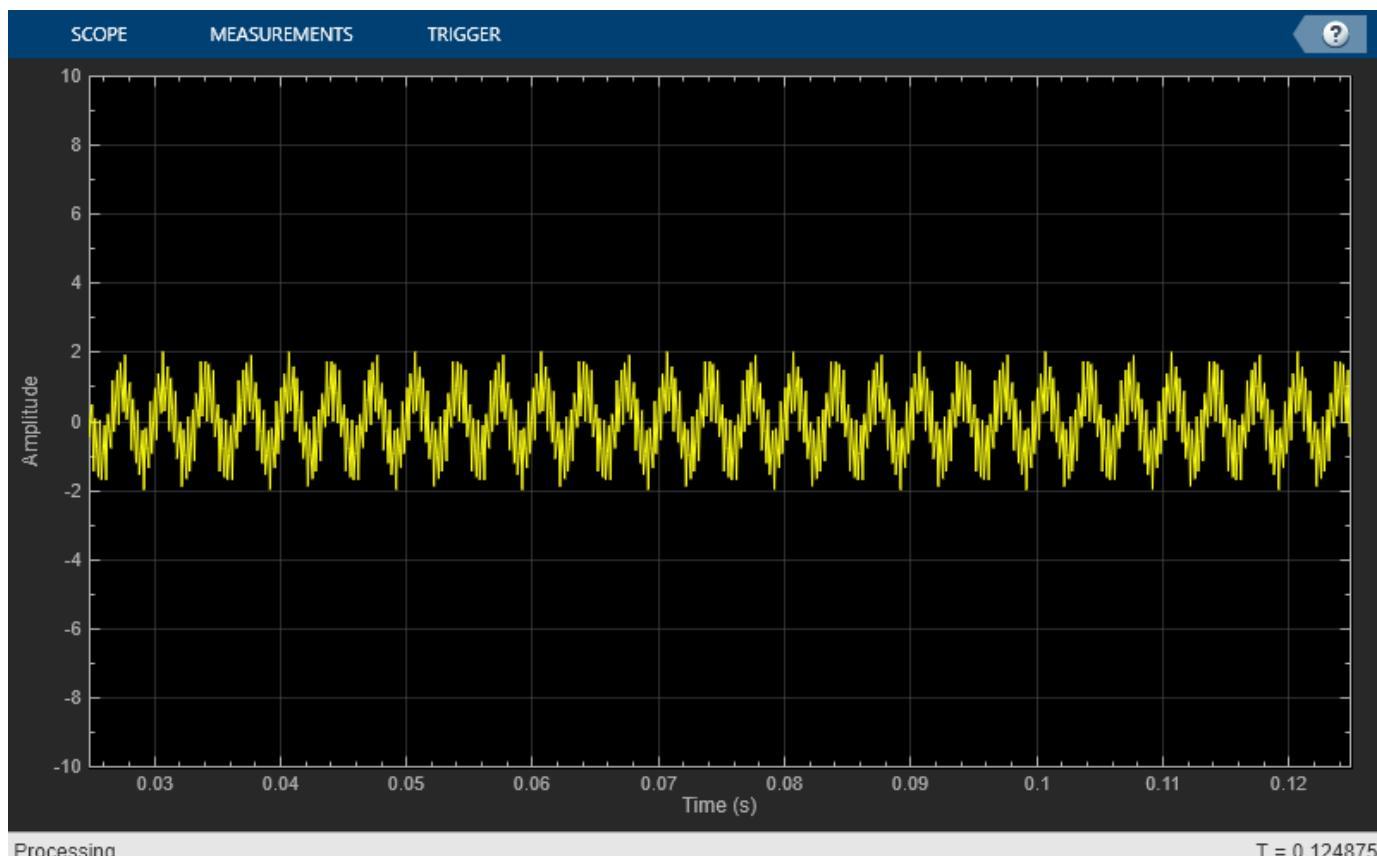
Create a time-domain sinusoidal signal. Display the signal by calling the time scope object.

Create a sinusoidal signal with two tones, one at 0.3 kHz and the other at 3 kHz.

```
t = (0:1000)'/8e3;
xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t);
```

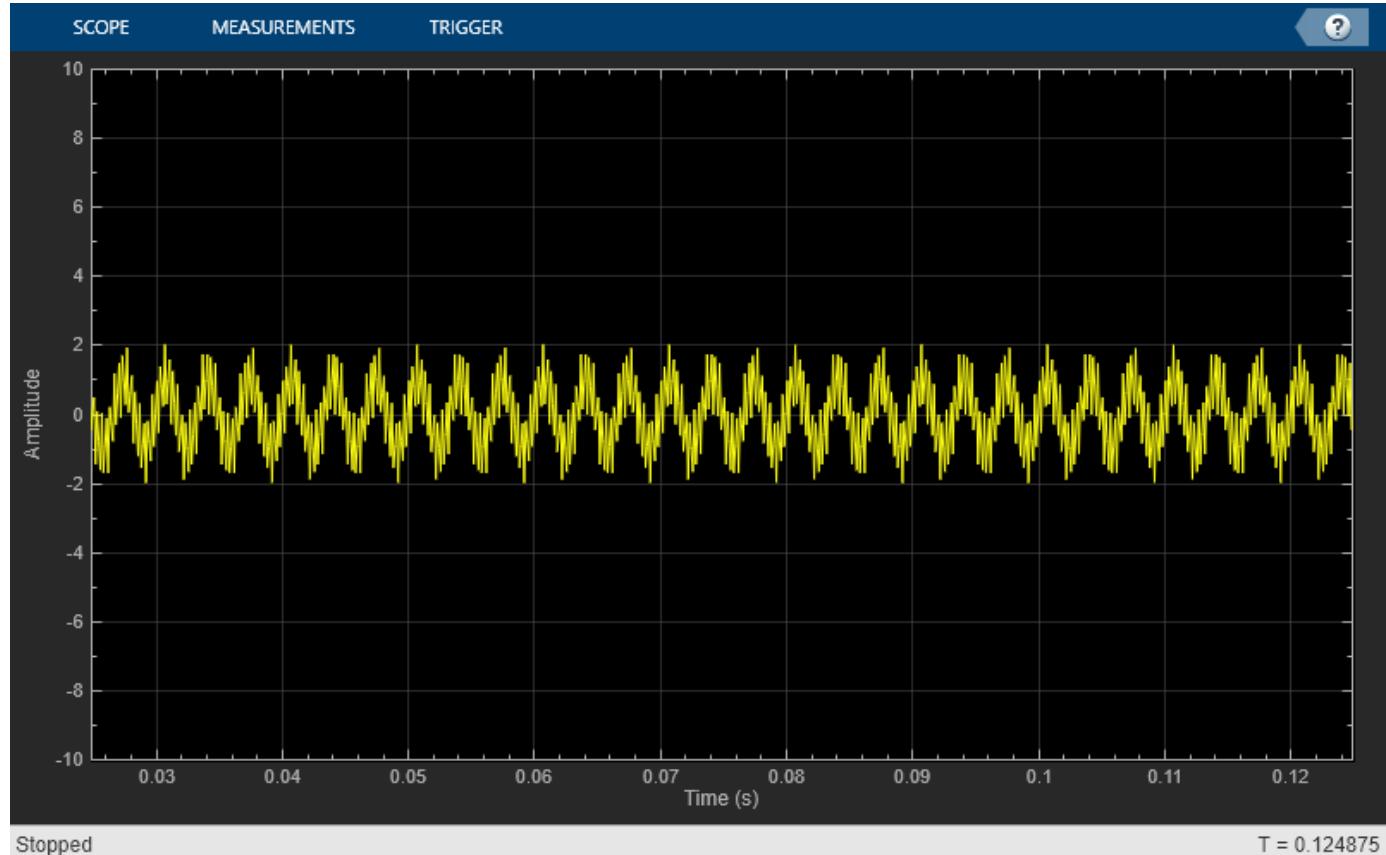
Create a `timescope` object and view the sinusoidal signal by calling the time scope object `scope`.

```
scope = timescope('SampleRate', 8e3, ...
    'TimeSpanSource', 'property', ...
    'TimeSpan', 0.1);
scope(xin)
```



Run `release` to allow changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope);
```

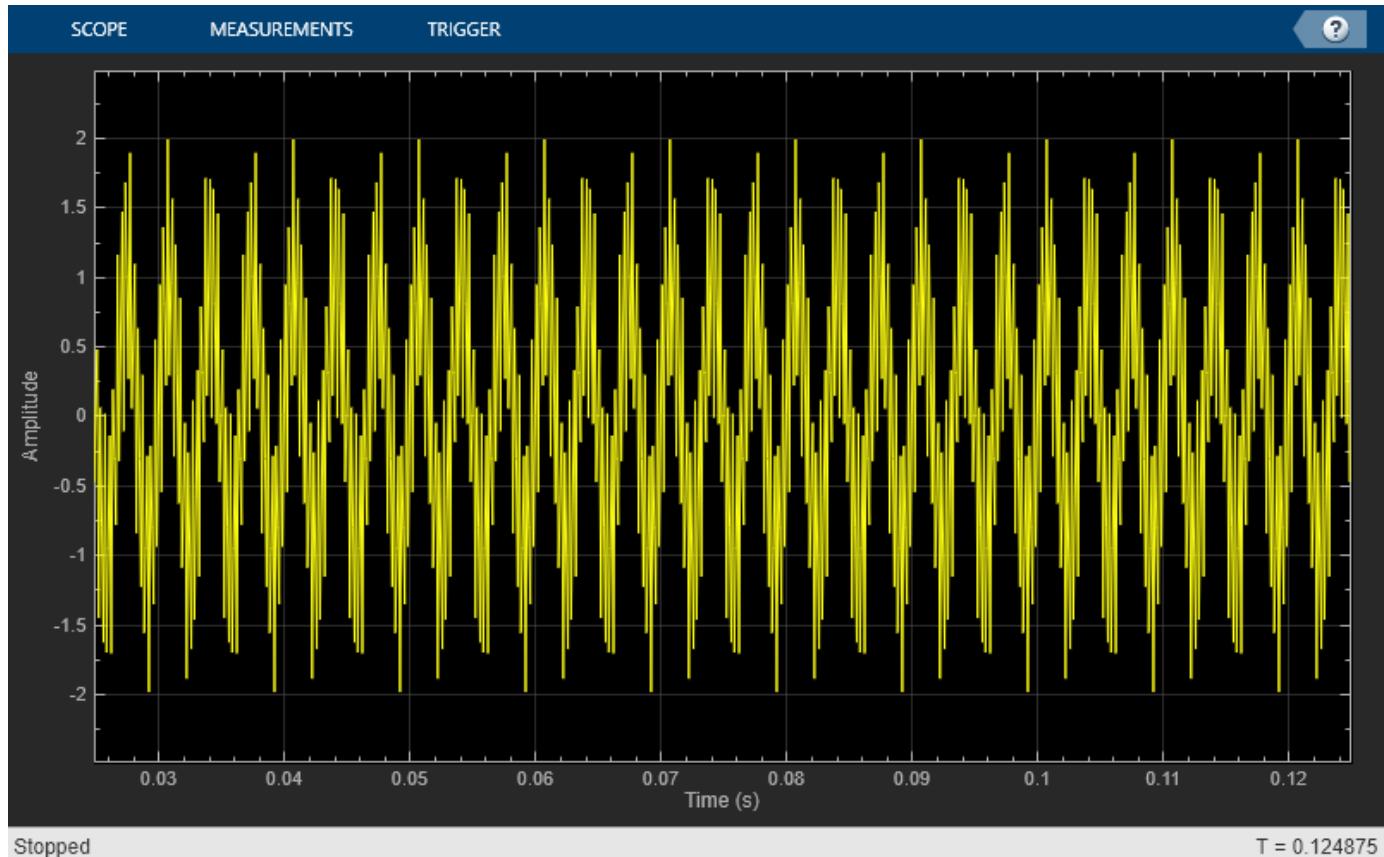


Hide the scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the scope window.

```
if(~isVisible(scope))
    show(scope)
end
```



## Input Arguments

**scope — Scope object**  
timescope object

Scope object whose visibility you want to query.

Example: `myScope = timescope; visibility = isVisible(myScope)`

## Output Arguments

**visibility — Scope visibility**  
`1 | 0`

If the scope window is open, the `isVisible` function returns `1` (`true`). Otherwise, the function returns `0` (`false`).

## See Also

**Functions**  
`generateScript` | `hide` | `show`

**Objects**  
timescope

**Introduced in R2020a**

# show

Display scope window

## Syntax

```
show(scope)
```

## Description

`show(scope)` shows the scope window.

## Examples

### View Sine Wave on Time Scope

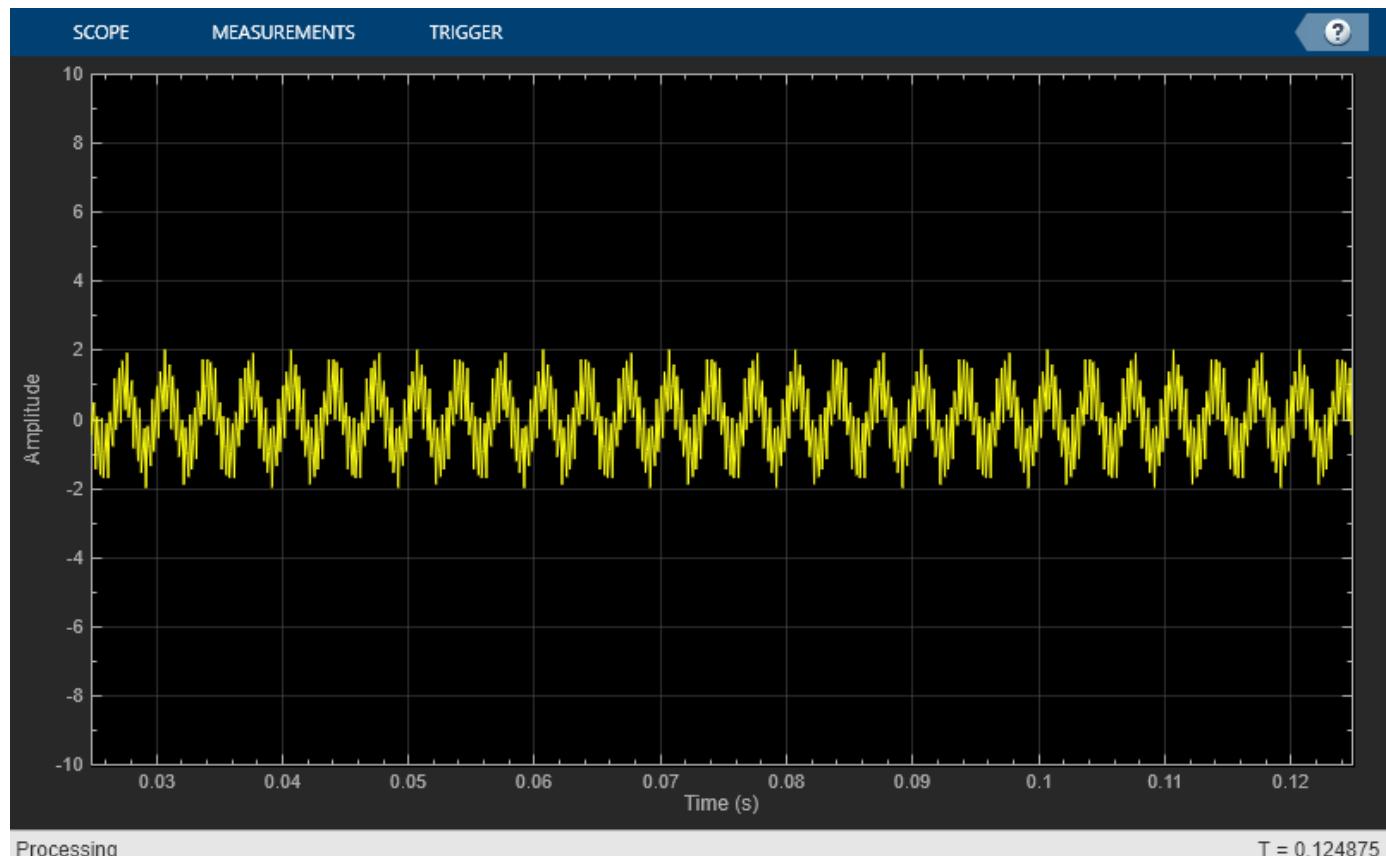
Create a time-domain sinusoidal signal. Display the signal by calling the time scope object.

Create a sinusoidal signal with two tones, one at 0.3 kHz and the other at 3 kHz.

```
t = (0:1000)'/8e3;
xin = sin(2*pi*0.3e3*t)+sin(2*pi*3e3*t);
```

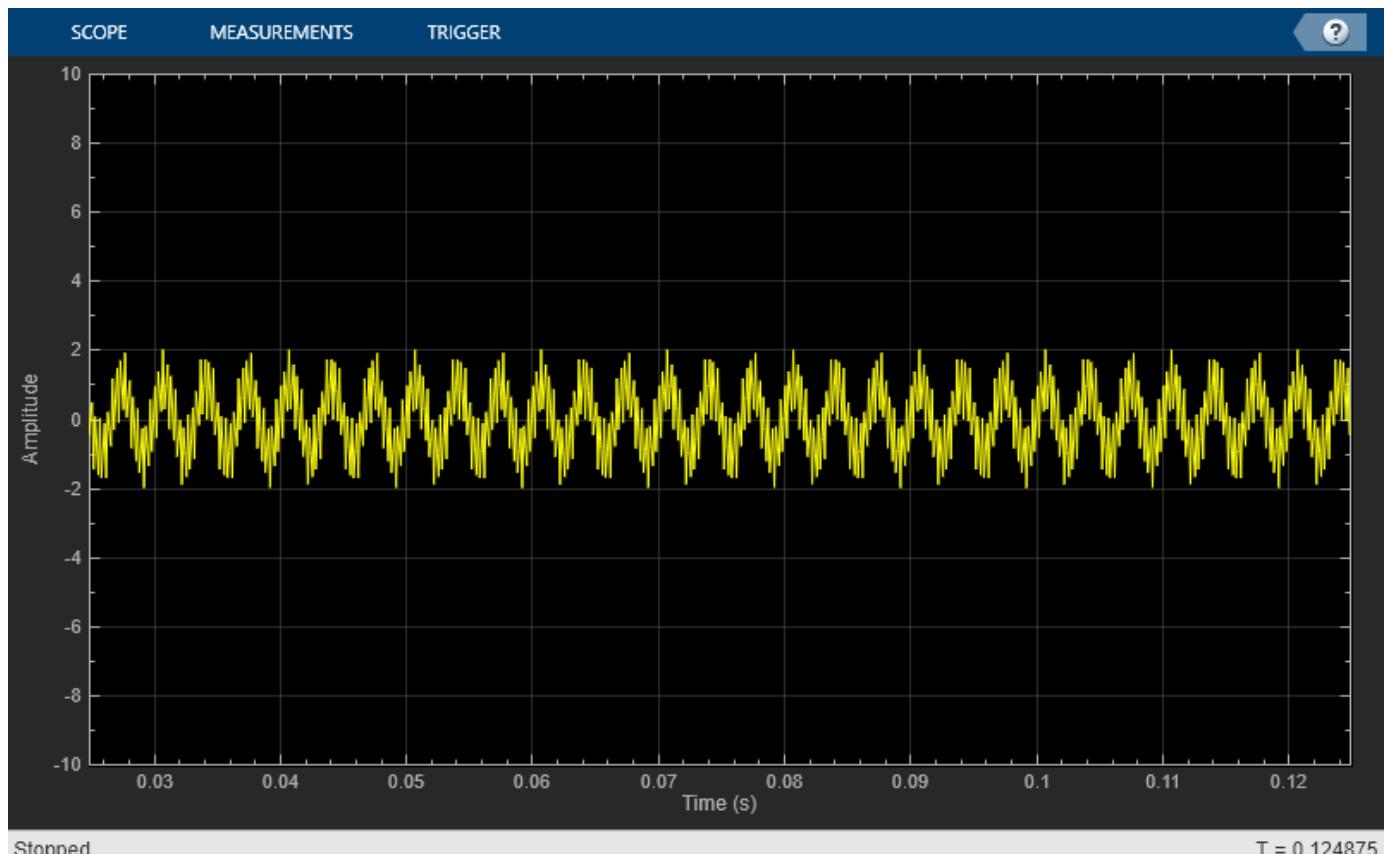
Create a `timescope` object and view the sinusoidal signal by calling the time scope object `scope`.

```
scope = timescope('SampleRate', 8e3, ...
    'TimeSpanSource', 'property', ...
    'TimeSpan', 0.1);
scope(xin)
```



Run `release` to allow changes to property values and input characteristics. The scope automatically scales the axes.

```
release(scope);
```

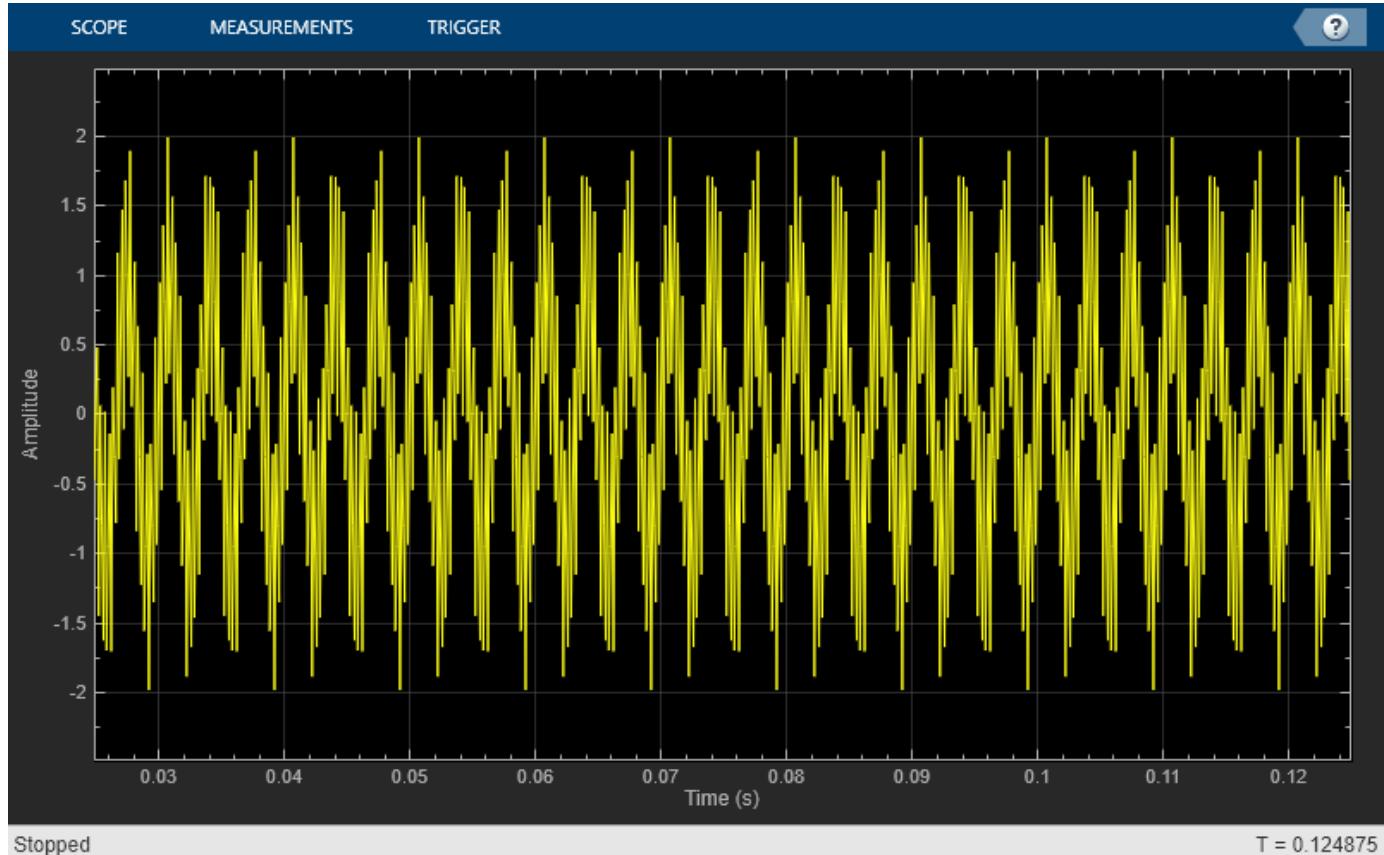


Hide the scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the scope window.

```
if(~isVisible(scope))
    show(scope)
end
```



## Input Arguments

### scope — Scope object

timescope object

Scope object whose window you want to show, specified as a timescope object.

Example: `myScope = timescope; show(myScope)`

## See Also

### Functions

`generateScript` | `hide` | `isVisible`

### Objects

`timescope`

**Introduced in R2020a**

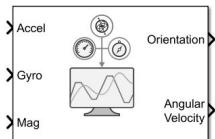
# Blocks

---

# AHRS

Orientation from accelerometer, gyroscope, and magnetometer readings

**Library:** Navigation Toolbox / Multisensor Positioning /  
Navigation Filters  
Sensor Fusion and Tracking Toolbox / Multisensor  
Positioning / Navigation Filters



## Description

The AHRS Simulink® block fuses accelerometer, magnetometer, and gyroscope sensor data to estimate device orientation.

## Ports

### Input

#### **Accel — Accelerometer readings in sensor body coordinate system (m/s<sup>2</sup>)**

*N*-by-3 matrix of real scalar

Accelerometer readings in the sensor body coordinate system in m/s<sup>2</sup>, specified as an *N*-by-3 matrix of real scalars. *N* is the number of samples, and the three columns of Accel represent the [x y z] measurements, respectively.

Data Types: single | double

#### **Gyro — Gyroscope readings in sensor body coordinate system (rad/s)**

*N*-by-3 matrix of real scalar

Gyroscope readings in the sensor body coordinate system in rad/s, specified as an *N*-by-3 matrix of real scalars. *N* is the number of samples, and the three columns of Gyro represent the [x y z] measurements, respectively.

Data Types: single | double

#### **Mag — Magnetometer readings in sensor body coordinate system (µT)**

*N*-by-3 matrix of real scalar

Magnetometer readings in the sensor body coordinate system in µT, specified as an *N*-by-3 matrix of real scalars. *N* is the number of samples, and the three columns of Mag represent the [x y z] measurements, respectively.

Data Types: single | double

### Output

#### **Orientation — Orientation of sensor body frame relative to navigation frame**

*M*-by-4 array of scalar | 3-by-3-by-*M*-element rotation matrix

Orientation of the sensor body frame relative to the navigation frame, return as an  $M$ -by-4 array of scalars or a 3-by-3-by- $M$  array of rotation matrices. Each row of the  $N$ -by-4 array is assumed to be the four elements of a quaternion. The number of input samples,  $N$ , and the **Decimation Factor** parameter determine the output size  $M$ .

Data Types: `single` | `double`

### **Angular Velocity — Angular velocity in sensor body coordinate system (rad/s)**

$M$ -by-3 array of real scalar (default)

Angular velocity with gyroscope bias removed in the sensor body coordinate system in rad/s, returned as an  $M$ -by-3 array of real scalars. The number of input samples,  $N$ , and the **Decimation Factor** parameter determine the output size  $M$ .

Data Types: `single` | `double`

## Parameters

### Main

#### **Reference frame — Navigation reference frame**

`NED` (default) | `ENU`

Navigation reference frame, specified as `NED` (North-East-Down) or `ENU` (East-North-Up).

#### **Decimation factor — Decimation factor**

`1` (default) | positive integer

Decimation factor by which to reduce the input sensor data rate, specified as a positive integer.

The number of rows of the inputs -- **Accel**, **Gyro**, and **Mag** -- must be a multiple of the decimation factor.

Data Types: `single` | `double`

#### **Initial process noise — Initial process noise**

`ahrsfilter.defaultProcessNoise` (default) | 12-by-12 matrix of real scalar

Initial process noise, specified as a 12-by-12 matrix of real scalars. The default value, `ahrsfilter.defaultProcessNoise`, is a 12-by-12 diagonal matrix as:

Columns 1 through 6

0.000006092348396	0	0	0	0	0
0	0.000006092348396	0	0	0	0
0	0	0.000006092348396	0	0	0
0	0	0	0.000076154354947	0	0.000076154354947
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Columns 7 through 12

```

0          0          0          0
0          0          0          0
0          0          0          0
0          0          0          0
0          0          0          0
0          0          0          0
0          0          0          0
0.009623610000000          0          0          0
0  0.009623610000000          0          0
0          0  0.009623610000000          0
0          0          0  0.600000000000000
0          0          0          0
0          0          0          0

```

Data Types: single | double

#### **Orientation format — Orientation output format**

'quaternion' (default) | 'Rotation matrix'

Output orientation format, specified as 'quaternion' or 'Rotation matrix':

- 'quaternion' -- Output is an  $M$ -by-4 array of real scalars. Each row of the array represents the four components of a quaternion.
- 'Rotation matrix' -- Output is a 3-by-3-by- $M$  rotation matrix.

The output size  $M$  depends on the input dimension  $N$  and the **Decimation Factor** parameter.

Data Types: char | string

#### **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 code. The first time you run a simulation, Simulink generates 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.

#### **Measurement Noise**

##### **Accelerometer noise ( $(\text{m/s}^2)^2$ ) — Variance of accelerometer signal noise ( $(\text{m/s}^2)^2$ )**

0.00019247 (default) | positive real scalar

Variance of accelerometer signal noise in  $(\text{m/s}^2)^2$ , specified as a positive real scalar.

Data Types: single | double

##### **Gyroscope noise ( $(\text{rad/s})^2$ ) — Variance of gyroscope signal noise ( $(\text{rad/s})^2$ )**

9.1385e-5 (default) | positive real scalar

Variance of gyroscope signal noise in  $(\text{rad/s})^2$ , specified as a positive real scalar.

Data Types: single | double

##### **Magnetometer noise ( $\mu\text{T}^2$ ) — Variance of magnetometer signal noise ( $\mu\text{T}^2$ )**

0.1 (default) | positive real scalar

Variance of magnetometer signal noise in  $\mu\text{T}^2$ , specified as a positive real scalar.

Data Types: single | double

**Gyroscope drift noise (rad/s) — Variance of gyroscope offset drift ((rad/s) $^2$ )**

3.0462e-13 (default) | positive real scalar

Variance of gyroscope offset drift in  $(\text{rad}/\text{s})^2$ , specified as a positive real scalar.

Data Types: single | double

### Environmental Noise

**Linear acceleration noise ((m/s $^2$ ) $^2$ ) — Variance of linear acceleration noise (m/s $^2$ ) $^2$**

0.0096236 (default) | positive real scalar

Variance of linear acceleration noise in  $(\text{m}/\text{s}^2)^2$ , specified as a positive real scalar. Linear acceleration is modeled as a lowpass-filtered white noise process.

Data Types: single | double

**Magnetic disturbance noise ( $\mu\text{T}^2$ ) — Variance of magnetic disturbance noise ( $\mu\text{T}^2$ )**

0.5 (default) | real finite positive scalar

Variance of magnetic disturbance noise in  $\mu\text{T}^2$ , specified as a real finite positive scalar.

Data Types: single | double

**Linear acceleration decay factor — Decay factor for linear acceleration drift**

0.5 (default) | scalar in the range [0,1)

Decay factor for linear acceleration drift, specified as a scalar in the range [0,1]. If linear acceleration changes quickly, set this parameter to a lower value. If linear acceleration changes slowly, set this parameter to a higher value. Linear acceleration drift is modeled as a lowpass-filtered white noise process.

Data Types: single | double

**Magnetic disturbance decay factor — Decay factor for magnetic disturbance**

0.5 (default) | positive scalar in the range [0,1]

Decay factor for magnetic disturbance, specified as a positive scalar in the range [0,1]. Magnetic disturbance is modeled as a first order Markov process.

Data Types: single | double

**Magnetic field strength ( $\mu\text{T}$ ) — Magnetic field strength ( $\mu\text{T}$ )**

50 (default) | real positive scalar

Magnetic field strength in  $\mu\text{T}$ , specified as a real positive scalar. The magnetic field strength is an estimate of the magnetic field strength of the Earth at the current location.

Data Types: single | double

## Algorithms

*Note: The following algorithm only applies to an NED reference frame.*

The AHRS block uses the nine-axis Kalman filter structure described in [1]. The algorithm attempts to track the errors in orientation, gyroscope offset, linear acceleration, and magnetic disturbance to output the final orientation and angular velocity. Instead of tracking the orientation directly, the indirect Kalman filter models the error process,  $x$ , with a recursive update:

$$x_k = \begin{bmatrix} \theta_k \\ b_k \\ a_k \\ d_k \end{bmatrix} = F_k \begin{bmatrix} \theta_{k-1} \\ b_{k-1} \\ a_{k-1} \\ d_{k-1} \end{bmatrix} + w_k$$

where  $x_k$  is a 12-by-1 vector consisting of:

- $\theta_k$  -- 3-by-1 orientation error vector, in degrees, at time  $k$
- $b_k$  -- 3-by-1 gyroscope zero angular rate bias vector, in deg/s, at time  $k$
- $a_k$  -- 3-by-1 acceleration error vector measured in the sensor frame, in g, at time  $k$
- $d_k$  -- 3-by-1 magnetic disturbance error vector measured in the sensor frame, in  $\mu\text{T}$ , at time  $k$

and where  $w_k$  is a 12-by-1 additive noise vector, and  $F_k$  is the state transition model.

Because  $x_k$  is defined as the error process, the *a priori* estimate is always zero, and therefore the state transition model,  $F_k$ , is zero. This insight results in the following reduction of the standard Kalman equations:

Standard Kalman equations:

$$\begin{aligned} x_k^- &= F_k x_{k-1}^+ \\ P_k^- &= F_k P_{k-1}^+ F_k^T + Q_k \\ y_k &= z_k - H_k x_k^- \\ S_k &= R_k + H_k P_k^- H_k^T \\ K_k &= P_k^- H_k^T (S_k)^{-1} \\ x_k^+ &= x_k^- + K_k y_k \\ P_k^+ &= P_k^- - K_k H_k P_k^- \end{aligned}$$

Kalman equations used in this algorithm:

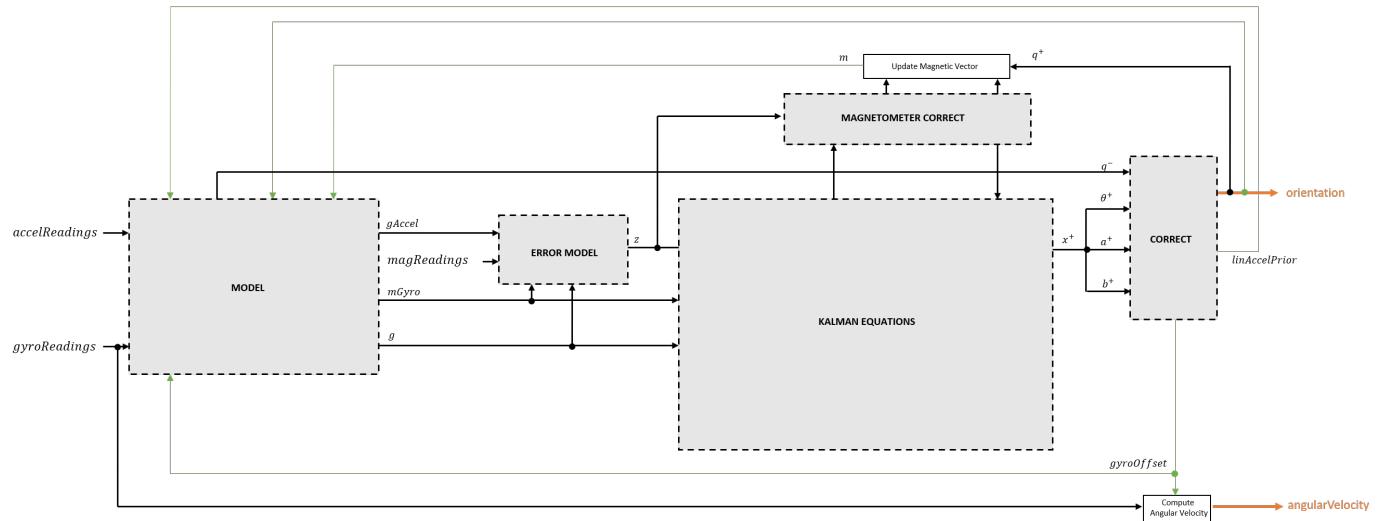
$$\begin{aligned} x_k^- &= 0 \\ P_k^- &= Q_k \\ y_k &= z_k \\ S_k &= R_k + H_k P_k^- H_k^T \\ K_k &= P_k^- H_k^T (S_k)^{-1} \\ x_k^+ &= K_k y_k \\ P_k^+ &= P_k^- - K_k H_k P_k^- \end{aligned}$$

where:

- $x_k^-$  -- predicted (*a priori*) state estimate; the error process
- $P_k^-$  -- predicted (*a priori*) estimate covariance
- $y_k$  -- innovation
- $S_k$  -- innovation covariance
- $K_k$  -- Kalman gain
- $x_k^+$  -- updated (*a posteriori*) state estimate
- $P_k^+$  -- updated (*a posteriori*) estimate covariance

$k$  represents the iteration, the superscript  $+$  represents an *a posteriori* estimate, and the superscript  $-$  represents an *a priori* estimate.

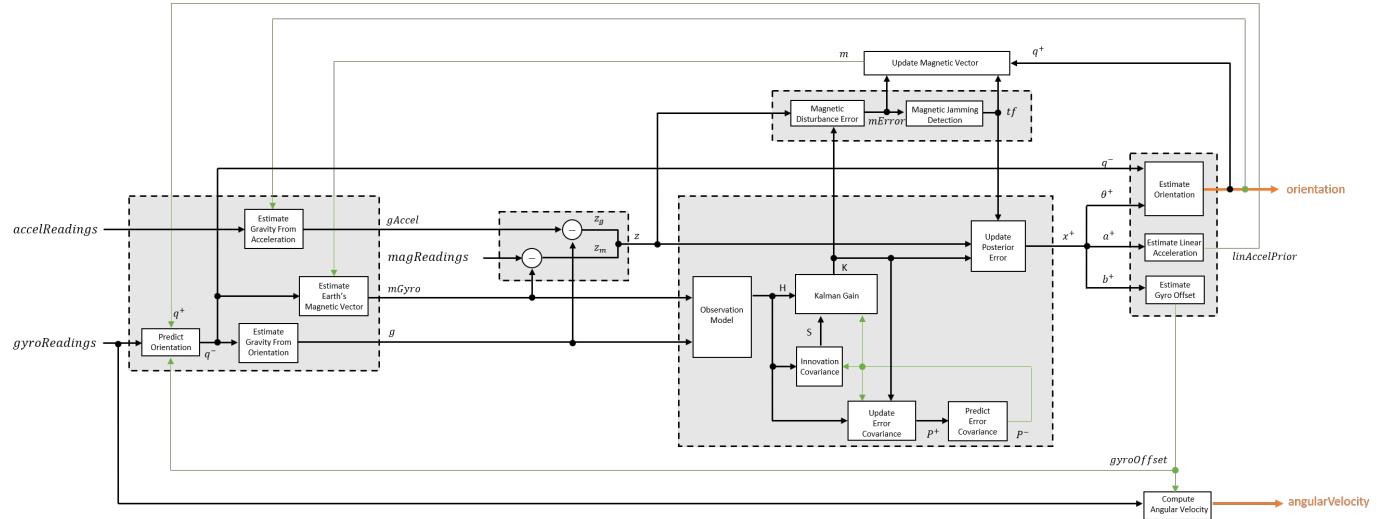
The graphic and following steps describe a single frame-based iteration through the algorithm.



Before the first iteration, the `accelReadings`, `gyroReadings`, and `magReadings` inputs are chunked into `DecimationFactor`-by-3 frames. For each chunk, the algorithm uses the most current accelerometer and magnetometer readings corresponding to the chunk of gyroscope readings.

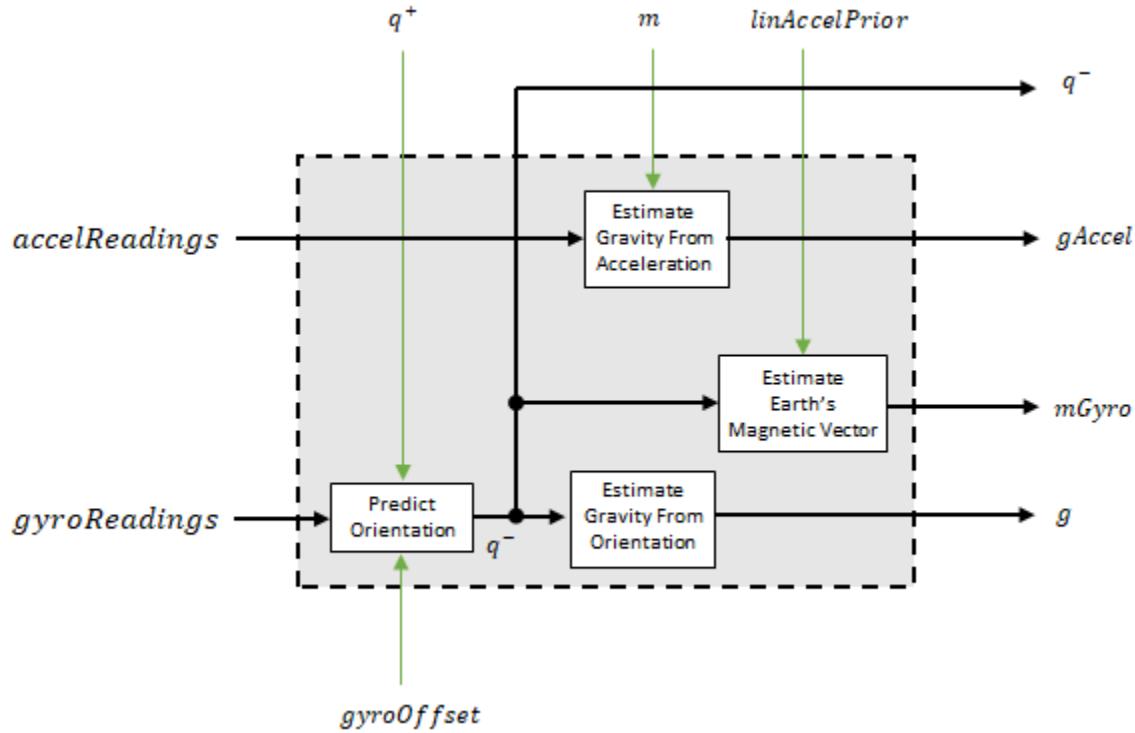
## Detailed Overview

Walk through the algorithm for an explanation of each stage of the detailed overview.



### Model

The algorithm models acceleration and angular change as linear processes.



### Predict Orientation

The orientation for the current frame is predicted by first estimating the angular change from the previous frame:

$$\Delta\varphi_{N \times 3} = \frac{(gyroReadings_{N \times 3} - gyroOffset_{1 \times 3})}{fs}$$

where  $N$  is the decimation factor specified by the Decimation factor and  $fs$  is the sample rate.

The angular change is converted into quaternions using the `rotvec` quaternion construction syntax:

$$\Delta Q_{N \times 1} = \text{quaternion}(\Delta\varphi_{N \times 3}, 'rotvec')$$

The previous orientation estimate is updated by rotating it by  $\Delta Q$ :

$$q^-_{1 \times 1} = (q^+_{1 \times 1}) \left( \prod_{n=1}^N \Delta Q_n \right)$$

During the first iteration, the orientation estimate,  $q^-$ , is initialized by `ecompass`.

### Estimate Gravity from Orientation

The gravity vector is interpreted as the third column of the quaternion,  $q^-$ , in rotation matrix form:

$$g_{1 \times 3} = (rPrior(:, 3))^T$$

### Estimate Gravity from Acceleration

A second gravity vector estimation is made by subtracting the decayed linear acceleration estimate of the previous iteration from the accelerometer readings:

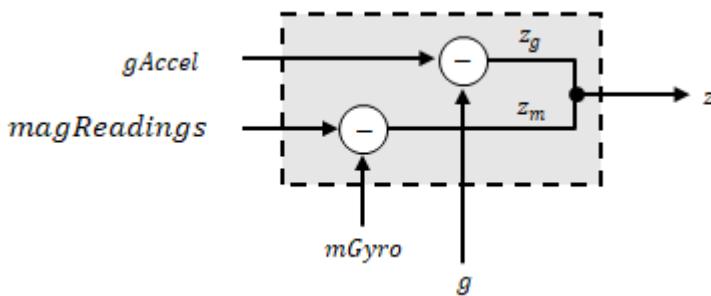
$$gAccel_{1 \times 3} = accelReadings_{1 \times 3} - linAccelprior_{1 \times 3}$$

### Estimate Earth's Magnetic Vector

Earth's magnetic vector is estimated by rotating the magnetic vector estimate from the previous iteration by the *a priori* orientation estimate, in rotation matrix form:

$$mGyro_{1 \times 3} = ((rPrior)(m^T))^T$$

### Error Model

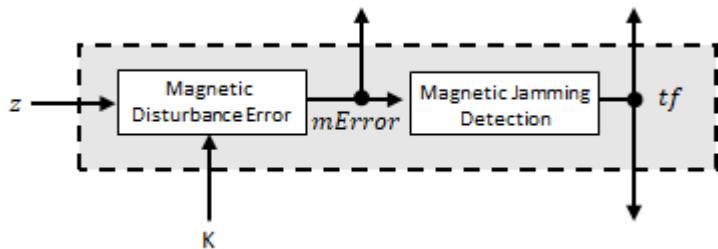


The error model combines two differences:

- The difference between the gravity estimate from the accelerometer readings and the gravity estimate from the gyroscope readings:  $z_g = g - gAccel$
- The difference between the magnetic vector estimate from the gyroscope readings and the magnetic vector estimate from the magnetometer:  $z_m = mGyro - magReadings$

## Magnetometer Correct

The magnetometer correct estimates the error in the magnetic vector estimate and detects magnetic jamming.



## Magnetometer Disturbance Error

The magnetic disturbance error is calculated by matrix multiplication of the Kalman gain associated with the magnetic vector with the error signal:

$$mError_{3 \times 1} = \left( (K(10:12, :)_{3 \times 6})(z_{1 \times 6})^T \right)^T$$

The Kalman gain,  $K$ , is the Kalman gain calculated in the current iteration.

## Magnetic Jamming Detection

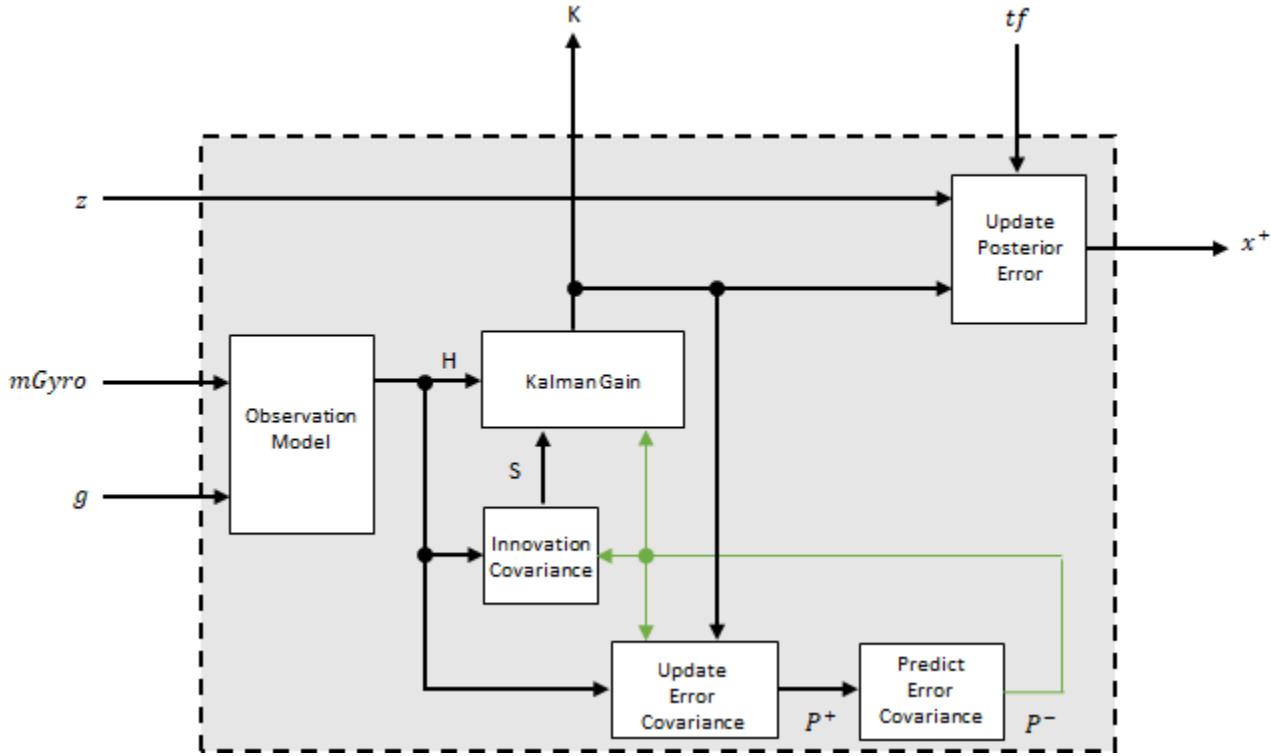
Magnetic jamming is determined by verifying that the power of the detected magnetic disturbance is less than or equal to four times the power of the expected magnetic field strength:

$$tf = \begin{cases} \text{true} & \text{if } \sum |mError|^2 > (4)(\text{ExpectedMagneticFieldStrength})^2 \\ \text{false} & \text{else} \end{cases}$$

`ExpectedMagneticFieldStrength` is a property of `ahrsfilter`.

## Kalman Equations

The Kalman equations use the gravity estimate derived from the gyroscope readings,  $g$ , the magnetic vector estimate derived from the gyroscope readings,  $mGyro$ , and the observation of the error process,  $z$ , to update the Kalman gain and intermediary covariance matrices. The Kalman gain is applied to the error signal,  $z$ , to output an *a posteriori* error estimate,  $x^+$ .



### Observation Model

The observation model maps the 1-by-3 observed states,  $g$  and  $mGyro$ , into the 6-by-12 true state,  $H$ .

The observation model is constructed as:

$$H_{3 \times 9} = \begin{bmatrix} 0 & g_z & -g_y & 0 & -\kappa g_z & \kappa g_y & 1 & 0 & 0 & 0 & 0 & 0 \\ -g_z & 0 & g_x & \kappa g_z & 0 & -\kappa g_x & 0 & 1 & 0 & 0 & 0 & 0 \\ g_y & -g_x & 0 & -\kappa g_y & \kappa g_x & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & m_z & -m_y & 0 & -\kappa m_z & -\kappa m_y & 0 & 0 & 0 & -1 & 0 & 0 \\ -m_z & 0 & m_x & \kappa m_z & 0 & -\kappa m_x & 0 & 0 & 0 & 0 & -1 & 0 \\ m_y & -m_x & 0 & -\kappa m_y & \kappa m_x & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

where  $g_x$ ,  $g_y$ , and  $g_z$  are the  $x$ -,  $y$ -, and  $z$ -elements of the gravity vector estimated from the *a priori* orientation, respectively.  $m_x$ ,  $m_y$ , and  $m_z$  are the  $x$ -,  $y$ -, and  $z$ -elements of the magnetic vector estimated from the *a priori* orientation, respectively.  $\kappa$  is a constant determined by the Sample rate and Decimation factor properties:  $\kappa = \text{Decimation factor}/\text{Sample rate}$ .

### Innovation Covariance

The innovation covariance is a 6-by-6 matrix used to track the variability in the measurements. The innovation covariance matrix is calculated as:

$$S_{6 \times 6} = R_{6 \times 6} + (H_{6 \times 12})(P_{12 \times 12}^-)(H_{6 \times 12})^T$$

where

- $H$  is the observation model matrix
- $P^-$  is the predicted (*a priori*) estimate of the covariance of the observation model calculated in the previous iteration
- $R$  is the covariance of the observation model noise, calculated as:

$$R_{6 \times 6} = \begin{bmatrix} accel_{noise} & 0 & 0 & 0 & 0 & 0 \\ 0 & accel_{noise} & 0 & 0 & 0 & 0 \\ 0 & 0 & accel_{noise} & 0 & 0 & 0 \\ 0 & 0 & 0 & mag_{noise} & 0 & 0 \\ 0 & 0 & 0 & 0 & mag_{noise} & 0 \\ 0 & 0 & 0 & 0 & 0 & mag_{noise} \end{bmatrix}$$

where

$$\begin{aligned} accel_{noise} &= \text{AccelerometerNoise} + \text{LinearAccelerationNoise} + \kappa^2 \\ &(\text{GyroscopeDriftNoise} + \text{GyroscopeNoise}) \end{aligned}$$

and

$$\begin{aligned} mag_{noise} &= \text{MagnetometerNoise} + \text{MagneticDisturbanceNoise} + \kappa^2 \\ &(\text{GyroscopeDriftNoise} + \text{GyroscopeNoise}) \end{aligned}$$

### Update Error Estimate Covariance

The error estimate covariance is a 12-by-12 matrix used to track the variability in the state.

The error estimate covariance matrix is updated as:

$$P_{12 \times 12}^+ = P_{12 \times 12}^- - (K_{12 \times 6})(H_{6 \times 12})(P_{12 \times 12}^-)$$

where  $K$  is the Kalman gain,  $H$  is the measurement matrix, and  $P^-$  is the error estimate covariance calculated during the previous iteration.

### Predict Error Estimate Covariance

The error estimate covariance is a 12-by-12 matrix used to track the variability in the state. The *a priori* error estimate covariance,  $P^-$ , is set to the process noise covariance,  $Q$ , determined during the previous iteration.  $Q$  is calculated as a function of the *a posteriori* error estimate covariance,  $P^+$ . When calculating  $Q$ , it is assumed that the cross-correlation terms are negligible compared to the autocorrelation terms, and are set to zero:

$Q =$

$$\begin{bmatrix}
 P^+(1) + \kappa^2 P^+(40) + \beta + \eta & 0 & 0 & -\kappa(P^+(40) + \beta) & 0 \\
 0 & P^+(14) + \kappa^2 P^+(53) + \beta + \eta & 0 & 0 & -\kappa(P^+(53) + \beta) \\
 0 & 0 & P^+(27) + \kappa^2 P^+(66) + \beta + \eta & 0 & 0 \\
 -\kappa(P^+(40) + \beta) & 0 & 0 & P^+(40) + \beta & 0 \\
 0 & -\kappa(P^+(53) + \beta) & 0 & 0 & P^+(53) + \beta \\
 0 & 0 & -\kappa(P^+(66) + \beta) & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0
 \end{bmatrix}$$

where

- $P^+$  -- is the updated (*a posteriori*) error estimate covariance
- $\kappa$  -- Decimation factor divided by sample rate.
- $\beta$  -- Gyroscope drift noise.
- $\eta$  -- Gyroscope noise.
- $\nu$  -- Linear acceleration decay factor.
- $\xi$  -- Linear acceleration noise.
- $\sigma$  -- Magnetic disturbance decay factor.
- $\gamma$  -- Magnetic disturbance noise.

### Kalman Gain

The Kalman gain matrix is a 12-by-6 matrix used to weight the innovation. In this algorithm, the innovation is interpreted as the error process,  $z$ .

The Kalman gain matrix is constructed as:

$$K_{12 \times 6} = (P_{12 \times 12}^-)^T (H_{6 \times 12})^T (S_{6 \times 6})^{-1}$$

where

- $P^-$  -- predicted error covariance
- $H$  -- observation model
- $S$  -- innovation covariance

### Update a Posteriori Error

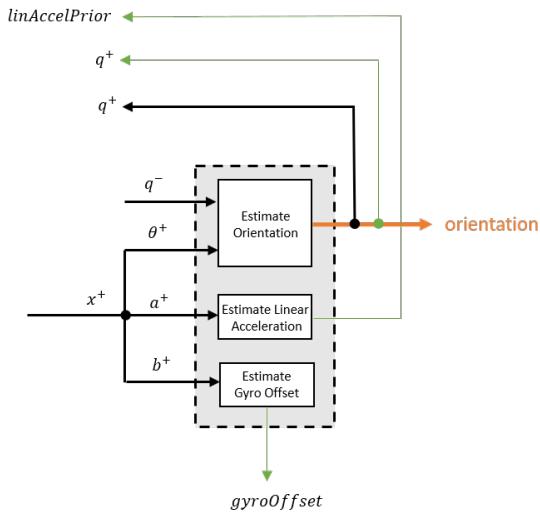
The *a posteriori* error estimate is determined by combining the Kalman gain matrix with the error in the gravity vector and magnetic vector estimations:

$$x_{12 \times 1} = (K_{12 \times 6})(z_1 \times 6)^T$$

If magnetic jamming is detected in the current iteration, the magnetic vector error signal is ignored, and the *a posteriori* error estimate is calculated as:

$$x_{9 \times 1} = (K(1:9, 1:3)(z_g)^T$$

## Correct



### Estimate Orientation

The orientation estimate is updated by multiplying the previous estimation by the error:

$$q^+ = (q^-)(\theta^+)$$

### Estimate Linear Acceleration

The linear acceleration estimation is updated by decaying the linear acceleration estimation from the previous iteration and subtracting the error:

$$\text{linAccelPrior} = (\text{linAccelPrior}_{k-1})\nu - b^+$$

where

- $\nu$  -- Linear acceleration decay factor

### Estimate Gyroscope Offset

The gyroscope offset estimation is updated by subtracting the gyroscope offset error from the gyroscope offset from the previous iteration:

$$\text{gyroOffset} = \text{gyroOffset}_{k-1} - a^+$$

### Compute Angular Velocity

To estimate angular velocity, the frame of `gyroReadings` are averaged and the gyroscope offset computed in the previous iteration is subtracted:

$$\text{angularVelocity}_{1 \times 3} = \frac{\sum \text{gyroReadings}_{N \times 3}}{N} - \text{gyroOffset}_{1 \times 3}$$

where  $N$  is the decimation factor specified by the `DecimationFactor` property.

The gyroscope offset estimation is initialized to zeros for the first iteration.

## Update Magnetic Vector

If magnetic jamming was not detected in the current iteration, the magnetic vector estimate,  $m$ , is updated using the *a posteriori* magnetic disturbance error and the *a posteriori* orientation.

The magnetic disturbance error is converted to the navigation frame:

$$mErrorNED_{3 \times 3} = \left( (rPost_{3 \times 3})^T (mError_{1 \times 3})^T \right)^T$$

The magnetic disturbance error in the navigation frame is subtracted from the previous magnetic vector estimate and then interpreted as inclination:

$$M = m - mErrorNED$$

$$inclination = \text{atan2}(M(3), M(1))$$

The inclination is converted to a constrained magnetic vector estimate for the next iteration:

$$m(1) = (\text{ExpectedMagneticFieldStrength})(\cos(inclination))$$

$$m(2) = 0$$

$$m(3) = (\text{ExpectedMagneticFieldStrength})(\sin(inclination))$$

## References

- [1] Open Source Sensor Fusion. <https://github.com/memsindustrygroup/Open-Source-Sensor-Fusion/tree/master/docs>
- [2] Roetenberg, D., H.J. Luinge, C.T.M. Baten, and P.H. Veltink. "Compensation of Magnetic Disturbances Improves Inertial and Magnetic Sensing of Human Body Segment Orientation." *IEEE Transactions on Neural Systems and Rehabilitation Engineering*. Vol. 13. Issue 3, 2005, pp. 395-405.

## Extended Capabilities

### C/C++ Code Generation

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

### See Also

`ahrsfilter` | `ecompass` | `gpsSensor` | `imuSensor` | `imufilter` | `quaternion`

### Topics

"Determine Orientation Using Inertial Sensors"

### Introduced in R2020a

# Global Nearest Neighbor Multi Object Tracker

Multi-sensor, multi-object tracker using GNN assignment

**Library:** Sensor Fusion and Tracking Toolbox



## Description

The Global Nearest Neighbor Multi Object Tracker block is capable of processing detections of many targets from multiple sensors, much like the `trackerGNN` System object. The tracker initializes, confirms, predicts, corrects, and deletes tracks based on a global nearest neighbor (GNN) assignment algorithm. The tracker estimates the state vector and state vector covariance matrix for each track. Each detection is assigned to at most one track. If the detection cannot be assigned to any track, the tracker initializes a new track.

Any new track starts in a tentative state. If enough detections are assigned to a tentative track, its status changes to confirmed. If the detection already has a known classification (the `ObjectClassID` field of the returned track is nonzero), that track is confirmed immediately. When a track is confirmed, the tracker considers the track to represent a physical object. If detections are not assigned to the track within a specifiable number of updates, the track is deleted.

## Ports

### Input

#### Detections — Detection list

Simulink bus containing MATLAB structure

Detection list, specified as a Simulink bus containing a MATLAB structure. The structure has the form:

Field	Description	Type
<code>NumDetections</code>	Number of detections	integer
<code>Detections</code>	Object detections	Array of object detection structures. The first <code>NumDetections</code> of these detections are actual detections.

The fields of `Detections` are:

Field	Description	Type
<code>Time</code>	Measurement time	<code>single</code> or <code>double</code>
<code>Measurement</code>	Object measurements	<code>single</code> or <code>double</code>
<code>MeasurementNoise</code>	Measurement noise covariance matrix	<code>single</code> or <code>double</code>

Field	Description	Type
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

See [objectDetection](#) for more detailed explanations of these fields.

**Note** The object detection structure contains a **Time** field. The time tag of each object detection must be less than or equal to the time of the current invocation of the block. The time tag must also be greater than the update time specified in the previous invocation of the block.

#### Prediction Time — Track update time

real scalar

Track update time, specified as a real scalar in seconds. The tracker updates all tracks to this time. The update time must always increase with each invocation of the block. Units are in seconds. The update time must be at least as large as the largest **Time** specified at the **Detections** input port.

If this port is not enabled, the simulation clock managed by Simulink determines the update time.

#### Dependencies

To enable this port, in the **Port Setting** tab, set **Prediction time source** to **Input port**.

#### Cost Matrix — Cost matrix

real-valued  $N_t$ -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 at the **All Tracks** output port on the previous invocation of the block.

In the first update to the tracker, or if the track 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**.

If this port is not enabled, the filter initialized by the **Filter initialization function** calculates the cost matrix using the distance method.

#### Dependencies

To enable this port, in the **Port Setting** tab, select **Enable cost matrix input**.

#### Detectable TrackIDs — 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 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 **Probability of detection used for track score** parameter.

Tracks whose identifiers are not included in **Detectable TrackIDs** are considered undetectable. The track deletion logic does not count the lack of detection as a "missed detection" for track deletion purposes.

If this port is not enabled, the tracker assumes all tracks to be detectable at each invocation of the block.

#### Dependencies

To enable this port, in the **Port Setting** tab, select **Enable detectable track IDs Input**.

#### Output

##### **Confirmed Tracks — Confirmed tracks**

Simulink bus containing MATLAB structure

Confirmed tracks, returned as a Simulink bus containing a MATLAB structure. The structure has the form:

Field	Description
NumTracks	Number of tracks
Tracks	Array of track structures of a length set by the <b>Maximum number of tracks</b> parameter. Only the first NumTracks of these are actual tracks.

The fields of the track structure are shown in "Track Structure" on page 4-26.

Depending on the track logic, a track is confirmed if:

- History - A track receives at least  $M$  detections in the last  $N$  updates.  $M$  and  $N$  are specified in **Confirmation threshold** for the **History** logic.
- Score - The track score is at least as high as the confirmation threshold specified in **Confirmation threshold** for the **Score** logic.

##### **Tentative Tracks — Tentative tracks**

Simulink bus containing MATLAB structure

Tentative tracks, returned as a Simulink bus containing a MATLAB structure. A track is tentative before it is confirmed.

The fields of the track structure are shown in "Track Structure" on page 4-26.

#### Dependencies

To enable this port, in the **Port Setting** tab, select **Enable tentative tracks output**.

##### **All Tracks — Confirmed and Tentative tracks**

Simulink bus containing MATLAB structure

Combined list of confirmed and tentative tracks, returned as a Simulink bus containing a MATLAB structure.

The fields of the track structure are shown in “Track Structure” on page 4-26.

### Dependencies

To enable this port, in the **Port Setting** tab, select **Enable all tracks output**.

### Info — Additional information for analyzing track updates

Simulink bus containing MATLAB structure

Additional information for analyzing track updates, returned as a Simulink bus containing a MATLAB structure.

This table shows the fields of the info structure:

Field	Description
TrackIDsAtStepBeginning	Track IDs when step began
CostMatrix	Cost of assignment matrix
Assignments	Assignments returned from the assignment function
UnassignedTracks	IDs of unassigned tracks returned from the tracker
UnassignedDetections	IDs of unassigned detections returned from the tracker
InitiatedTrackIDs	IDs of tracks initiated during the step
DeletedTrackIDs	IDs of tracks deleted during the step
TrackIDsAtStepEnd	Track IDs when the step ended

### Dependencies

To enable this port, in the **Port Setting** tab, select **Enable information output**.

## Parameters

### Tracker Management

#### Filter initialization function — Filter initialization function

@initcvekf (default) | function handle | character vector

Filter initialization function, specified as a function handle or as a character vector containing the name of a valid filter initialization function. The tracker uses the filter initialization function when creating new tracks.

Sensor Fusion and Tracking Toolbox supplies many initialization functions that you can use:

Initialization Function	Function Definition
initcvabf	Initialize constant-velocity alpha-beta filter
initcaabf	Initialize constant-acceleration alpha-beta filter

Initialization Function	Function Definition
<code>initcvekf</code>	Initialize constant-velocity extended Kalman filter.
<code>initcackf</code>	Initialize constant-acceleration cubature filter.
<code>initctckf</code>	Initialize constant-turn-rate cubature filter.
<code>initcvckf</code>	Initialize constant-velocity cubature filter.
<code>initcapf</code>	Initialize constant-acceleration particle filter.
<code>initctpf</code>	Initialize constant-turn-rate particle filter.
<code>initcvpf</code>	Initialize constant-velocity particle 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-turn-rate extended Kalman filter.
<code>initctukf</code>	Initialize constant-turn-rate unscented Kalman filter.
<code>initcvmscekf</code>	Initialize constant-velocity modified spherical coordinates extended Kalman filter.
<code>initrpekf</code>	Initialize constant-velocity range-parametrized extended Kalman filter.
<code>initapekf</code>	Initialize constant-velocity angle-parametrized extended Kalman filter.
<code>initekfimm</code>	Initialize tracking IMM filter.

You can also write your own initialization function. The function must have this syntax:

```
filter = filterInitializationFcn(detection)
```

The input to this function is a detection report like those created by `objectDetection`. The output of this function must be a filter object: `trackingKF`, `trackingEKF`, `trackingUKF`, `trackingCKF`, `trackingPF`, `trackingMSCEKF`, `trackingGSF`, `trackingIMM`, 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
```

#### **Assignment algorithm name — Assignment algorithm name**

```
'MatchPairs' (default) | 'Munkres' | 'Jonker-Volgenant' | 'Auction' | 'Custom'
```

Assignment algorithm, specified as '`MatchPairs`', '`Munkres`', '`Jonker-Volgenant`', '`Auction`', or '`Custom`'. `Munkres` is the only assignment algorithm that guarantees an optimal

solution, but it is also the slowest, especially for large numbers of detections and tracks. The other algorithms do not guarantee an optimal solution but can be faster for problems with 20 or more tracks and detections. Use 'Custom' to define your own assignment function and specify its name in the **CustomAssignmentFcn** property.

**Name of 'Custom' assignment function — Custom assignment function name**  
character vector

Custom assignment function name, specified as a character string. An assignment function must have this syntax:

```
[assignment,unTrs,unDets] = f(cost,costNonAssignment)
```

For an example of an assignment function and a description of its arguments, see [assignmunkres](#).

Example: 'mycustomfcn'

#### Dependencies

To enable this property, set the **Assignment algorithm name** name to 'Custom'.

**Threshold for assigning detections to tracks — Threshold for assigning detections to tracks**

30\*[1 Inf] (default) | positive scalar | 1-by-2 vector of positive values

Threshold for assigning detections to tracks (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, **trackingCKF** and **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).

**Maximum number of tracks — Maximum number of tracks**

200 (default) | positive integer

Maximum number of tracks that the block can maintain, specified as a positive integer.

**Maximum number of sensors — Maximum number of sensors**

20 (default) | positive integer

Maximum number of sensors that the block can process, specified as a positive integer. This value should be greater than or equal to the highest **SensorIndex** value input at the **Detections** input port.

**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 code. The first time you run a simulation, Simulink generates 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.

**Track Logic****Type of track confirmation and deletion logic — Confirmation and deletion logic type**

History (default) | Score

Confirmation and deletion logic type, selected as History or Score.

- **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.

**Confirmation threshold [M N] — Track confirmation threshold for history logic**

[2 3] (default) | real-valued 1-by-2 vector of positive integers

Track confirmation threshold for history logic, specified as a real-valued 1-by-2 vector of positive integers [M N]. A track is confirmed if it receives at least M detections in the last N updates.

**Dependencies**To enable this parameter, set **Type of track confirmation and deletion logic** to History.**Deletion threshold [P Q] — Track deletion threshold for history logic**

[5 5] (default) | real-valued 1-by-2 vector of positive integers

Track deletion threshold for history logic, specified as a real-valued 1-by-2 vector of positive integers [P Q]. If a confirmed track is not assigned to any detection P times in the last Q tracker updates, then the track is deleted.

**Dependencies**To enable this parameter, set **Type of track confirmation and deletion logic** to History.**Confirmation threshold [positive scalar] — Track confirmation threshold for score logic**

20 (default) | positive scalar

Track confirmation threshold for score logic, specified as a real-valued positive scalar. A track is confirmed if its score is at least as high as the confirmation threshold.

**Dependencies**To enable this parameter, set **Type of track confirmation and deletion logic** to Score.

**Deletion threshold [negative scalar] — Track deletion threshold for score logic**  
-7 (default) | scalar | negative scalar

Track deletion threshold for score logic, specified as a negative scalar. A track is deleted if its score decreases by at least the threshold from the maximum track score.

#### Dependencies

To enable this parameter, set **Type of track confirmation and deletion logic** to Score.

**Probability of detection used for track score — Probability of detection used for track score**

0.9 (default) | scalar in (0,1)

Probability of detection used for track score, specified as a positive scalar in (0,1).

Example: 0.5

#### Dependencies

To enable this parameter, set **Type of track confirmation and deletion logic** to Score.

**Rate of false positives used for track score — Probability of false alarm used for track score**

1e-6 (default) | scalar in (0,1)

The probability of false alarm used for track score, specified as a scalar in (0,1).

Example: 1e-5

#### Dependencies

To enable this parameter, set **Type of track confirmation and deletion logic** to Score.

**Volume of the sensor's detection bin — Volume of sensor detection bin**

1 (default) | positive scalar

The volume of a sensor detection bin, specified as a positive scalar. For example, if a radar produces a 4-D measurement, which includes azimuth, elevation, range, and range rate, the 4-D volume is defined by the radar angular beam width, the range bin width, and the range-rate bin width. Volume is used in calculating the track score when initializing and updating a track.

Example: 1.5

#### Dependencies

To enable this parameter, set **Type of track confirmation and deletion logic** to Score.

**Rate of new tracks per unit volume — Rate of new tracks per unit volume**

1 (default) | positive scalar

The rate of new tracks per unit volume, specified as a positive scalar. The rate of new tracks is used in calculating the track score during track initialization.

Example: 2.5

#### Dependencies

To enable this parameter, set **Type of track confirmation and deletion logic** to Score.

**Port Setting****Prediction time source — Source of prediction time**

Auto (default) | Input port

Source for prediction time, specified as **Input port** or **Auto**. Select **Input port** to input an update time by using the **Prediction Time** input port. Otherwise, the simulation clock managed by Simulink determines the update time.

**Enable cost matrix input — Enable input port for cost matrix**

off (default) | on

Select this check box to enable the input of a cost matrix by using the **Cost Matrix** input port.

**Enable detectable track IDs input — Enable detectable track IDs input**

off (default) | on

Select this check box to enable the **Detectable track IDs** input port.

**Enable tentative tracks output — Enable output port for tentative tracks**

off (default) | on

Select this check box to enable the output of tentative tracks through the **Tentative Tracks** output port.

**Enable all tracks output — Enable output port for all tracks**

off (default) | on

Select this check box to enable the output of all the tracks through the **All Tracks** output port.

**Enable information output — Enable output port for analysis information**

off (default) | on

Select this check box to enable the output port for analysis information through the **Info** output port.

**Source of output bus name — Source of output track bus name**

Auto (default) | Property

Source of the output track bus name, specified as:

- **Auto** — The block automatically creates an output track bus name.
- **Property** — Specify the output track bus name by using the **Specify an output bus name** parameter.

**Source of output info bus name — Source of output info bus name**

Auto (default) | Property

Source of the output info bus name, specified as one of these options:

- **Auto** — The block automatically creates an output info bus name.
- **Property** — Specify the output info bus name by using the **Specify an output bus name** parameter.

## Algorithms

### Tracker Logic Flow

When a GNN tracker processes detections, track creation and management follow these steps:

- 1 The tracker divides detections by originating sensor.
- 2 For each sensor:
  - a The tracker calculates the distances from detections to existing tracks and forms a cost matrix.
  - b Based on the costs, the tracker performs global nearest neighbor assignment using the algorithm specified by the **Assignment algorithm name** parameter.
  - c The assignment algorithm divides the detections and tracks into three groups:
    - Assigned one-to-one detection and track pairs
    - Unassigned detections
    - Unassigned tracks
- 3 Unassigned detections initialize new tracks. Using the unassigned detection, the tracker initializes a new track filter specified by the **Filter initialization function** parameter. The track logic for the new track is initialized as well.
 

The tracker checks if any of the unassigned detections from other sensors can be assigned to the new track. If so, the tracker updates the new track with the assigned detections from the other sensors. As a result, these detections no longer initialize new tracks.
- 4 The pairs of assigned tracks and detections are used to update each track. The track filter is updated using the **correct** method provided by the specified tracking filter. Also, the track logic is updated with a "hit". The tracker checks if the track meets the criteria for confirmation. If so, the tracker confirms the track and sets the **IsCoasted** field to **false**.
- 5 Unassigned tracks are updated with a "miss" and their **IsCoasted** field is set to **true**. The tracker checks if the track meets the criteria for deletion. If so, the tracker removes the track from the maintained track list.
- 6 All tracks are predicted to the latest time value (either the time provided by the **Prediction Time** input port, or the time determined by Simulink).

### Track Structure

The fields of the track structure are:

Field	Definition
SourceIndex	Unique source index used to distinguish tracking sources in a multiple tracker environment.
TrackID	Unique track identifier used to distinguish multiple tracks.
BranchID	Unique track branch identifier used to distinguish multiple track branches.
UpdateTime	Time at which the track is updated. Units are in seconds.

Field	Definition
Age	Number of times the track survived.
State	Value of state vector at the update time.
StateCovariance	Uncertainty covariance matrix.
TrackLogic	Confirmation and deletion logic type, returned as 'History' or 'Score'.
TrackLogicState	The current state of the track logic type. Based on the logic type TrackLogic, the logic state is returned as: <ul style="list-style-type: none"> <li>'History' - A 1-by-<math>K</math> logical array, where <math>K</math> is the number of latest track logical states recorded. In the array, 1 denotes hit and 0 denote miss.</li> <li>'Score' - A 1-by-2 array of real scalars, [cs, ms]. cs is the current score, and ms is the maximum score.</li> </ul>
IsConfirmed	Confirmation status. This field is true if the track is confirmed to be a real target.
IsCoasted	Coasting status. This field is true 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 true by default.
ObjectClassID	Integer value representing the object classification. The value 0 represents an unknown classification. Nonzero classifications apply only to confirmed tracks.
ObjectAttributes	Additional information of the track.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Blocks

Joint Probabilistic Data Association Multi Object Tracker

### Functions

`assignauction | assignjv | assignkbest | assignkbestsd | assignmunkres | assignsd | fusecovint | fusecovunion | fusexcov | getTrackPositions | getTrackVelocities`

### Objects

`objectDetection | objectTrack | trackHistoryLogic | trackScoreLogic | trackingABF | trackingCKF | trackingEKF | trackingGSF | trackingIMM | trackingKF | trackingMSCEKF | trackingPF | trackingUKF`

**System Objects**

[trackerGNN](#) | [trackerJPDA](#) | [trackerTOMHT](#)

**Topics**

“Introduction to Multiple Target Tracking”

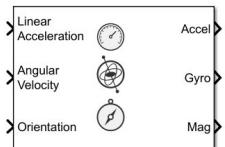
“Introduction to Assignment Methods in Tracking Systems”

**Introduced in R2019b**

# IMU

IMU simulation model

**Library:** Navigation Toolbox / Multisensor Positioning / Sensor Models  
 Sensor Fusion and Tracking Toolbox / Multisensor Positioning / Sensor Models



## Description

The IMU Simulink block models receiving data from an inertial measurement unit (IMU) composed of accelerometer, gyroscope, and magnetometer sensors.

## Ports

### Input

**Linear Acceleration — Acceleration of IMU in local navigation coordinate system (m/s<sup>2</sup>)**  
 $N$ -by-3 matrix of real scalar

Acceleration of the IMU in the local navigation coordinate system, specified as an  $N$ -by-3 matrix of real scalars in meters per second squared.  $N$  is the number of samples in the current frame.

Data Types: `single` | `double`

**Angular Velocity — Angular velocity of IMU in local navigation coordinate system (rad/s)**  
 $N$ -by-3 matrix of real scalar

Angular velocity of the IMU sensor body frame in the local navigation coordinate system, specified as an  $N$ -by-3 matrix of scalars in radians per second.  $N$  is the number of samples in the current frame.

Data Types: `single` | `double`

**Orientation — Orientation of IMU in local navigation coordinate system**  
 $N$ -by-4 array of real scalar | 3-by-3-by- $N$ -element rotation matrix

Orientation of the IMU sensor body frame with respect to the local navigation coordinate system, specified as an  $N$ -by-4 array of real scalars or a 3-by-3-by- $N$  rotation matrix. Each row of the  $N$ -by-4 array is assumed to be the four elements of a quaternion.  $N$  is the number of samples in the current frame.

Data Types: `single` | `double`

### Output

**Accel — Accelerometer measurement of IMU in sensor body coordinate system (m/s<sup>2</sup>)**  
 $N$ -by-3 matrix of real scalar

Accelerometer measurement of the IMU in the sensor body coordinate system, returned as an  $N$ -by-3 matrix of real scalars in meters per second squared.  $N$  is the number of samples in the current frame.

Data Types: single | double

**Gyro — Gyroscope measurement of IMU in sensor body coordinate system (rad/s)**

$N$ -by-3 matrix of real scalar

Gyroscope measurement of the IMU in the sensor body coordinate system, returned as an  $N$ -by-3 matrix of real scalars in radians per second.  $N$  is the number of samples in the current frame.

Data Types: single | double

**Mag — Magnetometer measurement of IMU in sensor body coordinate system ( $\mu$ T)**

$N$ -by-3 matrix of real scalar

Magnetometer measurement of the IMU in the sensor body coordinate system, returned as an  $N$ -by-3 matrix of real scalars in microtesla.  $N$  is the number of samples in the current frame.

Data Types: single | double

## Parameters

### Parameters

**Reference frame — Navigation reference frame**

NED (default) | ENU

Navigation reference frame, specified as NED (North-East-Down) or ENU (East-North-Up).

**Temperature ( $^{\circ}$ C) — Operating temperature of IMU ( $^{\circ}$ C)**

25 (default) | real scalar

Operating temperature of the IMU in degrees Celsius, specified as a real scalar.

When the block calculates temperature scale factors and environmental drift noises, 25  $^{\circ}$ C is used as the nominal temperature.

Data Types: single | double

**Magnetic field (NED) — Magnetic field vector expressed in NED navigation frame ( $\mu$ T)**

[27.5550, -2.4169, -16.0849] (default) | 1-by-3 vector of scalar

Magnetic field vector expressed in the NED navigation frame, specified as a 1-by-3 vector of scalars.

The default magnetic field corresponds to the magnetic field at latitude zero, longitude zero, and altitude zero.

### Dependencies

To enable this parameter, set **Reference frame** to NED.

Data Types: single | double

**MagneticField (ENU) — Magnetic field vector expressed in ENU navigation frame ( $\mu$ T)**

[-2.4169, 27.5550, 16.0849] (default) | 1-by-3 vector of scalar

Magnetic field vector expressed in the ENU navigation frame, specified as a 1-by-3 vector of scalars.

The default magnetic field corresponds to the magnetic field at latitude zero, longitude zero, and altitude zero.

### **Dependencies**

To enable this parameter, set **Reference frame** to ENU.

Data Types: single | double

### **Seed — Initial seed for randomization**

67 (default) | nonnegative integer

Initial seed of a random number generator algorithm, specified as a nonnegative integer.

Data Types: single | double

### **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 code. The first time that you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations if the model does not change. This option requires additional startup time.

### **Accelerometer**

#### **Maximum readings ( $\text{m/s}^2$ ) — Maximum sensor reading ( $\text{m/s}^2$ )**

inf (default) | real positive scalar

Maximum sensor reading in  $\text{m/s}^2$ , specified as a real positive scalar.

Data Types: single | double

#### **Resolution ( $(\text{m/s}^2)/\text{LSB}$ ) — Resolution of sensor measurements ( $(\text{m/s}^2)/\text{LSB}$ )**

0 (default) | real nonnegative scalar

Resolution of sensor measurements in  $(\text{m/s}^2)/\text{LSB}$ , specified as a real nonnegative scalar.

Data Types: single | double

#### **Constant offset bias ( $\text{m/s}^2$ ) — Constant sensor offset bias ( $\text{m/s}^2$ )**

[0 0 0] (default) | real scalar | real 3-element row vector

Constant sensor offset bias in  $\text{m/s}^2$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

#### **Axis skew (%) — Sensor axes skew (%)**

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Sensor axes skew in a percentage, specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Velocity random walk ( $\text{m/s}^2/\sqrt{\text{Hz}}$ ) — Velocity random walk ( $\text{m/s}^2/\sqrt{\text{Hz}}$ )**

[0 0 0] (default) | real scalar | real 3-element row vector

Velocity random walk in ( $\text{m/s}^2/\sqrt{\text{Hz}}$ ), specified as a real scalar or 3-element row vector. This property corresponds to the power spectral density of sensor noise. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Bias Instability ( $\text{m/s}^2$ ) — Instability of the bias offset ( $\text{m/s}^2$ )**

[0 0 0] (default) | real scalar | real 3-element row vector

Instability of the bias offset in  $\text{m/s}^2$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Acceleration random walk ( $(\text{m/s}^2)(\sqrt{\text{Hz}})$ ) — Acceleration random walk ( $(\text{m/s}^2)(\sqrt{\text{Hz}})$ )**

[0 0 0] (default) | real scalar | real 3-element row vector

Acceleration random walk of sensor in  $(\text{m/s}^2)(\sqrt{\text{Hz}})$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Bias from temperature ( $(\text{m/s}^2)/^\circ\text{C}$ ) — Sensor bias from temperature ( $(\text{m/s}^2)/^\circ\text{C}$ )**

[0 0 0] (default) | real scalar | real 3-element row vector

Sensor bias from temperature in  $(\text{m/s}^2)/^\circ\text{C}$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Temperature scale factor ( $%/^\circ\text{C}$ ) — Scale factor error from temperature ( $%/^\circ\text{C}$ )**

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Scale factor error from temperature in  $%/^\circ\text{C}$ , specified as a real scalar or real 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Gyroscope****Maximum readings (rad/s) — Maximum sensor reading (rad/s)**

inf (default) | real positive scalar

Maximum sensor reading in rad/s, specified as a real positive scalar.

Data Types: single | double

**Resolution ((rad/s)/LSB) — Resolution of sensor measurements ((rad/s)/LSB)**

0 (default) | real nonnegative scalar

Resolution of sensor measurements in (rad/s)/LSB, specified as a real nonnegative scalar.

Data Types: single | double

**Constant offset bias (rad/s) — Constant sensor offset bias (rad/s)**

[0 0 0] (default) | real scalar | real 3-element row vector

Constant sensor offset bias in rad/s, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Axis skew (%) — Sensor axes skew (%)**

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Sensor axes skew in a percentage, specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Bias from acceleration ((rad/s)/(m/s<sup>2</sup>) — Sensor bias from linear acceleration (rad/s)/(m/s<sup>2</sup>)**

[0 0 0] (default) | real scalar | real 3-element row vector

Sensor bias from linear acceleration in (rad/s)/(m/s<sup>2</sup>), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Angle random walk ((rad/s)/(√Hz)) — Acceleration random walk ((rad/s)/(√Hz))**

[0 0 0] (default) | real scalar | real 3-element row vector

Acceleration random walk of sensor in (rad/s)/(√Hz), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Bias Instability (rad/s) — Instability of the bias offset (rad/s)**

[0 0 0] (default) | real scalar | real 3-element row vector

Instability of the bias offset in rad/s, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Rate random walk ((rad/s)/(√Hz)) — Integrated white noise of sensor ((rad/s)(√Hz))**

[0 0 0] (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in (rad/s)(√Hz), specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Bias from temperature ((rad/s)/°C) — Sensor bias from temperature ((rad/s)/°C)**

[0 0 0] (default) | real scalar | real 3-element row vector

Sensor bias from temperature in (rad/s)/°C, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Temperature scale factor (%/°C) — Scale factor error from temperature (%/°C)**

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Scale factor error from temperature in %/°C, specified as a real scalar or real 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Magnetometer**

**Maximum readings (µT) — Maximum sensor reading (µT)**

inf (default) | real positive scalar

Maximum sensor reading in µT, specified as a real positive scalar.

Data Types: single | double

**Resolution ((µT)/LSB) — Resolution of sensor measurements ((µT)/LSB)**

0 (default) | real nonnegative scalar

Resolution of sensor measurements in (µT)/LSB, specified as a real nonnegative scalar.

Data Types: single | double

**Constant offset bias (µT) — Constant sensor offset bias (µT)**

[0 0 0] (default) | real scalar | real 3-element row vector

Constant sensor offset bias in µT, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Axis skew (%) — Sensor axes skew (%)**

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

Sensor axes skew in a percentage, specified as a real scalar or 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**White noise PSD (µT/√Hz) — Power spectral density of sensor noise (µT/√Hz)**

[0 0 0] (default) | real scalar | real 3-element row vector

Power spectral density of sensor noise in µT/√Hz, specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Bias Instability (µT) — Instability of the bias offset (µT)**

[0 0 0] (default) | real scalar | real 3-element row vector

Instability of the bias offset in  $\mu\text{T}$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Random walk ( $(\mu\text{T})^*\sqrt{\text{Hz}}$ ) — Integrated white noise of sensor ( $(\mu\text{T})^*\sqrt{\text{Hz}}$ )**

[0 0 0] (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in  $(\mu\text{T})^*\sqrt{\text{Hz}}$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Bias from temperature ( $\mu\text{T}/^\circ\text{C}$ ) — Sensor bias from temperature ( $\mu\text{T}/^\circ\text{C}$ )**

[0 0 0] (default) | real scalar | real 3-element row vector

Sensor bias from temperature in  $\mu\text{T}/^\circ\text{C}$ , specified as a real scalar or 3-element row vector. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

**Temperature scale factor (%/ $^\circ\text{C}$ ) — Scale factor error from temperature (%/ $^\circ\text{C}$ )**

[0 0 0] (default) | real scalar in the range [0,100] | real 3-element row vector in the range [0,100]

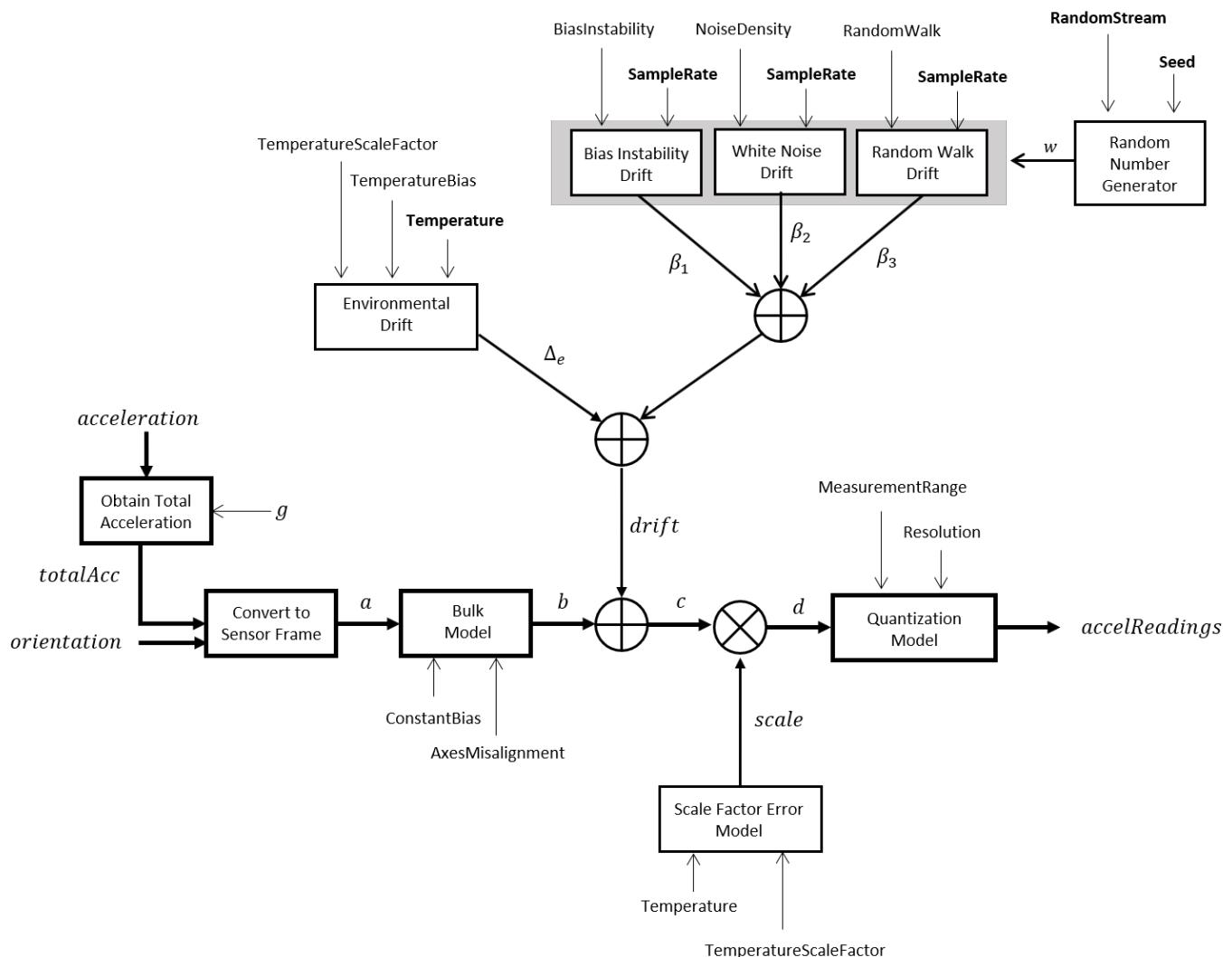
Scale factor error from temperature in %/ $^\circ\text{C}$ , specified as a real scalar or real 3-element row vector with values ranging from 0 to 100. Any scalar input is converted into a real 3-element row vector where each element has the input scalar value.

Data Types: single | double

## Algorithms

### Accelerometer

The accelerometer model uses the ground-truth orientation and acceleration inputs and the `imuSensor` and `accelparams` properties to model accelerometer readings.



### Obtain Total Acceleration

To obtain the total acceleration (*totalAcc*), the acceleration is preprocessed by negating and adding the gravity constant vector ( $g = [0; 0; 9.8] \text{ m/s}^2$ ) as:

$$\text{totalAcc} = -\text{acceleration} + g$$

### Convert to Sensor Frame

Then the total acceleration is converted from the local navigation frame to the sensor frame using:

$$a = (\text{orientation})(\text{totalAcc})^T$$

If the orientation is input in quaternion form, it is converted to a rotation matrix before processing.

### Bulk Model

The ground-truth acceleration in the sensor frame, *a*, passes through the bulk model, which adds axes misalignment and bias:

$$b = \begin{pmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{pmatrix} (a^T)^T + \text{ConstantBias}$$

where ConstantBias is a property of `accelparams`, and  $\alpha_1$ ,  $\alpha_2$ , and  $\alpha_3$  are given by the first, second, and third elements of the AxesMisalignment property of `accelparams`.

### Bias Instability Drift

The bias instability drift is modeled as white noise biased and then filtered:

$$\beta_1 = h_1 * (w)(\text{BiasInstability})$$

where BiasInstability is a property of `accelparams`, and  $h_1$  is a filter defined by the SampleRate property:

$$H_1(z) = \frac{1}{1 - \frac{1}{2}z^{-1}}$$

### White Noise Drift

White noise drift is modeled by multiplying elements of the white noise random stream by the standard deviation:

$$\beta_2 = (w)\left(\sqrt{\frac{\text{SampleRate}}{2}}\right)(\text{NoiseDensity})$$

where SampleRate is an `imuSensor` property, and NoiseDensity is an `accelparams` property. Elements of  $w$  are random numbers given by settings of the `imuSensor` random stream.

### Random Walk Drift

The random walk drift is modeled by biasing elements of the white noise random stream and then filtering:

$$\beta_3 = h_2 * (w)\left(\frac{\text{RandomWalk}}{\sqrt{\frac{\text{SampleRate}}{2}}}\right)$$

where RandomWalk is a property of `accelparams`, SampleRate is a property of `imuSensor`, and  $h_2$  is a filter defined as:

$$H_2(z) = \frac{1}{1 - z^{-1}}$$

### Environmental Drift Noise

The environmental drift noise is modeled by multiplying the temperature difference from a standard with the temperature bias:

$$\Delta_e = (\text{Temperature} - 25)(\text{TemperatureBias})$$

where Temperature is a property of `imuSensor`, and TemperatureBias is a property of `acelparams`. The constant 25 corresponds to a standard temperature.

### Scale Factor Error Model

The temperature scale factor error is modeled as:

$$scaleFactorError = 1 + \left( \frac{\text{Temperature} - 25}{100} \right) (\text{TemperatureScaleFactor})$$

where Temperature is a property of `imuSensor`, and TemperatureScaleFactor is a property of `acelparams`. The constant 25 corresponds to a standard temperature.

### Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

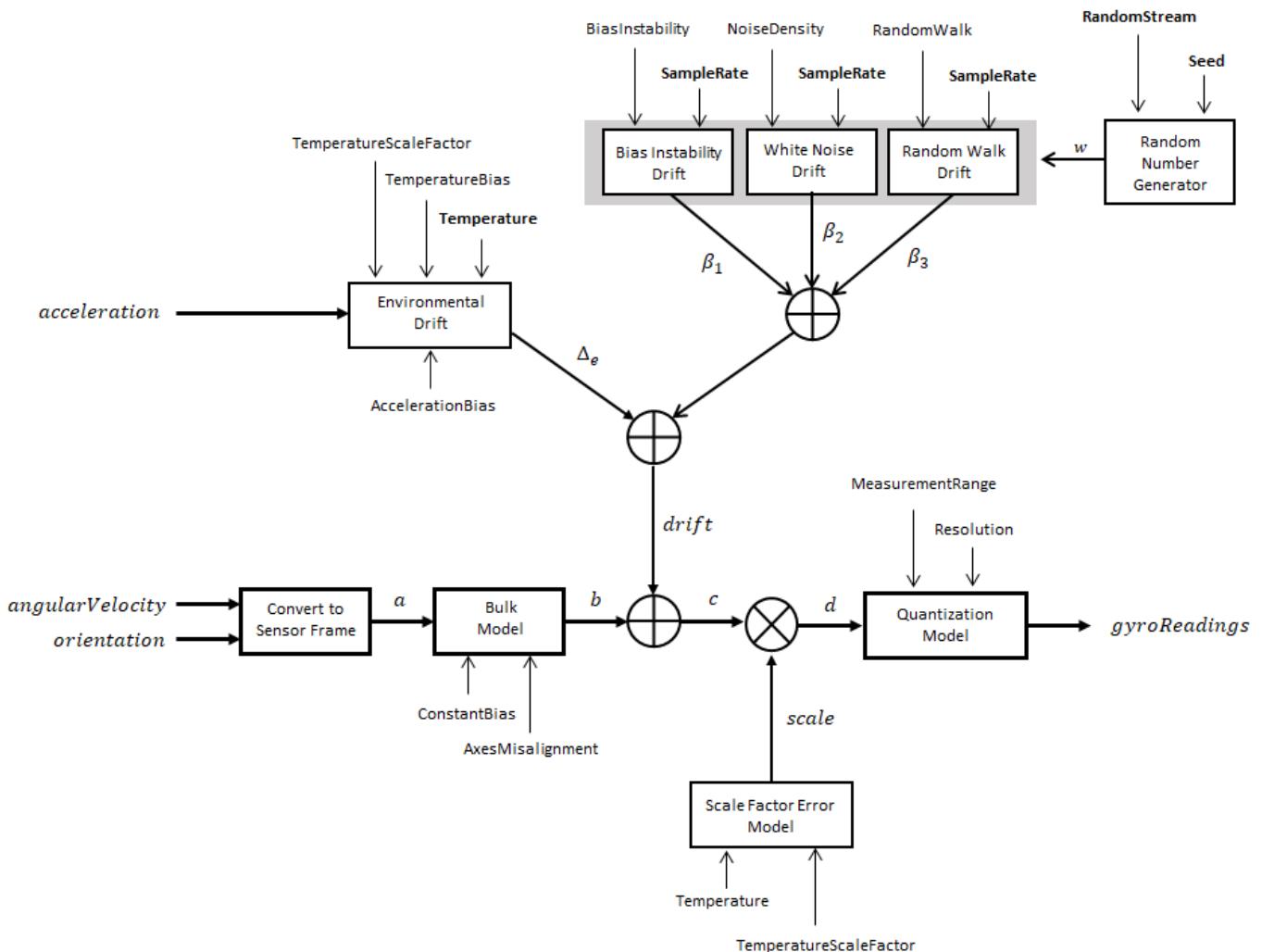
and then setting the resolution:

$$\text{accelReadings} = (\text{Resolution}) \left( \text{round} \left( \frac{e}{\text{Resolution}} \right) \right)$$

where MeasurementRange is a property of `acelparams`.

### Gyroscope

The gyroscope model uses the ground-truth orientation, acceleration, and angular velocity inputs, and the `imuSensor` and `gyroparms` properties to model accelerometer readings.



### Convert to Sensor Frame

The ground-truth angular velocity is converted from the local frame to the sensor frame using the ground-truth orientation:

$$a = (\text{orientation})(\text{angularVelocity})^T$$

If the orientation is input in quaternion form, it is converted to a rotation matrix before processing.

### Bulk Model

The ground-truth angular velocity in the sensor frame, *a*, passes through the bulk model, which adds axes misalignment and bias:

$$b = \begin{pmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{pmatrix} (a^T)^T + \text{ConstantBias}$$

where ConstantBias is a property of `gyroparams`, and  $\alpha_1$ ,  $\alpha_2$ , and  $\alpha_3$  are given by the first, second, and third elements of the AxesMisalignment property of `gyroparams`.

### Bias Instability Drift

The bias instability drift is modeled as white noise biased and then filtered:

$$\beta_1 = h_1 * (w)(\text{BiasInstability})$$

where BiasInstability is a property of `gyroparams` and  $h_1$  is a filter defined by the SampleRate property:

$$H_1(z) = \frac{1}{1 - \frac{1}{2}z^{-1}}$$

### White Noise Drift

White noise drift is modeled by multiplying elements of the white noise random stream by the standard deviation:

$$\beta_2 = (w)\left(\sqrt{\frac{\text{SampleRate}}{2}}\right)(\text{NoiseDensity})$$

where SampleRate is an `imuSensor` property, and NoiseDensity is an `gyroparams` property. The elements of  $w$  are random numbers given by settings of the `imuSensor` random stream.

### Random Walk Drift

The random walk drift is modeled by biasing elements of the white noise random stream and then filtering:

$$\beta_3 = h_2 * (w)\left(\frac{\text{RandomWalk}}{\sqrt{\frac{\text{SampleRate}}{2}}}\right)$$

where RandomWalk is a property of `gyroparams`, SampleRate is a property of `imuSensor`, and  $h_2$  is a filter defined as:

$$H_2(z) = \frac{1}{1 - z^{-1}}$$

### Environmental Drift Noise

The environmental drift noise is modeled by multiplying the temperature difference from a standard with the temperature bias:

$$\Delta_e = (\text{Temperature} - 25)(\text{TemperatureBias})$$

where Temperature is a property of `imuSensor`, and TemperatureBias is a property of `gyroparams`. The constant 25 corresponds to a standard temperature.

### Scale Factor Error Model

The temperature scale factor error is modeled as:

$$scaleFactorError = 1 + \left( \frac{\text{Temperature} - 25}{100} \right) (\text{TemperatureScaleFactor})$$

where Temperature is a property of `imuSensor`, and TemperatureScaleFactor is a property of `gyroparams`. The constant 25 corresponds to a standard temperature.

### Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

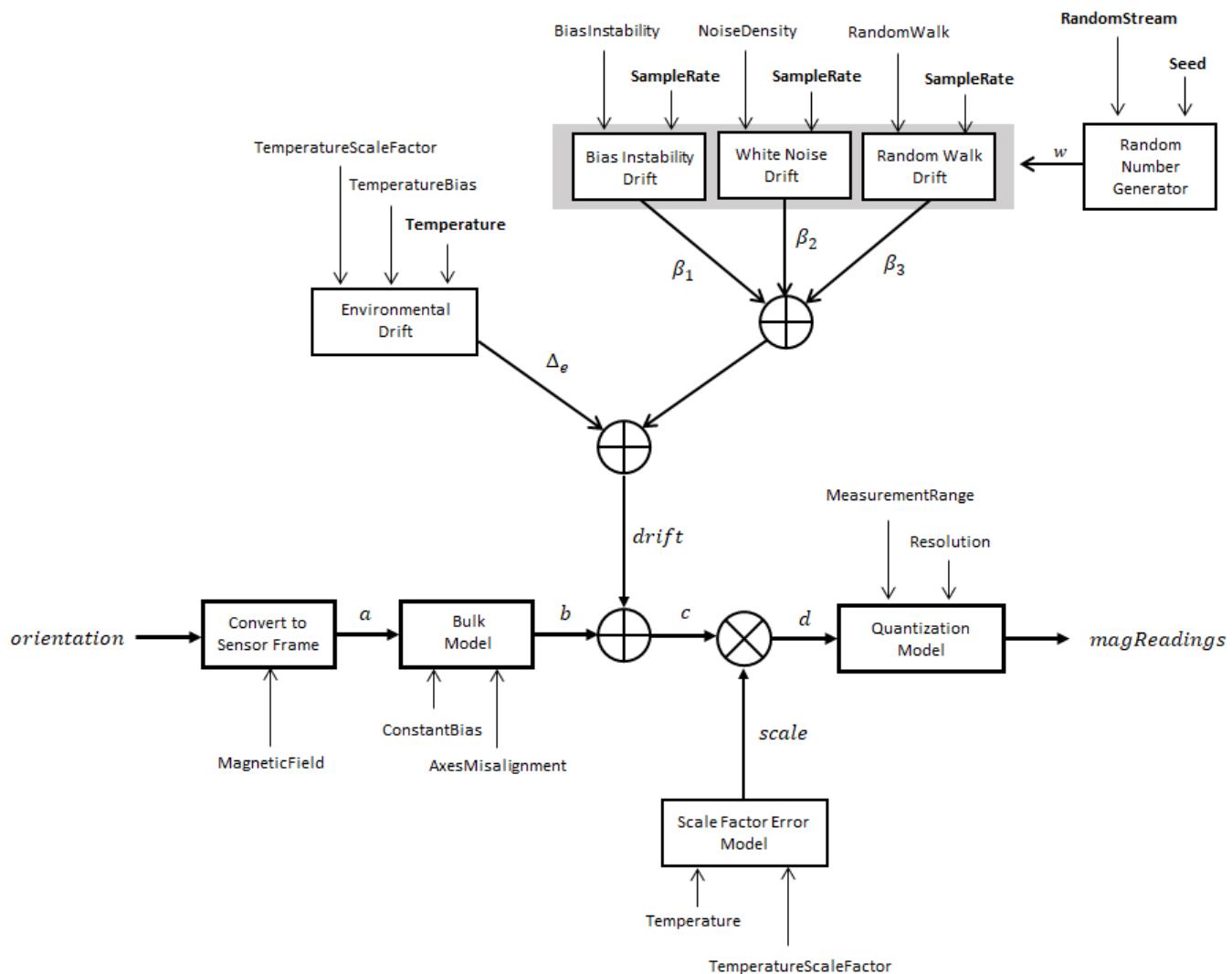
and then setting the resolution:

$$gyroReadings = (\text{Resolution}) \left( \text{round} \left( \frac{e}{\text{Resolution}} \right) \right)$$

where MeasurementRange is a property of `gyroparams`.

### Magnetometer

The magnetometer model uses the ground-truth orientation and acceleration inputs, and the `imuSensor` and `magparams` properties to model magnetometer readings.



### Convert to Sensor Frame

The ground-truth acceleration is converted from the local frame to the sensor frame using the ground-truth orientation:

$$a = (\text{orientation})(\text{totalAcc})^T$$

If the orientation is input in quaternion form, it is converted to a rotation matrix before processing.

### Bulk Model

The ground-truth acceleration in the sensor frame, *a*, passes through the bulk model, which adds axes misalignment and bias:

$$b = \begin{pmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{pmatrix} (a^T)^T + \text{ConstantBias}$$

where ConstantBias is a property of `magparams`, and  $\alpha_1$ ,  $\alpha_2$ , and  $\alpha_3$  are given by the first, second, and third elements of the AxesMisalignment property of `magparams`.

### Bias Instability Drift

The bias instability drift is modeled as white noise biased and then filtered:

$$\beta_1 = h_1 * (w)(\text{BiasInstability})$$

where BiasInstability is a property of `magparams` and  $h_1$  is a filter defined by the SampleRate property:

$$H_1(z) = \frac{1}{1 - \frac{1}{2}z^{-1}}$$

### White Noise Drift

White noise drift is modeled by multiplying elements of the white noise random stream by the standard deviation:

$$\beta_2 = (w)\left(\sqrt{\frac{\text{SampleRate}}{2}}\right)(\text{NoiseDensity})$$

where SampleRate is an `imuSensor` property, and NoiseDensity is an `magparams` property. The elements of  $w$  are random numbers given by settings of the `imuSensor` random stream.

### Random Walk Drift

The random walk drift is modeled by biasing elements of the white noise random stream and then filtering:

$$\beta_3 = h_2 * (w)\left(\frac{\text{RandomWalk}}{\sqrt{\frac{\text{SampleRate}}{2}}}\right)$$

where RandomWalk is a property of `magparams`, SampleRate is a property of `imuSensor`, and  $h_2$  is a filter defined as:

$$H_2(z) = \frac{1}{1 - z^{-1}}$$

### Environmental Drift Noise

The environmental drift noise is modeled by multiplying the temperature difference from a standard with the temperature bias:

$$\Delta_e = (\text{Temperature} - 25)(\text{TemperatureBias})$$

where Temperature is a property of `imuSensor`, and TemperatureBias is a property of `magparams`. The constant 25 corresponds to a standard temperature.

### Scale Factor Error Model

The temperature scale factor error is modeled as:

$$\text{scaleFactorError} = 1 + \left( \frac{\text{Temperature} - 25}{100} \right) (\text{TemperatureScaleFactor})$$

where Temperature is a property of `imuSensor`, and TemperatureScaleFactor is a property of `magparams`. The constant 25 corresponds to a standard temperature.

### Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

and then setting the resolution:

$$\text{magReadings} = (\text{Resolution}) \left( \text{round} \left( \frac{e}{\text{Resolution}} \right) \right)$$

where MeasurementRange is a property of `magparams`.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Classes

`accelparams` | `gyroparams` | `magparams`

### System Objects

`gpsSensor` | `imuSensor` | `insSensor`

### Topics

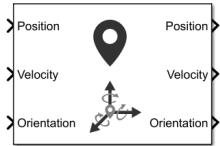
“Model IMU, GPS, and INS/GPS”

## Introduced in R2020a

# INS

Simulate INS sensor

**Library:** Navigation Toolbox / Multisensor Positioning / Sensor Models  
 Sensor Fusion and Tracking Toolbox / Multisensor Positioning / Sensor Models



## Description

The block simulates an INS sensor, which outputs noise-corrupted position, velocity, and orientation based on the corresponding inputs. To change the level of noise present in the output, you can vary the roll, pitch, yaw, position, and velocity accuracies . The accuracy is defined as the standard deviation of the noise.

## Ports

### Input

#### Position — Position of INS sensor

*N*-by-3 matrix of scalar

Position of the INS sensor relative to the navigation frame, specified as an *N*-by-3 matrix of scalars. *N* is the number of samples.

Data Types: `single` | `double`

#### Velocity — Velocity of INS sensor

*N*-by-3 matrix of scalar

Velocity of the INS sensor relative to the navigation frame, specified as an *N*-by-3 matrix of scalars. *N* is the number of samples.

Data Types: `single` | `double`

#### Orientation — Orientation of INS sensor

3-by-3-by-*N* array of scalar | *N*-by-4 matrix of scalar

Orientation of the INS sensor relative to the navigation frame, specified as either a 3-by-3-by-*N* array of scalars, where each page of the array (3-by-3 matrix) is a rotation matrix or as an *N*-by-4 matrix of scalars, where each row of the matrix is the four elements of a quaternion. *N* is the number of samples.

Data Types: `single` | `double`

## Output

### Position — Position of INS sensor

*N*-by-3 matrix of scalar

Position of the INS sensor relative to the navigation frame, returned as an *N*-by-3 matrix of scalars. *N* is the number of samples in the input.

Data Types: `single` | `double`

### Velocity — Velocity of INS sensor

*N*-by-3 matrix of scalar

Velocity of the INS sensor relative to the navigation frame, returned as an *N*-by-3 matrix of scalars. *N* is the number of samples in the inputs.

Data Types: `single` | `double`

### Orientation — Orientation of INS sensor

3-by-3-by-*N* array of scalar | *N*-by-4 matrix of scalar

Orientation of the INS sensor relative to the navigation frame, returned either as a 3-by-3-by-*N* array of scalars, where each page of the array (3-by-3 matrix) is a rotation matrix, or an *N*-by-4 matrix of scalars, where each row of the matrix is the four elements of a quaternion. The exact type of orientation output follows the type of orientation input. *N* is the number of samples in the input.

Data Types: `single` | `double`

## Parameters

### Roll (X-axis) Accuracy (deg) — 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 defined as rotation around the x-axis of the sensor body. Roll noise is modeled as white process noise with standard deviation equal to the specified **Roll accuracy** in degrees.

Data Types: `single` | `double`

### Pitch (Y-axis) Accuracy (deg) — 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 defined as rotation around the y-axis of the sensor body. Pitch noise is modeled as white process noise with standard deviation equal to the specified **Pitch Accuracy** in degrees.

Data Types: `single` | `double`

### Yaw (Z-axis) Accuracy (deg) — 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 defined as rotation around the  $z$ -axis of the sensor body. Yaw noise is modeled as white process noise with standard deviation equal to the specified **Yaw Accuracy** in degrees.

Data Types: `single` | `double`

**Position Accuracy (m) — Accuracy of position measurement (deg)**

1 (default) | nonnegative real scalar

Accuracy of the position measurement of the sensor body in meters, specified as a nonnegative real scalar.

Position noise is modeled as white process noise with a standard deviation equal to the specified **Position Accuracy** in degrees.

Data Types: `single` | `double`

**Velocity Accuracy (m/s) — Accuracy of velocity measurement (deg)**

1 (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 white process noise with a standard deviation equal to the specified **Velocity Accuracy** in degrees.

Data Types: `single` | `double`

**Initial Seed — Initial seed for randomization**

67 (default) | nonnegative integer

Initial seed of a random number generator algorithm, specified as a nonnegative integer.

Data Types: `single` | `double`

**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 code. The first time that you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations if the model does not change. This option requires additional startup time.

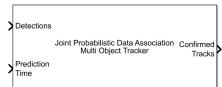
**See Also**

`IMU` | `insSensor`

**Introduced in R2020b**

# Joint Probabilistic Data Association Multi Object Tracker

Joint probabilistic data association tracker  
**Library:** Sensor Fusion and Tracking Toolbox



## Description

The Joint Probabilistic Data Association Multi Object Tracker block is capable of processing detections of multiple targets from multiple sensors. The tracker uses joint probabilistic data association to assign detections to each track. The tracker applies a soft assignment, in which multiple detections can contribute to each track. The tracker initializes, confirms, corrects, predicts (performs coasting), and deletes tracks. The tracker estimates the state vector and state estimate error covariance matrix for each track. Each detection is assigned to at least one track. If the detection cannot be assigned to any existing track, the tracker creates a new track.

Any new track starts in a *tentative* state. If enough detections are assigned to a tentative track, its status changes to *confirmed*. If the detection already has a known classification (the `ObjectClassID` field of the returned track is nonzero), that corresponding track is confirmed immediately. When a track is confirmed, the tracker considers the track to represent a physical object. If detections are not assigned to the track within a specifiable number of updates, the track is deleted.

## Ports

### Input

#### Detections — Detection list

Simulink bus containing MATLAB structure

Detection list, specified as a Simulink bus containing a MATLAB structure. The structure has the form:

Field	Description	Type
<code>NumDetections</code>	Number of detections	Integer
<code>Detections</code>	Object detections	Array of <code>objectDetection</code> structures. The first <code>NumDetections</code> of these detections are actual detections.

The fields of detections are:

Field	Description	Type
<code>Time</code>	Measurement time	Single or Double
<code>Measurement</code>	Object measurements	Single or Double

Field	Description	Type
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

See [objectDetection](#) for more detailed explanation of these fields.

---

**Note** The object detection structure contains a **Time** field. The time tag of each object detection must be less than or equal to the time of the current invocation of the block. The time tag must also be greater than the update time specified in the previous invocation of the block.

---

#### Prediction Time — Track update time

real scalar

Track update time, specified as a real scalar in seconds. The tracker updates all tracks to this time. The update time must always increase with each invocation of the block. The update time must be at least as large as the largest **Time** specified in the **Detections** input port.

If the port is not enabled, the simulation clock managed by Simulink determines the update time.

#### Dependencies

To enable this port, on the **Port Setting** tab, set **Prediction time source** to **Input port**.

#### Cost Matrix — Cost matrix

real-valued  $N_t$ -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 from the **All Tracks** output port on the previous invocation of the block.

In the first update to the tracker, or if 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**.

If this port is not enabled, the filter initialized by the **Filter initialization function** calculates the cost matrix using the distance method.

#### Dependencies

To enable this port, on the **Port Setting** tab, select **Enable cost matrix input**.

**Detectable TrackIDs — 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 enables you to add the detection probability for each track.

Tracks whose identifiers are not included in **Detectable TrackIDs** are considered undetectable. The track deletion logic does not count the lack of detection as a "missed detection" for track deletion purposes.

If this port is not enabled, the tracker assumes all tracks to be detectable at each invocation of the block.

**Dependencies**

To enable this port, on the **Port Setting** tab, select **Enable detectable track IDs Input**.

**Output****Confirmed Tracks — Confirmed tracks**

Simulink bus containing MATLAB structure

Confirmed tracks, returned as a Simulink bus containing a MATLAB structure. The structure has the form:

Field	Description
NumTracks	Number of tracks
Tracks	Array of track structures of a length set by the <b>Maximum number of tracks</b> parameter. Only the first NumTracks of these are actual tracks.

The fields of the track structure are shown in "Track Structure" on page 4-59.

Depending on the track logic, a track is confirmed if:

- History – A track receives at least  $M$  detections in the last  $N$  updates.  $M$  and  $N$  are specified in **Confirmation threshold** for the History logic.
- Integrated – The integrated probability of track existence is higher than the confirmation threshold specified in **Confirmation threshold** for the Integrated logic.

**Tentative Tracks — Tentative tracks**

Simulink bus containing MATLAB structure

Tentative tracks, returned as a Simulink bus containing a MATLAB structure. A track is tentative before it is confirmed.

The fields of the track structure are shown in "Track Structure" on page 4-59.

**Dependencies**

To enable this port, on the **Port Setting** tab, select **Enable tentative tracks output**.

**All Tracks – Confirmed and tentative tracks**

Simulink bus containing MATLAB structure

Combined list of confirmed and tentative tracks, returned as a Simulink bus containing a MATLAB structure.

The fields of the track structure are shown in “Track Structure” on page 4-59.

**Dependencies**

To enable this port, on the **Port Setting** tab, select **Enable all tracks output**.

**Info – Additional information for analyzing track updates**

Simulink bus containing MATLAB structure

Additional information for analyzing track updates, returned as a Simulink bus containing a MATLAB structure.

This table shows the fields of the info structure:

Field	Description
TrackIDsAtStepBeginning	Track IDs when step began.
CostMatrix	Cost matrix for assignment.
Clusters	Cell array of cluster reports. See “Feasible Joint Events” on page 4-58 for more details.
InitiatedTrackIDs	IDs of tracks initiated during the step.
DeletedTrackIDs	IDs of tracks deleted during the step.
TrackIDsAtStepEnd	Track IDs when the step ended.

The **Clusters** field can include multiple cluster reports. Each cluster report is a structure containing:

Field	Description
DetectionIndices	Indices of clustered detections.
TrackIDs	Track IDs of clustered tracks.
ValidationMatrix	Validation matrix of the cluster. See “Feasible Joint Events” on page 4-58 for more details.
SensorIndex	Index of the originating sensor of the clustered detections.
TimeStamp	Mean time stamp of clustered detections.
MarginalProbabilities	Matrix of marginal posterior joint association probabilities.

**Dependencies**

To enable this port, on the **Port Setting** tab, select **Enable information output**.

## Parameters

### Tracker Management

#### **Filter initialization function — Filter initialization function**

@initcvekf (default) | function handle | character vector

Filter initialization function, specified as a function handle or as a character vector containing the name of a valid filter initialization function. The tracker uses the filter initialization function when creating new tracks.

Sensor Fusion and Tracking Toolbox supplies many initialization functions:

Initialization Function	Function Definition
initcvkf	Initialize constant-velocity linear Kalman filter.
initcakf	Initialize constant-acceleration linear Kalman filter.
initcvabf	Initialize constant-velocity alpha-beta filter
initcaabf	Initialize constant-acceleration alpha-beta filter
initcvekf	Initialize constant-velocity extended Kalman filter.
initcaekf	Initialize constant-acceleration extended Kalman filter.
initrpekf	Initialize constant-velocity range-parametrized extended Kalman filter.
initapekf	Initialize constant-velocity angle-parametrized extended Kalman filter.
initctekf	Initialize constant-turn-rate extended Kalman filter.
initcackf	Initialize constant-acceleration cubature filter.
initctckf	Initialize constant-turn-rate cubature filter.
initcvckf	Initialize constant-velocity cubature filter.
initcvukf	Initialize constant-velocity unscented Kalman filter.
initcaukf	Initialize constant-acceleration unscented Kalman filter.
initctukf	Initialize constant-turn-rate unscented Kalman filter.
initcvmscekf	Initialize constant-velocity extended Kalman filter in modified spherical coordinates.
initekfimm	Initialize tracking IMM filter.

You can also write your own initialization function using this syntax:

```
filter = filterInitializationFcn(detection)
```

The input to this function is a detection report like those created by `objectDetection`. The output of this function must be a filter object: `trackingKF`, `trackingEKF`, `trackingUKF`, `trackingCKF`, `trackingGSF`, `trackingIMM`, `trackingMSCEKF`, or `trackingABF`.

For guidance in writing this function, use the `type` command to examine the details of built-in MATLAB functions. For example:

```
type initcvekf
```

---

**Note** trackerJPDA does not accept all filter initialization functions in Sensor Fusion and Tracking Toolbox. The full list of filter initialization functions available in Sensor Fusion and Tracking Toolbox are given in the **Initialization** section of “Estimation Filters”.

---

**Feasible joint events generation function name — Feasible joint events generation function name**

`@jpdaEvents` (default) | function handle | character vector

Feasible joint events generation function name, specified as a function handle or as a character vector containing the name of a feasible joint events generation function. This function generates feasible joint event matrices from admissible events (usually given by a validation matrix) of a tracking scenario. A validation matrix is a binary matrix listing all possible detections-to-track associations. For details, see `jpadEvents`.

You can also write your own generation function. The function must have this syntax:

```
FJE = myfunction(ValidationMatrix)
```

The input and output of this function must exactly follow the formats used in `jpdaEvents`. For guidance in writing this function, use the `type` command to examine the details of `jpdaEvents`:

```
type jpdaEvents
```

**Maximum number of tracks — Maximum number of tracks**

`100` (default) | positive integer

Maximum number of tracks that the block can maintain, specified as a positive integer.

**Maximum number of sensors — Maximum number of sensors**

`20` (default) | positive integer

Maximum number of sensors that the block can process, specified as a positive integer. This value should be greater than or equal to the highest `SensorIndex` value input at the **Detections** input port.

**Absolute tolerance between time stamps of detections — Absolute tolerance between time stamps of detections**

`20` (default) | positive integer

Absolute time tolerance between detections for the same sensor, specified as a positive scalar. The block expects detections from a sensor to have identical time stamps. However, if the time stamp differences between detections of a sensor are within the margin specified by this parameter, these detections will be used to update the track estimate based on the average time of these detections.

**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 code. The first time you run a simulation, Simulink generates 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.

### Assignment

#### **Threshold for assigning detections to tracks — Threshold for assigning detections to tracks**

30\*[1 Inf] (default) | positive scalar | 1-by-2 vector of positive values

Threshold for assigning detections to tracks (or gating threshold), specified as a positive scalar or 1-by-2 vector of  $[C_1, C_2]$ , where  $C_1 \leq C_2$ . If specified as a scalar, the specified value,  $val$ , is 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 the accurate normalized distance between them is less than  $C_1$ . See the **distance** function used with tracking filters (such as **trackingCKF** and **trackingEKF**) for explanation of the distance calculation.

Tips:

- Increase the value of  $C_2$  if there are track and detection combinations that should be calculated for assignment but are not. Decrease this value if the 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 this value if there are detections that are assigned to tracks they should not be assigned to (too far away).

#### **Threshold to initialize a track — Threshold to initialize a track**

0 (default) | scalar in the range [0, 1]

The probability threshold to initialize a new track, specified as a scalar in the range [0, 1]. If the probabilities of associating a detection with any of the existing tracks are all smaller than **InitializationThreshold**, the detection is used to initialize a new track. This allows detections that are within the validation gate of a track but have an association probability lower than the initialization threshold to spawn a new track.

Example: 0.1

#### **Probability of detection — Probability of detection**

0.9 (default) | scalar in the range [0, 1]

Probability of detection, specified as a scalar in the range [0, 1]. This property is used in calculations of the marginal posterior probabilities of association and the probability of track existence when initializing and updating a track.

#### **Spatial density of clutter measurements — Spatial density of clutter measurements**

1e-5 (default) | positive scalar

Spatial density of clutter measurements, specified as a positive scalar. The clutter density describes the expected number of false positive detections per unit volume. It is used as the parameter of a

Poisson clutter model. When **Type of track confirmation and deletion logic** is set to 'Integrated', this parameter is also used in calculating the initial probability of track existence.

#### Track Logic

##### **Type of track confirmation and deletion logic – Confirmation and deletion logic type**

History (default) | Integrated

Confirmation and deletion logic type, selected 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.
- Integrated – Track confirmation and deletion is based on the probability of track existence, which is integrated in the assignment function.

##### **Confirmation threshold [M N] – Track confirmation threshold for history logic**

[2, 3] (default) | real-valued 1-by-2 vector of positive integers

Track confirmation threshold for history logic, specified as a real-valued 1-by-2 vector of positive integers [M N]. A track is confirmed if it receives at least M detections in the last N updates.

#### Dependencies

To enable this parameter, set **Type of track confirmation and deletion logic** to 'History'.

##### **Deletion threshold [P Q] – Track deletion threshold for history logic**

[5, 5] (default) | real-valued 1-by-2 vector of positive integers

Track deletion threshold for history logic, specified as a real-valued 1-by-2 vector of positive integers, [P Q]. If, in P of the last Q tracker updates, a confirmed track is not assigned to any detection that has a likelihood greater than the **Threshold for registering 'hit' or 'miss'** parameter, then that track is deleted.

#### Dependencies

To enable this parameter, set **Type of track confirmation and deletion logic** to 'History'.

##### **Threshold for registering 'hit' or 'miss' – Threshold for registering a 'Hit' or a 'Miss'**

0.2 (default) | scalar in the range [0, 1]

Threshold for registering a 'hit' or 'miss', specified as a scalar in the range [0, 1]. The track history logic registers a 'miss' and the track will be coasted if the sum of the marginal probabilities of assignments is below this threshold. Otherwise, the track history logic registers a 'hit'.

#### Dependencies

To enable this parameter, set **Type of track confirmation and deletion logic** to 'History'.

##### **Confirmation threshold [Probability] – Track confirmation threshold for integrated logic**

0.95 (default) | positive scalar

Track confirmation threshold for integrated logic, specified as a real-valued positive scalar. A track is confirmed if its probability of existence is greater than or equal to the confirmation threshold.

**Dependencies**

To enable this parameter, set **Type of track confirmation and deletion logic** to 'Integrated'.

**Deletion threshold [Probability] – Track deletion threshold for integrated logic**

0.1 (default) | positive scalar

Track deletion threshold for integrated logic, specified as a positive scalar. A track is deleted if its probability of existence drops below this threshold.

**Dependencies**

To enable this parameter, set **Type of track confirmation and deletion logic** to 'Integrated'.

**Spatial density of new targets – Spatial density of new targets**

1e-5 (default) | positive scalar

Spatial density of new targets, specified as a positive scalar. The new target density describes the expected number of new tracks per unit volume in the measurement space. It is used in calculating the probability of track existence during track initialization.

**Dependencies**

To enable this parameter, set **Type of track confirmation and deletion logic** to 'Integrated'.

**Time rate of true target deaths – Time rate of true target deaths**

0.01 (default) | scalar in the range [0, 1]

Time rate of true target deaths, specified as a scalar in the range [0, 1]. This parameter describes the probability with which true targets disappear. It is related to the propagation of the probability of track existence (*PTE*) :

$$PTE(t + \delta t) = (1 - DeathRate)^{\delta t} PTE(t)$$

where *DeathRate* is the time rate of true target deaths, and  $\delta t$  is the time interval since the previous update time *t*.

**Dependencies**

To enable this parameter, set **Type of track confirmation and deletion logic** to 'Integrated'.

**Port Setting****Prediction time source – Source of prediction time**

Auto (default) | Input port

Source for prediction time, specified as Input port or Auto. Select Input port to input an update time by using the **Prediction Time** input port. Otherwise, the simulation clock managed by Simulink determines the update time.

**Enable cost matrix input – Enable input port for cost matrix**

off (default) | on

Select this check box to enable the input of a cost matrix by using the **Cost Matrix** input port.

**Enable detectable track IDs input — Enable detectable track IDs input**  
off (default) | on

Select this check box to enable the **Detectable track IDs** input port.

**Enable tentative tracks output — Enable output port for tentative tracks**  
off (default) | on

Select this check box to enable the output of tentative tracks through the **Tentative Tracks** output port.

**Enable all tracks output — Enable output port for all tracks**  
off (default) | on

Select this check box to enable the output of all the tracks through the **All Tracks** output port.

**Enable information output — Enable output port for analysis information**  
off (default) | on

Select this check box to enable the output port for analysis information through the **Info** output port.

**Source of output bus name — Source of output track bus name**  
Auto (default) | Property

Source of the output track bus name, specified as:

- Auto — The block automatically creates an output track bus name.
- Property — Specify the output track bus name by using the **Specify an output bus name** parameter.

**Source of output info bus name — Source of output info bus name**  
Auto (default) | Property

Source of the output info bus name, specified as one of these options:

- Auto — The block automatically creates an output info bus name.
- Property — Specify the output info bus name by using the **Specify an output bus name** parameter.

## Algorithms

### Tracker Logic Flow

When a joint probabilistic data association (JPDA) tracker processes detections, track creation and management follow these steps:

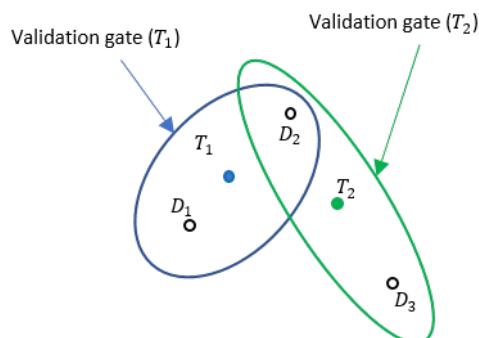
- 1 The tracker divides detections into multiple groups by originating sensor.
- 2 For each sensor:
  - a The tracker calculates the distances from detections to existing tracks and forms a **costMatrix**.
  - b The tracker creates a validation matrix based on the assignment threshold (or gate threshold) of the existing tracks. A validation matrix is a binary matrix listing all possible detections-to-track associations. For details, see “Feasible Joint Events” on page 4-58.

- c Tracks and detections are then separated into clusters. A cluster can contain one track or multiple tracks if these tracks share common detections within their validation gates. A validation gate is a spatial boundary, in which the predicted detection of the track has a high likelihood to fall. For details, see "Feasible Joint Events" on page 4-58.
- 3 Update all clusters following the order of the mean detection time stamp within the cluster. For each cluster, the tracker:
  - a Generates all feasible joint events. For details, see `jpdEvents`.
  - b Calculates the posterior probability of each joint event.
  - c Calculates the marginal probability of each individual detection-track pair in the cluster.
  - d Reports weak detections. Weak detections are the detections that are within the validation gate of at least one track, but have probability association to all tracks less than the `InitializationThreshold`.
  - e Updates tracks in the cluster using `correctjpd`.
- 4 Unassigned detections (detections not in any cluster) and weak detections spawn new tracks.
- 5 The tracker checks all tracks for deletion. Tracks are deleted based on the number of scans without association using 'History' logic or based on their probability of existence using 'Integrated' track logic.
- 6 All tracks are predicted to the latest time value (either the time input if provided, or the latest mean cluster time stamp).

### Feasible Joint Events

In the typical workflow for a tracking system, the tracker needs to determine if a detection can be associated with any of the existing tracks. If the tracker only maintains one track, the assignment can be done by evaluating the validation gate around the predicted measurement and deciding if the measurement falls within the *validation gate*. In the measurement space, the validation gate is a spatial boundary, such as a 2-D ellipse or a 3-D ellipsoid, centered at the predicted measurement. The validation gate is defined using the probability information (state estimation and covariance, for example) of the existing track, such that the correct or ideal detections have high likelihood (97% probability, for example) of falling within this validation gate.

However, if a tracker maintains multiple tracks, the data association process becomes more complicated, because one detection can fall within the validation gates of multiple tracks. For example, in the following figure, tracks  $T_1$  and  $T_2$  are actively maintained in the tracker, and each of them has its own validation gate. Since the detection  $D_2$  is in the intersection of the validation gates of both  $T_1$  and  $T_2$ , the two tracks ( $T_1$  and  $T_2$ ) are connected and form a *cluster*. A cluster is a set of connected tracks and their associated detections.



To represent the association relationship in a cluster, the validation matrix is commonly used. Each row of the validation matrix corresponds to a detection while each column corresponds to a track. To account for the eventuality of each detection being clutter, a first column is added and usually referred to as "Track 0" or  $T_0$ . If detection  $D_i$  is inside the validation gate of track  $D_j$ , then the  $(j, i+1)$  entry of the validation matrix is 1. Otherwise, it is zero. For the cluster shown in the figure, the validation matrix  $\Omega$  is

$$\Omega = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Note that all the elements in the first column of  $\Omega$  are 1, because any detection can be clutter or false alarm. One important step in the logic of joint probabilistic data association (JPDA) is to obtain all the feasible independent joint events in a cluster. Two assumptions for the feasible joint events are:

- A detection cannot be emitted by more than one track.
- A track cannot be detected more than once by the sensor during a single scan.

Based on these two assumptions, feasible joint events (FJEs) can be formulated. Each FJE is mapped to an FJE matrix  $\Omega_p$  from the initial validation matrix  $\Omega$ . For example, with the validation matrix  $\Omega$ , eight FJE matrices can be obtained:

$$\Omega_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \Omega_2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \Omega_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad \Omega_4 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

$$\Omega_5 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \quad \Omega_6 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \Omega_7 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \Omega_8 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

As a direct consequence of the two assumptions, the  $\Omega_p$  matrices have exactly one "1" value per row. Also, except for the first column which maps to clutter, there can be at most one "1" per column. When the number of connected tracks grows in a cluster, the number of FJE increases rapidly. The `jpdaEvents` function uses an efficient depth-first search algorithm to generate all the feasible joint event matrices.

## Track Structure

The fields of a track structure are:

Field	Definition
SourceIndex	Unique source index used to distinguish tracking sources in a multiple tracker environment.
TrackID	Unique track identifier used to distinguish multiple tracks.
BranchID	Unique track branch identifier used to distinguish multiple track branches.
UpdateTime	Time at which the track is updated. Units are in seconds.
Age	Number of times the track survived.

Field	Definition
State	Value of state vector at the update time.
StateCovariance	Uncertainty covariance matrix.
TrackLogic	Confirmation and deletion logic type, returned as 'History' or 'Integrated'.
TrackLogicState	The current state of the track logic type. Based on the logic type <code>TrackLogic</code> , the logic state is returned as: <ul style="list-style-type: none"> <li>'History' - A 1-by-<math>K</math> logical array, where <math>K</math> is the number of latest track logical states recorded. In the array, 1 denotes hit and 0 denote miss.</li> <li>'Integrated' - A nonnegative scalar. The scalar represents the integrated probability of existence of the track. The default value is 0.5.</li> </ul>
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.
ObjectClassID	Integer value representing the object classification. The value 0 represents an unknown classification. Nonzero classifications apply only to confirmed tracks.
ObjectAttributes	Additional information of the track.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Blocks

Global Nearest Neighbor Multi Object Tracker

### Functions

`correctJpda` | `getTrackPositions` | `getTrackVelocities` | `jPDAEvents` | `predictTracksToTime`

### Objects

`objectDetection` | `objectTrack` | `trackHistoryLogic` | `trackingABF` | `trackingCKF` | `trackingEKF` | `trackingIMM` | `trackingKF` | `trackingUKF`

### System Objects

`staticDetectionFuser` | `trackerGNN` | `trackerTOMHT`

**Topics**

“Introduction to Multiple Target Tracking”  
“Introduction to Assignment Methods in Tracking Systems”

**Introduced in R2019b**

# Track-Oriented Multi-Hypothesis Tracker

Track-Oriented Multi-Hypothesis Tracker

**Library:** Sensor Fusion and Tracking Toolbox



## Description

The Track-Oriented Multi-Hypothesis Tracker block processes detections of multi targets from multiple sensors. The tracker block initializes, confirms, predicts, corrects, and deletes tracks. Inputs to the tracker block are detection reports generated by `objectDetection`, `radarSensor`, `monostaticRadarSensor`, `irSensor`, or `sonarSensor` objects. The tracker block estimates the state vector and state vector covariance matrix for each track. The tracker assigns detections based on a track-oriented, multi-hypothesis approach.

Any new track starts in a *tentative* state. If enough detections are assigned to a tentative track, its status changes to *confirmed*. If the detection already has a known classification (the `ObjectClassID` field of the returned track is nonzero), that track is confirmed immediately. When a track is confirmed, the multi-object tracker considers the track to represent a physical object. If detections are not assigned to the track within a specifiable number of updates, the track is deleted. For an overview of how the tracker functions, see “Algorithms” on page 4-72.

## Ports

### Input

#### Detections — Detection list

Simulink bus containing MATLAB structure

Detection list, specified as a Simulink bus containing a MATLAB structure. The structure has the form:

Field	Description	Type
<code>NumDetections</code>	Number of detections	Integer
<code>Detections</code>	Object detections	Array of <code>objectDetection</code> structures. The first <code>NumDetections</code> of these detections are actual detections.

The fields of the detections structure are:

Field	Description	Type
<code>Time</code>	Measurement time	Single or Double
<code>Measurement</code>	Object measurements	Single or Double

Field	Description	Type
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

See [objectDetection](#) for a more detailed explanation of these fields.

---

**Note** The object detection structure contains a **Time** field. The time tag of each object detection must be less than or equal to the time at the current invocation of the block. The time tag must also be greater than the update time specified in the previous invocation of the block.

---

#### Prediction Time — Track update time

real scalar

Track update time, specified as a real scalar in seconds. The tracker updates all tracks to this time. The update time must always increase with each invocation of the block. The update time must be at least as large as the largest **Time** specified in the **Detections** input port.

If the port is not enabled, the simulation clock managed by Simulink determines the update time.

#### Dependencies

To enable this port, on the **Port Setting** tab, set **Prediction time source** to **Input port**.

#### Cost Matrix — Cost matrix

real-valued  $N$ -by- $M$  matrix

Cost matrix, specified as a real-valued  $N$ -by- $M$  matrix, where  $N$  is the number of branches and  $M$  is the number of current detections.

The rows of the cost matrix must be in the same order as the list of branches. Branches are ordered as they appear in the list of branches from the **All Branches** output port on the previous invocation of the block. The columns correspond to the detections.

In the first update to the tracker, or if the tracker has no previous tracks, assign the cost matrix a size of  $[0, N]$ . 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**.

If this port is not enabled, the filter initialized by the **Filter initialization function** calculates the cost matrix using the distance method.

#### Dependencies

To enable this port, on the **Port Setting** tab, select **Enable cost matrix input**.

**Detectable BranchIDs — Detectable Branch IDs**real-valued  $M$ -by-1 vector | real-valued  $M$ -by-2 matrix

Detectable branch IDs, specified as a real-valued  $M$ -by-1 vector or  $M$ -by-2 matrix. Detectable branches are branches that the sensors expect to detect. The first column of the matrix contains a list of branch IDs that the sensors report as detectable. The optional second column enables you to add the detection probability for each branch. Branches are listed in the **All Branches** output from the previous invocation of the block.

Tracks whose identifiers are not included in **Detectable BranchIDs** are considered undetectable. The track deletion logic does not count the lack of detection as a "missed detection" for track deletion purposes.

If this port is not enabled, the tracker assumes all tracks to be detectable at each invocation of the block.

**Dependencies**

To enable this port, on the **Port Setting** tab, select **Enable detectable branch IDs input**.

**Output****Confirmed Tracks — Confirmed tracks**

Simulink bus containing MATLAB structure

Confirmed tracks, returned as a Simulink bus containing a MATLAB structure. The structure has the form:

Field	Description
NumTracks	Number of tracks
Tracks	Array of track structures of a length set by the <b>Maximum number of tracks</b> parameter. Only the first NumTracks of these are actual tracks.

The fields of the track structure are shown in "Track Structure" on page 4-74.

A track is confirmed if it satisfies the threshold specified in the **Confirmation threshold** parameter under the **Track Logic** tab.

**Tentative Tracks — Tentative tracks**

Simulink bus containing MATLAB structure

Tentative tracks, returned as a Simulink bus containing a MATLAB structure. A track is tentative before it is confirmed.

The fields of the track structure are shown in "Track Structure" on page 4-74.

**Dependencies**

To enable this port, on the **Port Setting** tab, select **Enable tentative tracks output**.

**All Tracks — Confirmed and tentative tracks**

Simulink bus containing MATLAB structure

Combined list of confirmed and tentative tracks, returned as a Simulink bus containing a MATLAB structure.

The fields of the track structure are shown in “Track Structure” on page 4-74.

### Dependencies

To enable this port, on the **Port Setting** tab, select **Enable all tracks output**.

### Info — Additional information for analyzing track updates

Simulink bus containing MATLAB structure

Additional information for analyzing track updates, returned as a Simulink bus containing a MATLAB structure.

This table shows the fields of the info structure:

Field	Description
BranchIDsAtStepBeginning	Branch IDs when the update began.
CostMatrix	Cost of assignment matrix.
Assignments	Assignments returned from the <code>assignTOMHT</code> function.
UnassignedTracks	IDs of unassigned branches returned from the tracker.
UnassignedDetections	IDs of unassigned detections returned from the tracker.
InitialBranchHistory	Branch history after branching and before pruning.
InitialBranchScores	Branch scores before pruning.
KeptBranchHistory	Branch history after initial pruning.
KeptBranchScores	Branch scores after initial pruning.
Clusters	Logical array mapping branches to clusters. Branches belong in the same cluster if they share detections in their history or belong to the same track either directly or through other branches.
TrackIncompatibility	Branch incompatibility matrix. The $(i, j)$ element is true if the $i$ -th and $j$ -th branches have shared detections in their history or belong to the same track.
GlobalHypotheses	Logical matrix mapping branches to global hypotheses. Compatible branches can belong in the same hypotheses.
GlobalHypScores	Total score of global hypotheses.
PrunedBranches	Logical array of branches that the <code>pruneTrackBranches</code> function determines to prune.

<code>GlobalBranchProbabilities</code>	Global probability of each branch existing in the global hypotheses.
<code>BranchesDeletedByPruning</code>	Branches deleted by the tracker.
<code>BranchIDsAtStepEnd</code>	Branch IDs when the update ended.

#### Dependencies

To enable this port, on the **Port Setting** tab, select **Enable information output**.

#### All Branches — All branches

Simulink bus containing MATLAB structure

All branches, returned as a Simulink bus containing a MATLAB structure.

The fields of the branch structure are the same as the “Track Structure” on page 4-74.

#### Dependencies

To enable this port, on the **Port Setting** tab, select **Enable all branches output**.

## Parameters

### Tracker Management

#### Tracker identifier — Unique tracker identifier

0 (default) | nonnegative integer

Unique tracker identifier, specified as a nonnegative integer. This parameter is passed as the `SourceIndex` in the tracker outputs, and distinguishes tracks that come from different trackers in a multiple-tracker system. You must specify this property as a positive integer to use the track outputs as inputs to a `trackFuser` object.

Example: 1

#### Filter initialization function — Filter initialization function

@initcvekf (default) | function handle | character vector

Filter initialization function, specified as a function handle or as a character vector containing the name of a filter initialization function. The tracker uses a filter initialization function when creating new tracks.

Sensor Fusion and Tracking Toolbox provides many initialization functions that are compatible with this block.

Initialization Function	Function Definition
<code>initcvabf</code>	Initialize constant-velocity alpha-beta filter
<code>initcaabf</code>	Initialize constant-acceleration alpha-beta filter
<code>initcvekf</code>	Initialize constant-velocity extended Kalman filter.
<code>initcackf</code>	Initialize constant-acceleration cubature filter.
<code>initctckf</code>	Initialize constant-turn-rate cubature filter.
<code>initcvckf</code>	Initialize constant-velocity cubature filter.

Initialization Function	Function Definition
<code>initcapf</code>	Initialize constant-acceleration particle filter.
<code>initctpf</code>	Initialize constant-turn-rate particle filter.
<code>initcvpf</code>	Initialize constant-velocity particle 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-turn-rate extended Kalman filter.
<code>initctukf</code>	Initialize constant-turn-rate unscented Kalman filter.
<code>initcvmscekf</code>	Initialize constant-velocity modified spherical coordinates extended Kalman filter.
<code>initrpekf</code>	Initialize constant-velocity range-parametrized extended Kalman filter.
<code>initapekf</code>	Initialize constant-velocity angle-parametrized extended Kalman filter.
<code>initekfimm</code>	Initialize tracking IMM 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 the `objectDetection` object. The output of this function must be a filter object: `trackingKF`, `trackingEKF`, `trackingUKF`, `trackingCKF`, `trackingPF`, `trackingMSCEKF`, `trackingGSF`, `trackingIMM`, or `trackingABF`.

To guide you in writing this function, you can examine the details of the supported functions from within MATLAB. For example:

```
type initcvekf
```

Data Types: function\_handle | char

#### **Threshold for assigning detections to tracks – Threshold for assigning detections to tracks**

`30*[0.3 0.7 1 Inf]` (default) | positive scalar | 1-by-3 vector of positive values | 1-by-4 vector of positive values

Threshold for assigning detections to tracks, specified as a positive scalar, a 1-by-3 vector of non-decreasing positive values,  $[C_1, C_2, C_3]$ , or a 1-by-4 vector of non-decreasing positive values,  $[C_1, C_2, C_3, C_4]$ . If specified as a scalar, the specified value,  $val$ , will be expanded to  $[0.3, 0.7, 1, \text{Inf}] * val$ . If specified as  $[C_1, C_2, C_3]$ , it will be expanded as  $[C_1, C_2, C_3, \text{Inf}]$ .

The thresholds control (1) the assignment of a detection to a track, (2) the creation of a new branch from a detection, and (3) the creation of a new branch from an unassigned track. The threshold values must satisfy:  $C_1 \leq C_2 \leq C_3 \leq C_4$ .

- $C_1$  defines a distance such that if a track has an assigned detection with lower distance than  $C_1$ , the track is no longer considered unassigned and does not create an unassigned track branch.
- $C_2$  defines a distance such that if a detection has been assigned to a track with lower distance than  $C_2$ , the detection is no longer considered unassigned and does not create a new track branch.
- $C_3$  defines the maximum distance for assigning a detection to a track.
- $C_4$  defines combinations of track and detection for which an accurate normalized cost calculation is performed. 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  $C_4$ .

Tips:

- Increase the value of  $C_3$  if there are detections that should be assigned to tracks but are not. Decrease the value if there are detections that are assigned to tracks they should not be assigned to (too far away).
- Increasing the values  $C_1$  and  $C_2$  helps control the number of track branches that are created. However, doing so reduces the number of branches (hypotheses) each track has.
- Increase the value of  $C_4$  if there are combinations of track and detection that should be calculated for assignment but are not. Decrease it if cost calculation takes too long.

Data Types: `single` | `double`

**Maximum number of tracks — Maximum number of tracks**

100 (default) | positive integer

Maximum number of tracks that the block can maintain, specified as a positive integer.

**Maximum number of sensors — Maximum number of sensors**

20 (default) | positive integer

Maximum number of sensors that can be connected to the tracker, specified as a positive integer. `MaxNumSensors` must be greater than or equal to the largest value of `SensorIndex` found in all the detections used to update the tracker. `SensorIndex` is one of the properties of an `objectDetection` object. The block's `MaxNumSensors` property determines how many sets of `ObjectAttributes` fields each output track can have.

**Track state parameters — Parameters of track-state reference frame**

`struct` | `struct array`

Parameters of 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.

Data Types: `struct`

**Track output method — Track output method**

'Tracks' (default) | 'Hypothesis' | 'Clusters'

Track output method, specified as 'Tracks', 'Hypothesis', or 'Clusters'.

- 'Tracks' – Output the centroid of each track based on its track branches.
- 'Hypothesis' – Output branches that are in certain hypotheses. If you choose this option, list the hypotheses to output using the `HypothesesToOutput` property.
- 'Clusters' – Output the centroid of each cluster. Similar to the 'Tracks' output, but includes all tracks within a cluster.

Data Types: char

#### **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 the **Interpreted execution** mode, you can debug the source code of the block.
- **Code generation** – Simulate the model using generated C code. The first time you run a simulation, Simulink generates 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.

#### **Hypotheses Management**

##### **Maximum number of hypotheses to be maintained — Maximum number of hypotheses to be maintained**

5 (default) | positive integer

Maximum number of hypotheses maintained by the tracks in cases of ambiguity, specified as a positive integer. Larger values increase the computational load.

Example: 10

Data Types: single | double

##### **Maximum number of track branches per track — Maximum number of track branches per track**

3 (default) | positive integer

Maximum number of track branches (hypotheses) allowed for each track, specified as a positive integer. Larger values increase the computational load.

Data Types: single | double

##### **Maximum number of scans maintained in the branch history — Maximum number of scans maintained in the branch history**

4 (default) | positive integer

Maximum number of scans maintained in the branch history, specified as a positive integer. The number of track history scans is typically from 2 through 6. Larger values increase the computational load.

Data Types: single | double

##### **Minimum probability required to keep a branch — Minimum probability required to keep a branch**

.001 (default) | positive scalar

Minimum probability required to keep a track branch, specified as a positive scalar less than one. Any track with probability lower than the specified probability is pruned. Typical values are 0.001 to 0.005.

Example: .003

Data Types: single | double

#### **N-scan pruning method — N-scan pruning method**

'None' (default) | 'Hypothesis'

N-scan pruning method, specified as 'None' or 'Hypothesis'. In N-scan pruning, branches that belong to the same track are pruned (deleted) if, in the N-scans history, they contradict the most likely branch for the same track. The most-likely branch is defined in one of two ways:

- 'None' - No N-scan pruning is performed.
- 'Hypothesis' - The chosen branch is in the most likely hypothesis.

Example: 'Hypothesis'

#### **Track Logic**

#### **Confirmation threshold [positive scalar] — Minimum score required to confirm track**

20 (default) | positive scalar

Minimum score required to confirm a track, specified as a positive scalar. Any track with a score higher than this threshold is confirmed.

Example: 12

Data Types: single | double

#### **Deletion threshold [negative scalar] — Maximum score drop for track deletion**

-7 (default) | scalar

The maximum score drop before a track is deleted, specified as a scalar. Any track with a score that falls by more than this parameter from the maximum score is deleted. Deletion threshold is affected by the probability of a false alarm.

Example: -1

Data Types: single | double

#### **Probability of detection used for track score — Probability of detection used for track score**

0.9 (default) | positive scalar between 0 and 1

Probability of detection, specified as a positive scalar between 0 and 1. This property is used to compute track score.

Example: 0.5

Data Types: single | double

#### **Rate of false positives used for track score — Probability of false alarm used for track score**

1e-6 (default) | scalar

The probability of false alarm, specified as a scalar. This property is used to compute track score.

Example: 1e-5

Data Types: single | double

**Volume of the sensor's detection bin — Volume of sensor measurement bin**

1 (default) | positive scalar

The volume of a sensor measurement bin, specified as a positive scalar. For example, if a radar produces a 4-D measurement, which includes azimuth, elevation, range, and range rate, the 4-D volume is defined by the radar angular beam width, the range bin width, and the range-rate bin width. Volume is used in calculating the track score when initializing and updating a track.

Example: 1.5

Data Types: single | double

**Rate of new tracks per unit volume — Rate of new tracks per unit volume**

1 (default) | positive scalar

The rate of new tracks per unit volume, specified as a positive scalar. The parameter is used in calculating the track score during track initialization.

Example: 2.5

Data Types: single | double

**Port Setting**

**Prediction time source — Source of prediction time**

Auto (default) | Input port

Source for prediction time, specified as Input port or Auto. Select Input port to input an update time by using the **Prediction Time** input port. Otherwise, the simulation clock managed by Simulink determines the update time.

**Enable cost matrix input — Enable input port for cost matrix**

off (default) | on

Select this parameter to enable the input of a cost matrix by using the **Cost Matrix** input port.

**Enable detectable branch IDs input — Enable detectable branch IDs input**

off (default) | on

Select this parameter to enable the **Detectable branch IDs** input port.

**Enable tentative tracks output — Enable output port for tentative tracks**

off (default) | on

Select this parameter to enable the output of tentative tracks through the **Tentative Tracks** output port.

**Enable all tracks output — Enable output port for all tracks**

off (default) | on

Select this parameter to enable the output of all the tracks through the **All Tracks** output port.

**Enable information output — Enable output port for analysis information**

off (default) | on

Select this parameter to enable the output port for analysis information through the **Info** output port.

**Enable all branches output — Enable output port for all branches**  
off (default) | on

Select this parameter to enable the output of all the branches through the **All Branches** output port.

**Source of output bus name — Source of output track bus name**  
Auto (default) | Property

Source of the output track bus name, specified as:

- Auto — The block automatically creates an output track bus name.
- Property — Specify the output track bus name by using the **Specify an output bus name** parameter.

**Source of output info bus name — Source of output information bus name**  
Auto (default) | Property

Source of the output info bus name, specified as:

- Auto — The block automatically creates an output info bus name.
- Property — Specify the output info bus name by using the **Specify an output info bus name** parameter.

**Dependencies**

To enable this parameter, on the **Port Setting** tab, select **Enable information output**.

## Algorithms

### Tracker Logic Flow

When you process detections using the tracker, track creation and management follow these steps.

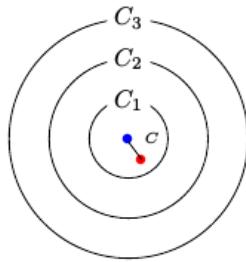
- 1 The tracker attempts to assign detections to existing tracks.
- 2 The track allows for multiple hypotheses about the assignment of detections to tracks.
- 3 Unassigned detections result in the creation of new tracks.
- 4 Assignments of detections to tracks create branches for the assigned tracks.
- 5 Tracks with no assigned detections are coasted (predicted).
- 6 All track branches are scored. Branches with low initial scores are pruned.
- 7 Clusters of branches that share detections (incompatible branches) in their history are generated.
- 8 Global hypotheses of compatible branches are formulated and scored.
- 9 Branches are scored based on their existence in the global hypotheses. Low-scored branches are pruned.
- 10 Additional pruning is performed based on N-scan history.
- 11 All tracks are corrected and predicted to the input time.

## Assignment Thresholds for Multi-Hypothesis Tracker

Three assignment thresholds,  $C_1$ ,  $C_2$ , and  $C_3$ , control (1) the assignment of a detection to a track, (2) the creation of a new branch from a detection, and (3) the creation of a new branch from an unassigned track. The threshold values must satisfy:  $C_1 \leq C_2 \leq C_3$ .

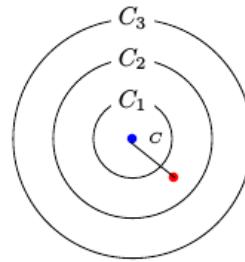
If the cost of an assignment is  $C = \text{costmatrix}(i, j)$ , the following hypotheses are created based on comparing the cost to the values of the assignment thresholds. Below each comparison, there is a list of the possible hypotheses.

- Track
- Detection



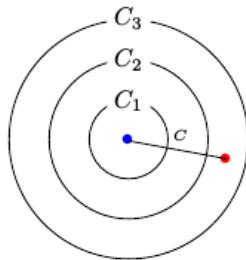
$C \leq C_1$   
Single Hypothesis

- (1) Detection is assigned to track. A branch is created updating the track with this detection.



$C_1 < C \leq C_2$   
Two Hypotheses

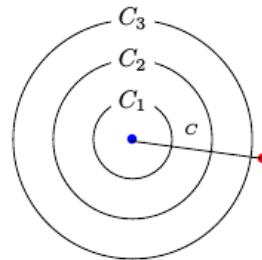
- (1) Detection is assigned to track. A branch is created updating the track with this detection.  
(2) Track is not assigned to detection and is coasted.



$C_2 < C \leq C_3$

Three Hypotheses

- (1) Detection is assigned to track. A branch is created updating the track with this detection.  
(2) Track is not assigned to detection and is coasted.  
(3) Detection is not assigned and creates a new track (branch).



$C_3 < C$

Single Hypothesis

- (1) Detection is not assigned and creates a new track (branch).

Tips:

- Increase the value of  $C_3$  if there are detections that should be assigned to tracks but are not. Decrease the value if there are detections that are assigned to tracks they should not be assigned to (too far away).
- Increasing the values  $C_1$  and  $C_2$  helps control the number of track branches that are created. However, doing so reduces the number of branches (hypotheses) each track has.
- To allow each track to be unassigned, set  $C_1 = 0$ .
- To allow each detection to be unassigned, set  $C_2 = 0$ .

### Data Precision

All numeric inputs can be single or double precision, but they all must have the same precision.

### Track Structure

The fields of a track structure are:

Field	Definition
SourceIndex	Unique source index used to distinguish tracking sources in a multiple tracker environment.
TrackID	Unique track identifier used to distinguish multiple tracks.
BranchID	Unique track branch identifier used to distinguish multiple track branches.
UpdateTime	Time at which the track is updated. Units are in seconds.
Age	Number of times the track survived.
State	Value of state vector at the update time.
StateCovariance	Uncertainty covariance matrix.
TrackLogic	Confirmation and deletion logic type, returned as 'History' or 'Integrated'.
TrackLogicState	The current state of the track logic type. Based on the logic type <code>TrackLogic</code> , the logic state is returned as: <ul style="list-style-type: none"> <li>• 'History' - A 1-by-<math>K</math> logical array, where <math>K</math> is the number of latest track logical states recorded. In the array, 1 denotes hit and 0 denote miss.</li> <li>• 'Integrated' - A nonnegative scalar. The scalar represents the integrated probability of existence of the track. The default value is 0.5.</li> </ul>
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.

Field	Definition
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.
ObjectClassID	Integer value representing the object classification. The value <code>0</code> represents an unknown classification. Nonzero classifications apply only to confirmed tracks.
ObjectAttributes	Additional information of the track.

## References

- [1] Werthmann, J. R. "Step-by-Step Description of a Computationally Efficient Version of Multiple Hypothesis Tracking." In *International Society for Optics and Photonics*, Vol. 1698, pp. 228-301, 1992.
- [2] Blackman, S., and R. Popoli. *Design and Analysis of Modern Tracking Systems*. Artech House Radar Library, Boston, 1999.

## Extended Capabilities

### C/C++ Code Generation

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

## See Also

### Functions

`getTrackPositions` | `getTrackVelocities`

### Objects

`objectDetection` | `objectTrack` | `trackingABF` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

### System Objects

`irSensor` | `monostaticRadarSensor` | `radarSensor` | `sonarSensor` | `trackerGNN`

### Introduced in R2020a



# Apps

---

# Tracking Scenario Designer

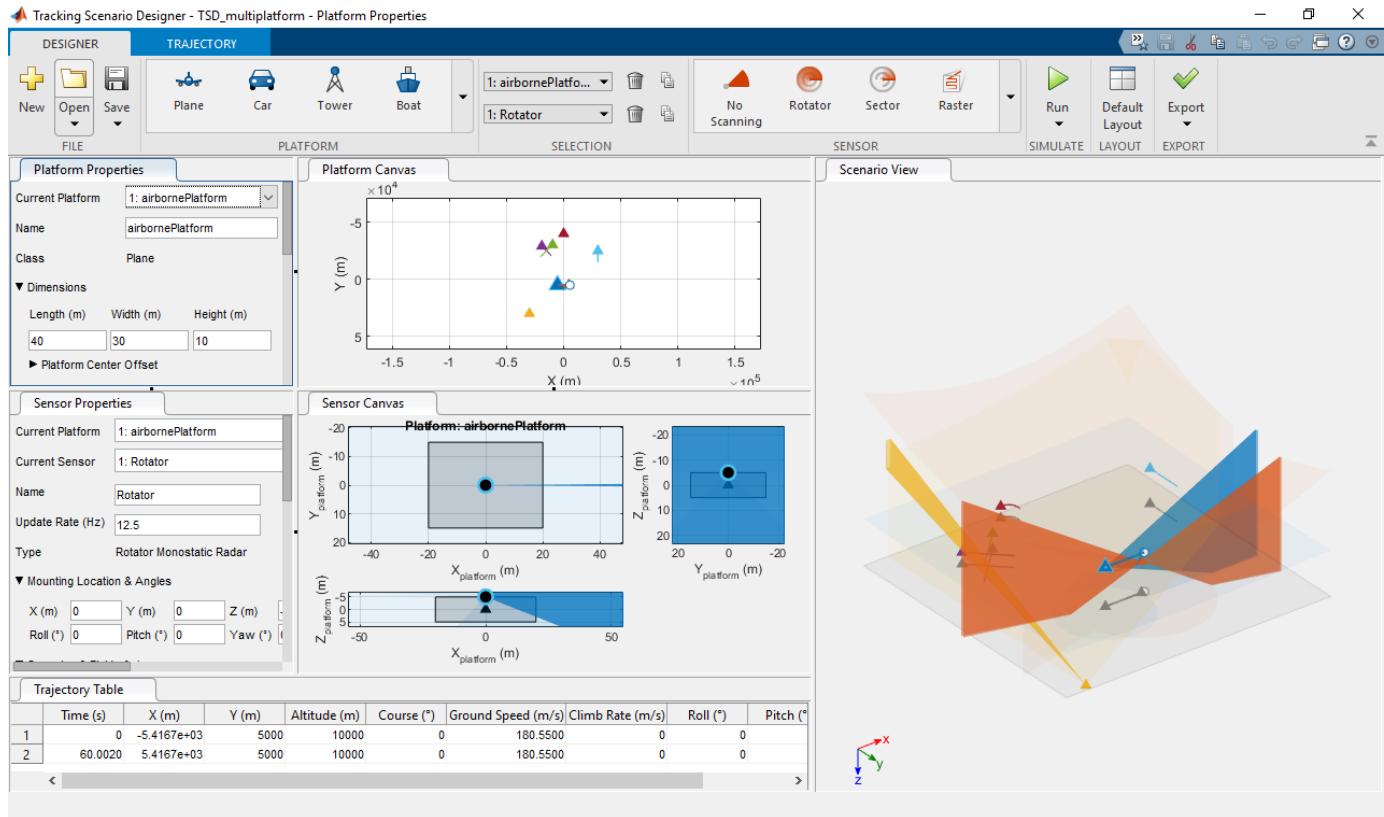
Design tracking scenarios, configure platforms and sensors, and generate synthetic object detections

## Description

The **Tracking Scenario Designer** app enables you to design and visualize synthetic tracking scenarios for testing your estimation and tracking systems.

Using the app, you can:

- Create platforms (including planes, cars, towers, and boats) using an interactive interface and configure platform properties in the tracking scenario.
- Configure 2D or 3D trajectories (including position, orientation, and velocities) of platforms using waypoint trajectories in the tracking scenario.
- Create radar sensors mounted on the platform and configure sensor properties.
- Simulate the tracking scenario and dynamically visualize the platform trajectories, sensor coverages, and object detections.
- Generate MATLAB code of the scenario and sensors, and then programmatically modify the scenario for application purposes. You can also import the previously saved scenario back into the app for further simulation.
- Import a `trackingScenario` object in the app for visualization and further design of the tracking scenario. See “Programmatic Use” on page 5-25 for the limitations of importing a `trackingScenario` object.



## Open the Tracking Scenario Designer App

- MATLAB Toolstrip: On the Apps tab, under **Signal processing and communications**, click the  app icon.
- MATLAB command prompt: Enter `trackingScenarioDesigner`.

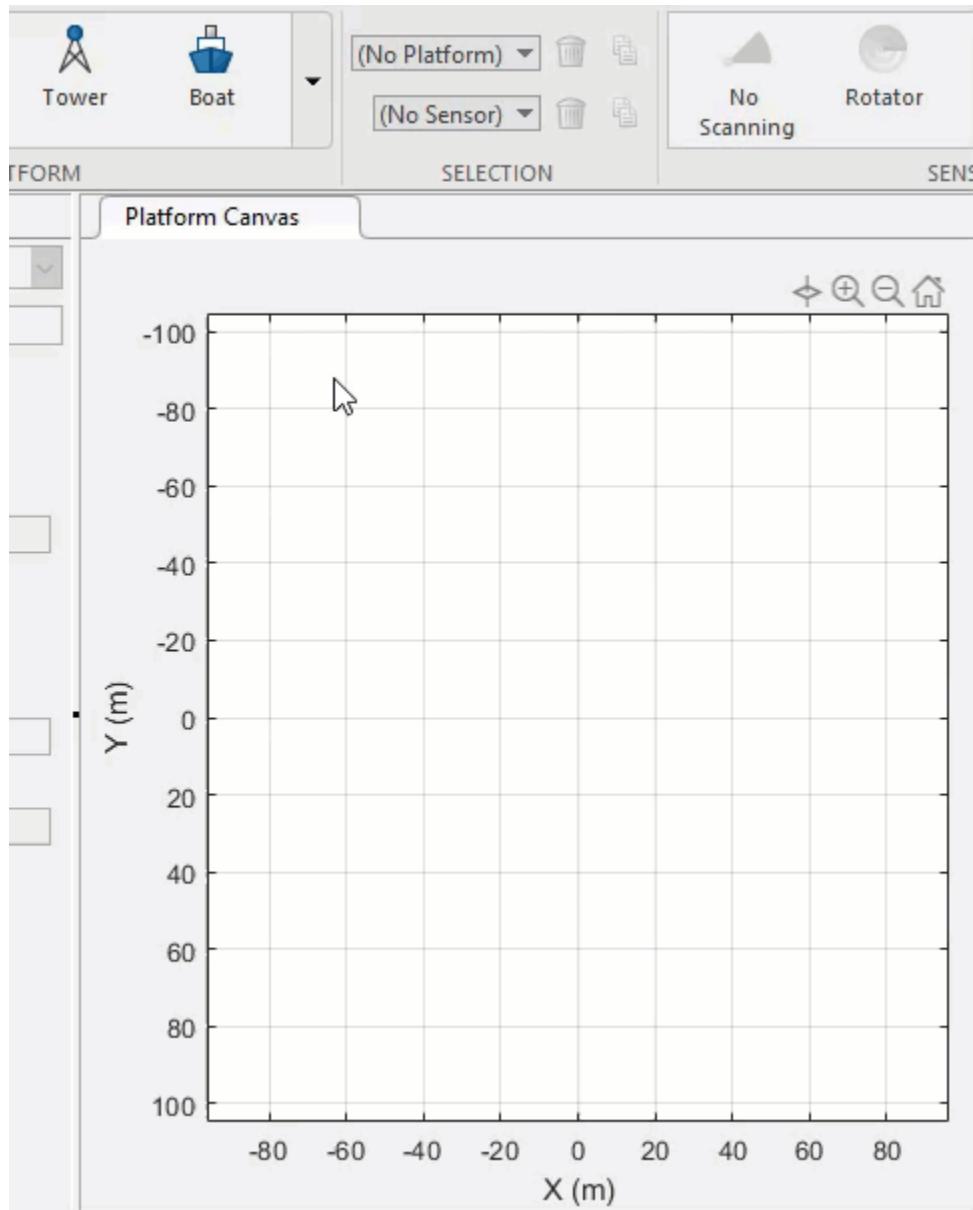
## Examples

### Set Up Platforms in Tracking Scenario Designer

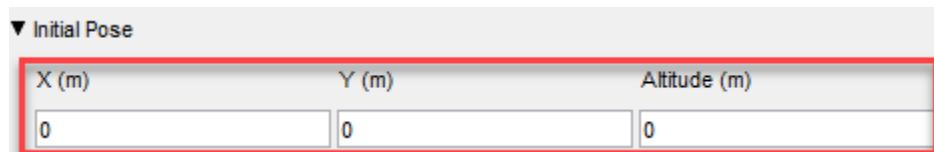
To launch the Tracking Scenario Designer, use the command:

```
trackingScenarioDesigner
```

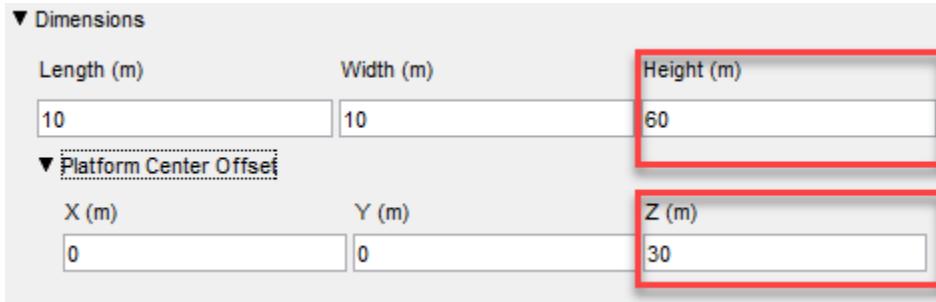
To add a platform in the app, select one platform (tower, for example) from the **PLATFORM** tab and click the **Platform Canvas** to place the platform.



You can change the platform properties through the **Platform Properties** tab. For example, to set the platform center to the origin, set all initial position coordinates to zero in **Initial Pose**.

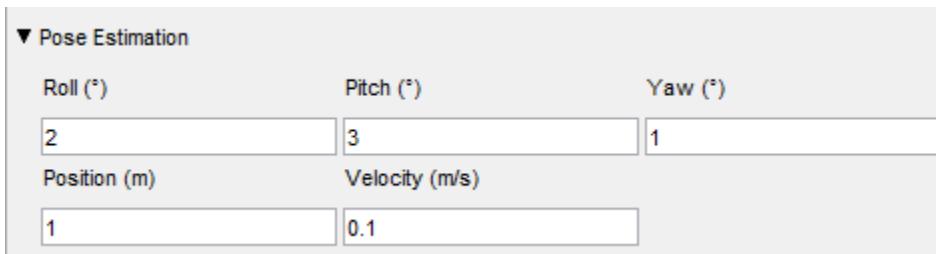


You can also change the **Length**, **Width**, and **Height** of the platform. By default, the **Tower** platform's offset in the z direction is half of the platform height, which places the tower center at its bottom. If the offset is zero, then the platform center collocates with the tower's geometric center.

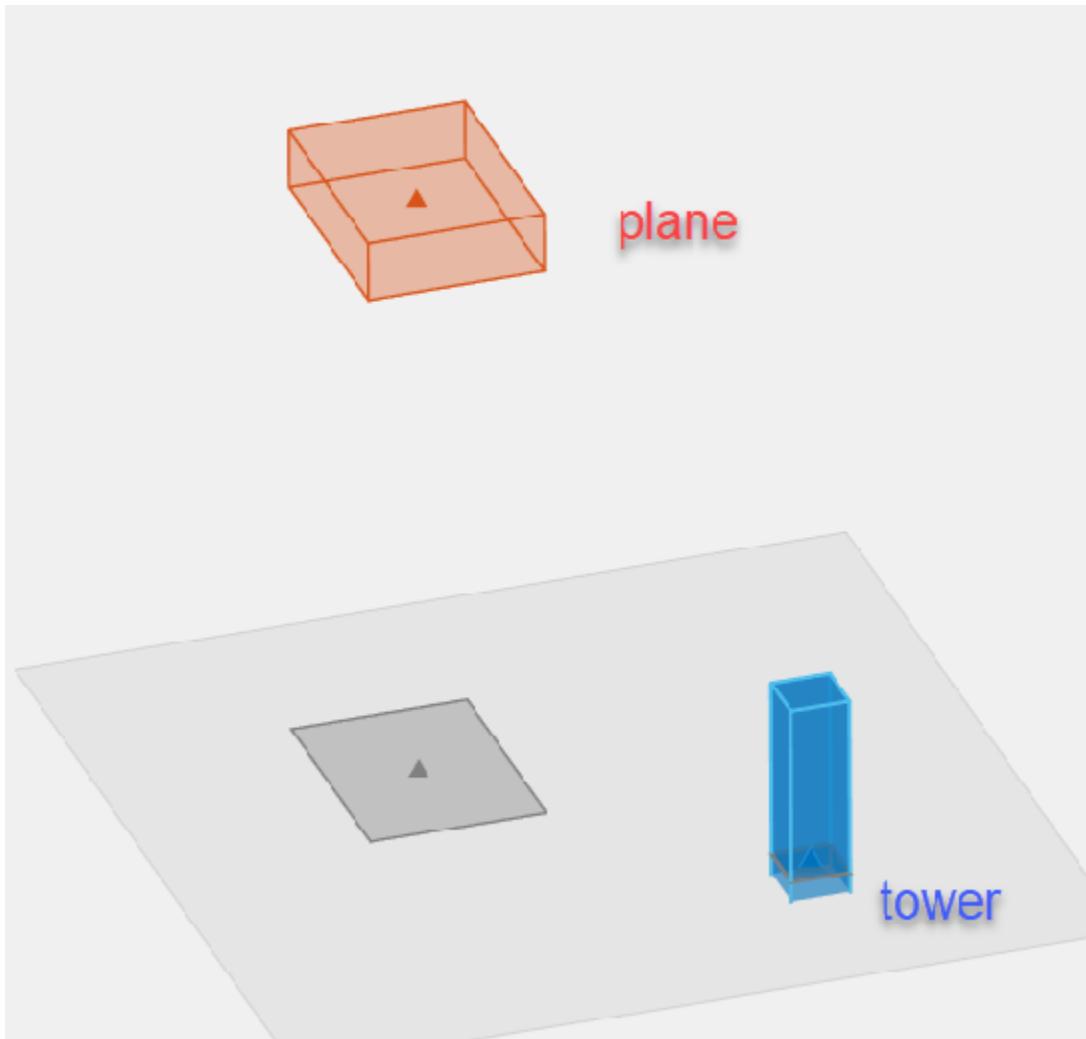


The center offset is defined as the position vector from the geometric center of a platform to the specified center of the platform.

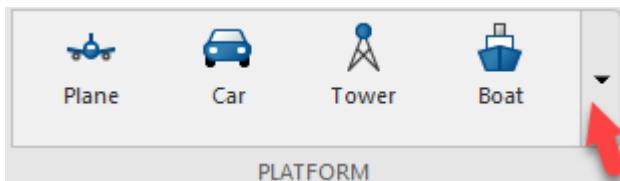
In the app, you can also specify the uncertainty of the estimated platform pose through the **Pose Estimation** tab. The value of each parameter in the tab represents the standard deviation of the corresponding quantity. The standard deviation setup is useful for some practical tracking considerations. For example, the accuracy of a sensor mounted on a tower is impacted if the pose of the tower includes errors. In the app, if you set the standard deviations to be nonzero values for a platform with a mounting sensor, you can observe the inaccuracy of the sensor detections introduced by these standard deviations.



You can also add other platforms in the app. Add a **Plane** platform on the canvas and set its initial position as [50, -50, 100]. You can see the center of the plane (red) is at its geometric center by default.



You can change the default setting of any class (and define new classes) using the **Platform Gallery Editor**, which you can open by clicking the drop-down arrow on the **PLATFORM** tab.



You cannot edit the class of a currently used platform. To delete a platform, select the platform from the drop-down list and click the delete (trash can) icon.



## Set Up Trajectories of Platforms in Tracking Scenario Designer

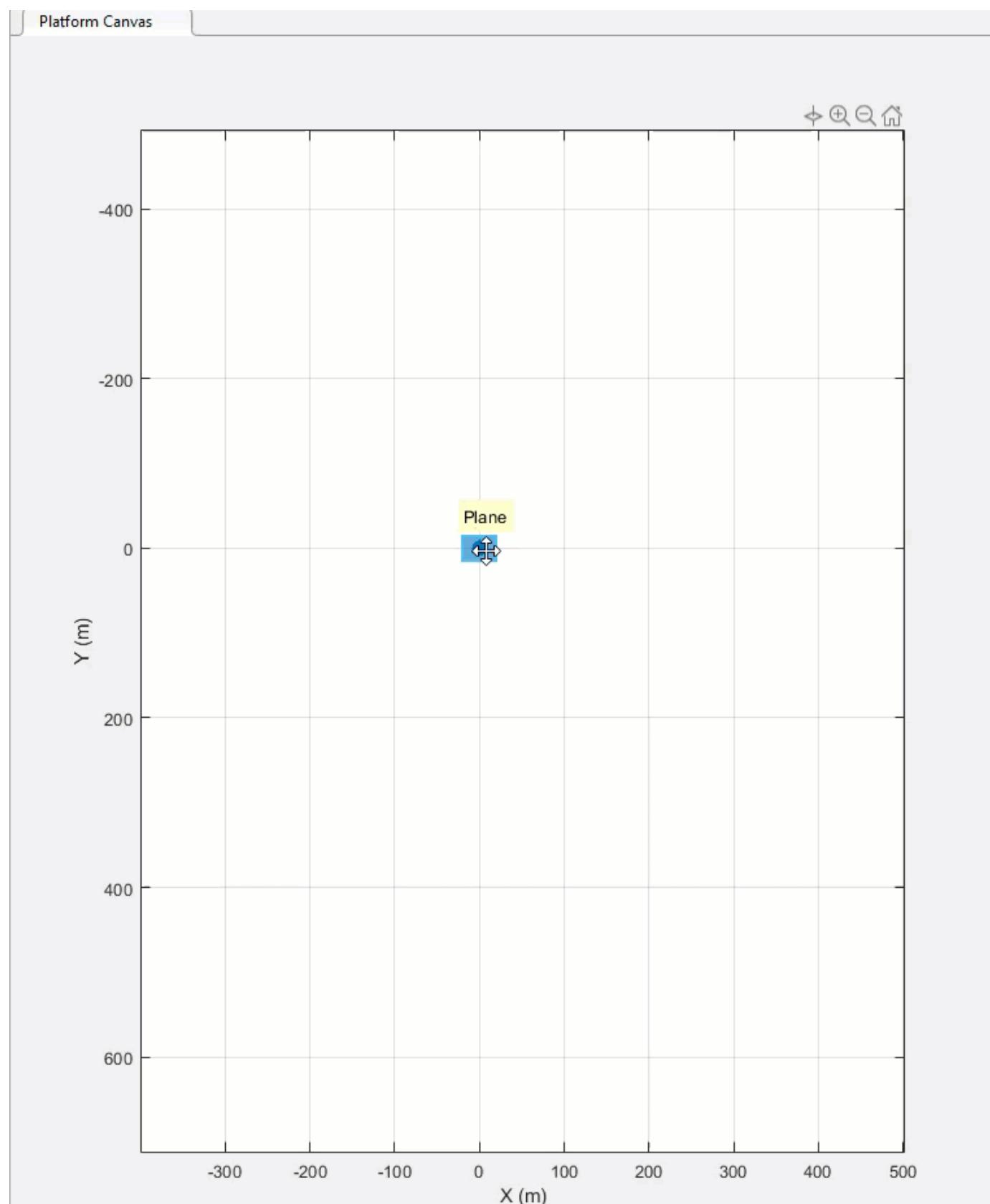
To launch the Tracking Scenario Designer, use the command:

```
trackingScenarioDesigner
```

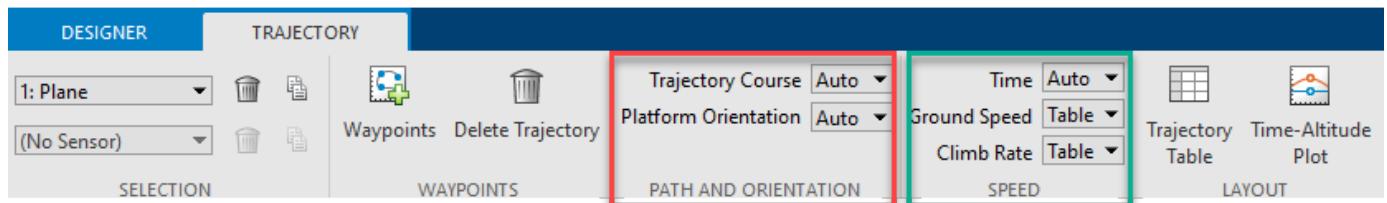
Add a **Plane** platform on the platform canvas and place the plane at [0, 0, 1000] by specifying its initial position through the **Initial Pose** tab as:



Next, add a few waypoints to the platform. Right-click the platform and select **Add Waypoints**, or select the platform and click **Waypoints** on the **TRAJECTORY** toolbar. Then consecutively click the canvas to add waypoints. To end the action, on the keyboard, click **Enter**. You can drag the waypoints to change the trajectory. The specified trajectory represents the trajectory of the platform center defined in the **Platform Center Offset** tab.

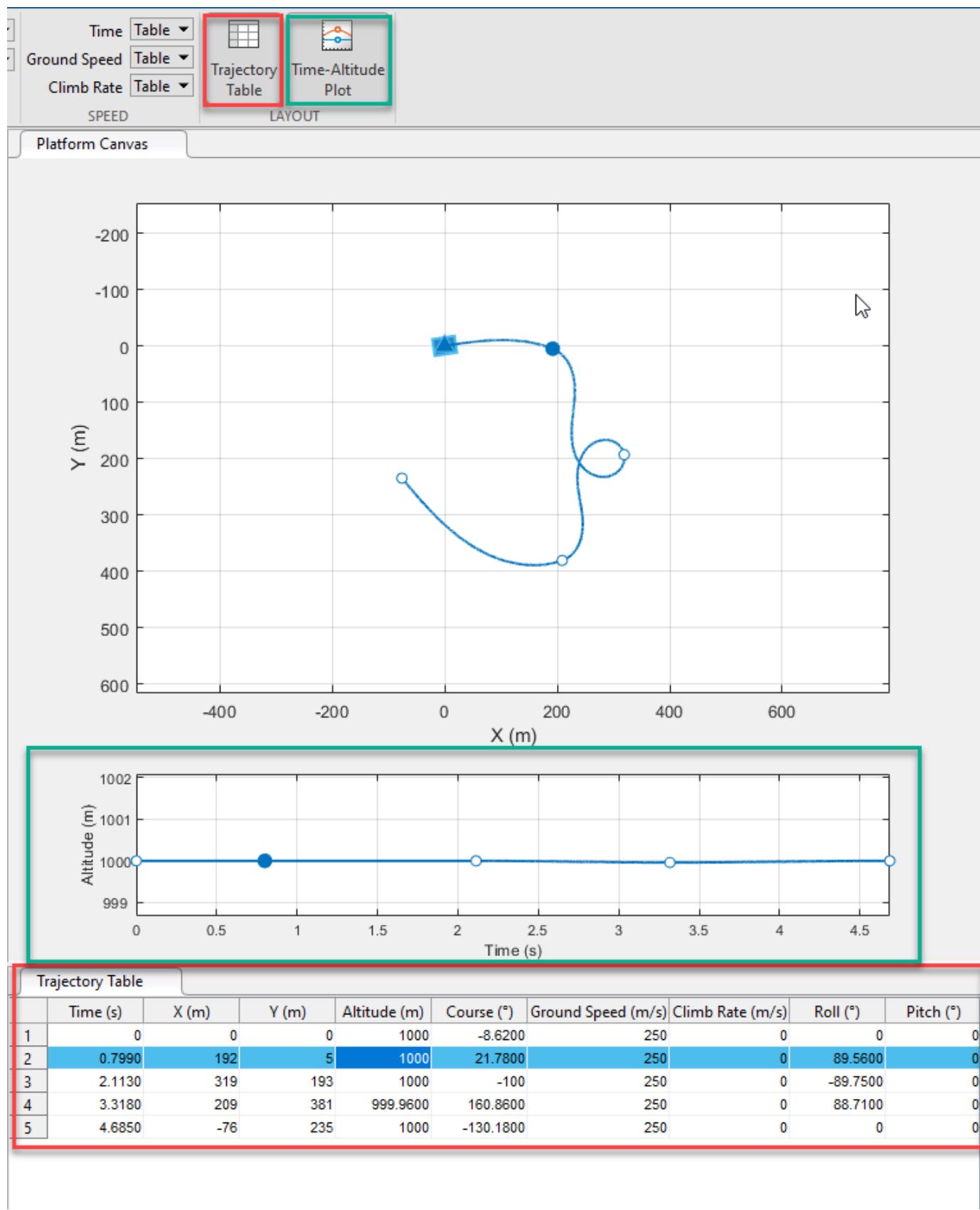


On the **TRAJECTORY** tab, if the **Trajectory Course** and the **Platform Orientation** parameters are set to **auto**, the app calculates the trajectory by fitting a smooth curve including all the waypoints and aligning the platform orientation with the trajectory. With **Time** set to **Auto**, the app calculates the trajectory duration (**Time**) based on the default platform speed, which can be specified through the **PLATFORM Gallery Editor**.



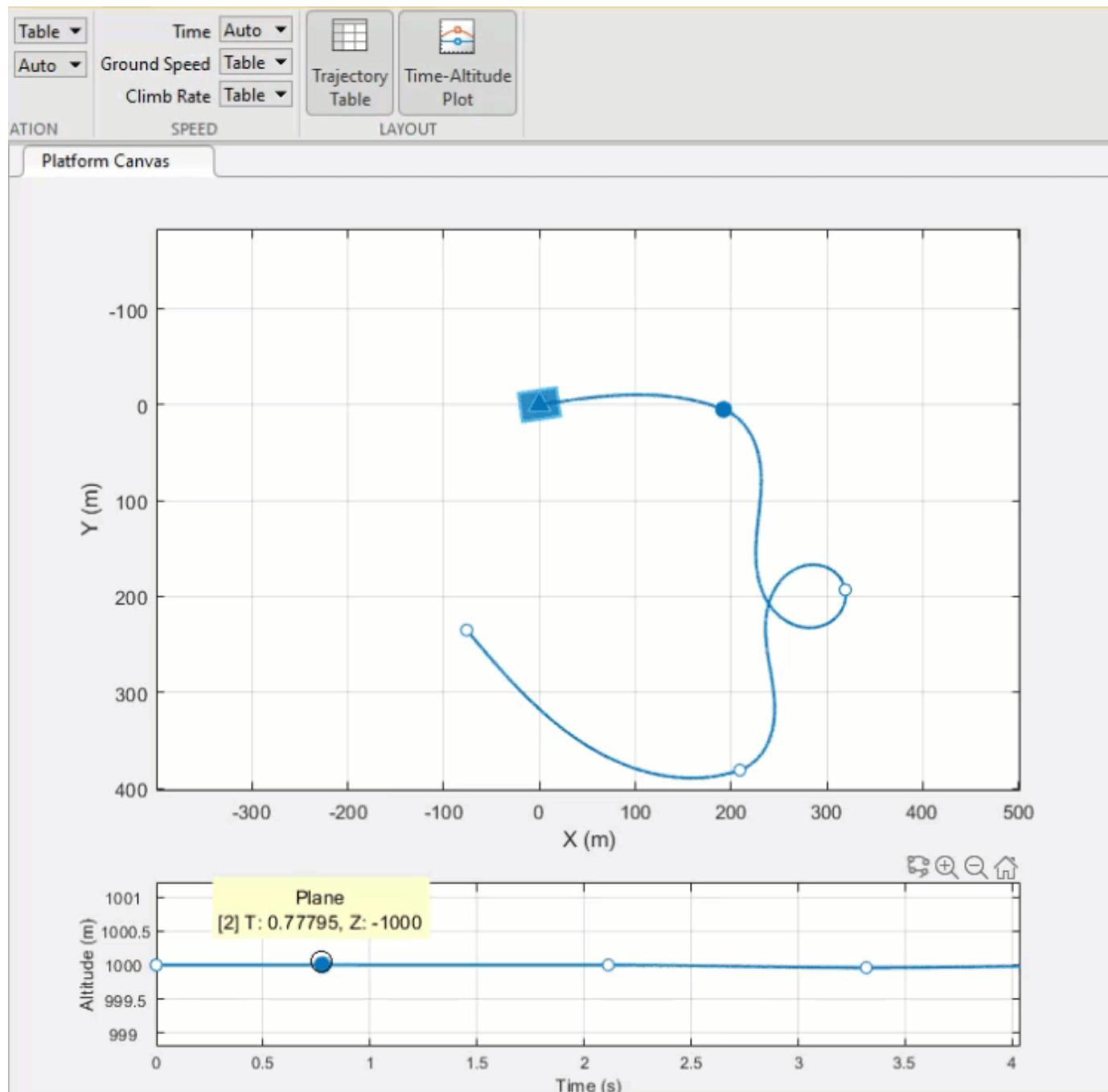
To display the trajectory table below, click **Trajectory Table**. To display the Time-Altitude plot, click **Time-Altitude Plot**.

## 5 Apps

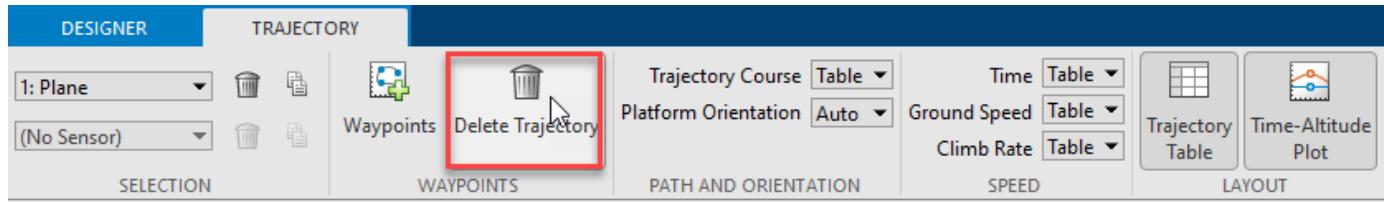


After changing a trajectory parameter selection from **Auto** to **Table**, you can edit the corresponding quantity in the **Trajectory Table**. After you edit the table, observe the change of the trajectory .

You can drag points up and down in altitude in the Time-Altitude plot. After setting **Time** to **Table**, you can drag points forward and backward in time.



To delete a trajectory, select the trajectory and click **Delete Trajectory**.



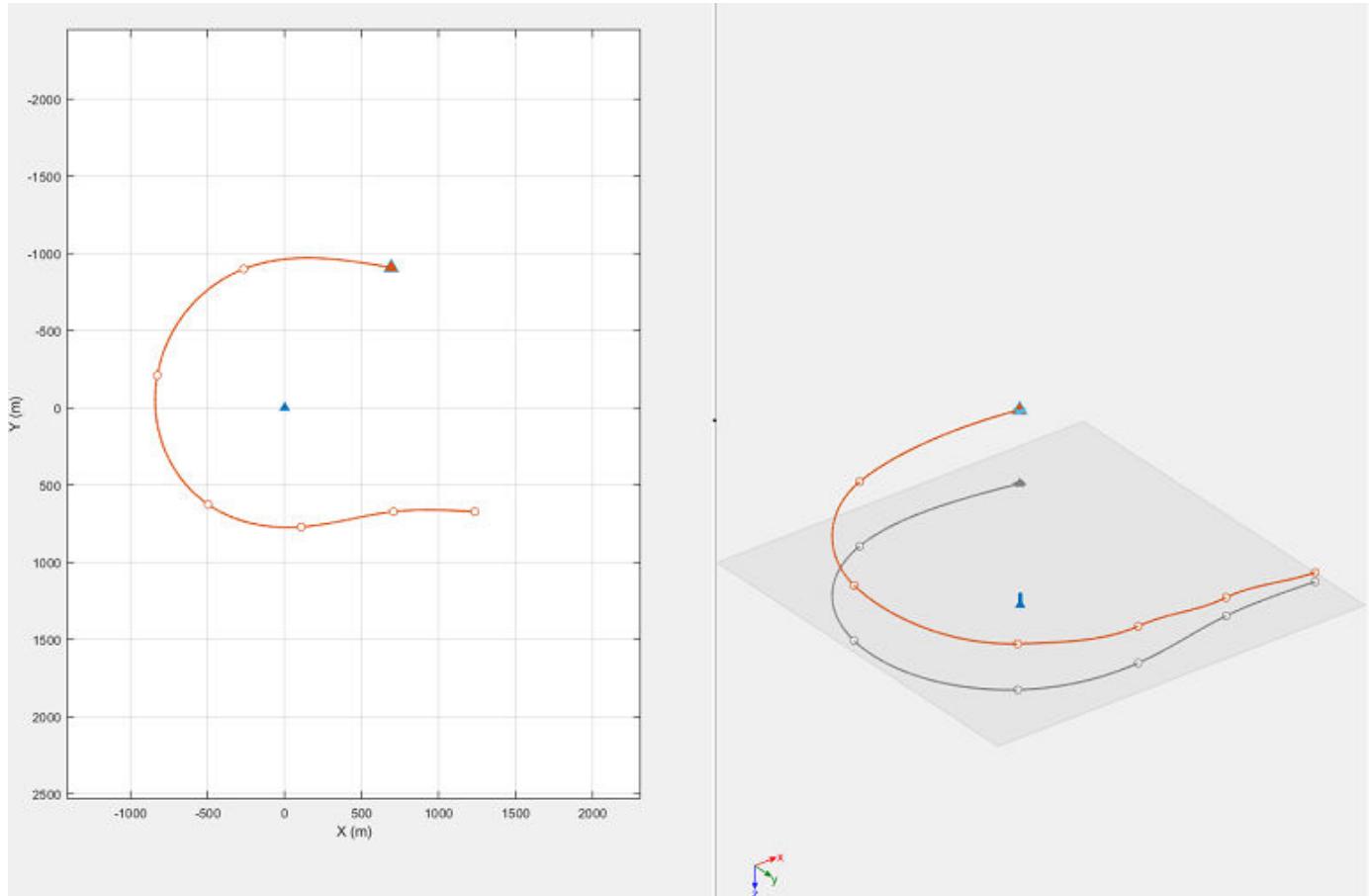
## Set Up Sensors In Tracking Scenario Designer

The MAT-file `TSD_Platforms` was previously saved with a tracking scenario session. To launch the application and load the session file, use the command:

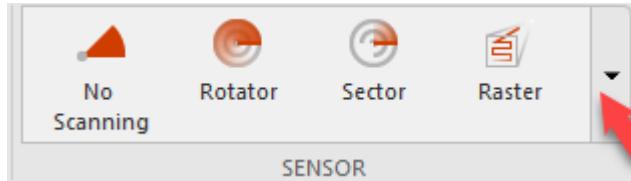
```
trackingScenarioDesigner('TSD_Platforms.mat')
```

The application opens and loads the scenario. The scenario contains two platforms:

- A 60-meter high tower located at the origin of the local NED frame.
- A target traveling at a course speed of 750 m/s around the tower.

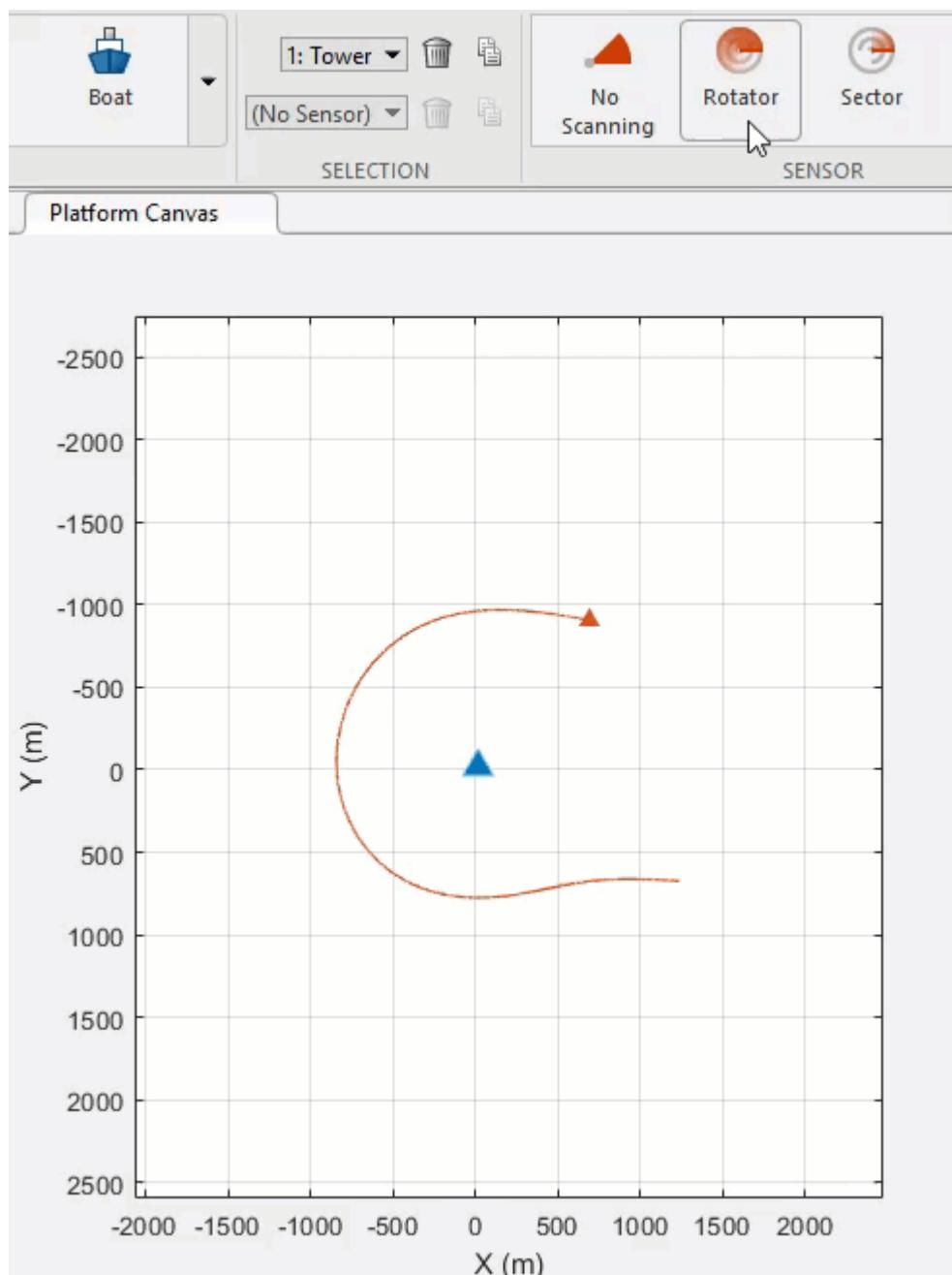


Next, mount a sensor on the top of the tower to monitor its surroundings. There are four predefined classes of sensors available in the app.



You can also click the drop-down arrow to edit the existing classes or add new classes of sensors.

In the app, you select the tower platform, choose a rotator sensor, and place it on the top of the tower. Click the projection button to enable a y-z projection view.

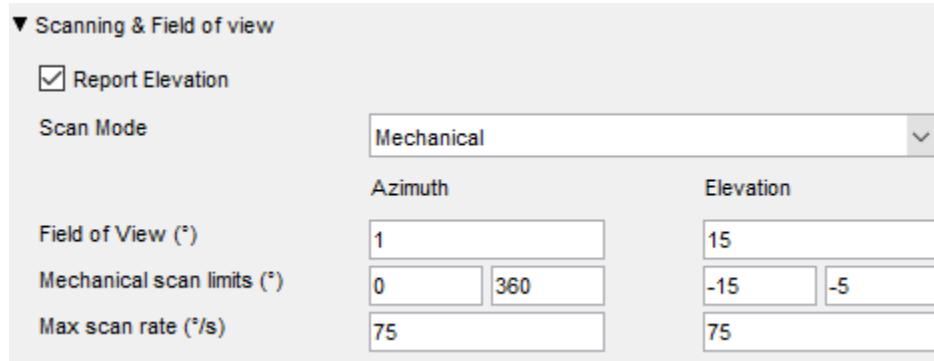


The sensor is positioned at the bottom of the tower by default. To move the sensor to the top of the tower, change its **Mounting Location & Angles**.

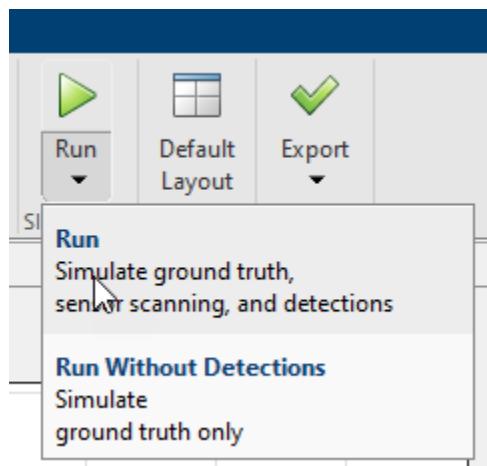
▼ Mounting Location & Angles

X (m)	0	Y (m)	0	Z (m)	-60
Roll (°)	0	Pitch (°)	0	Yaw (°)	0

Enable detection in the elevation by selecting **Report Elevation**. Set the sensor's **Field of View** for **Elevation** to 15 deg to allow a wide coverage region in elevation. Set the **Mechanical scan limits** for **Elevation** to [-15, -5] deg to let the sensor "stare up".



To simulate the tracking scenario and observe the detection of the target generated by the sensor, Click **Run**. (You can also choose **Run Without Detections**.)

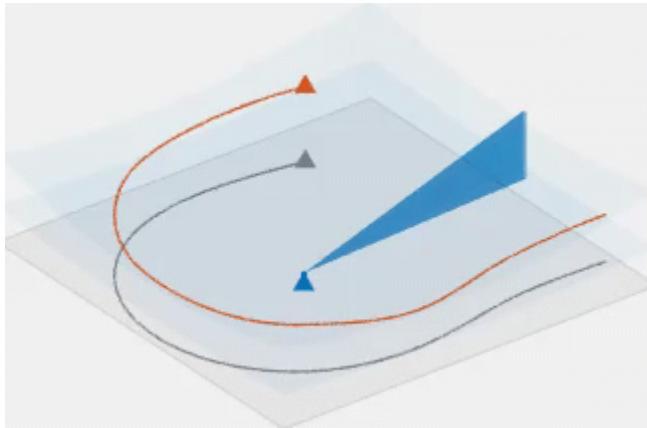


You find that the sensor generates only one detection. You can let the sensor scan faster and generate more detections by adjusting its scan rate using two parameters:

- Update Rate — Determines the number of field of view slices the sensor steps through per second.
- Field of View — Determines the width of each sensor field of view slice or beam.

In the app, increase the **Update Rate** of the sensor to 200 Hz. With the azimuthal field of view set as 1 deg, the resulting scan rate in the azimuth is 200 deg/s, which is above the default **Max scan rate** (75 deg/s). Increase **Max scan rate** to 300 deg/s to allow a high scan rate.

Click **Run** to simulate the scenario again. The sensor now generates multiple sets of detections.



You can also export the script of the scenario by clicking **Export**. Using the exported script, you can modify the scenario programmatically and use the generated scenario to test various tracking algorithms. See “Design and Simulate Tracking Scenario with Tracking Scenario Designer” example for more details on how to modify the generated scenario.

- “Design and Simulate Tracking Scenario with Tracking Scenario Designer”

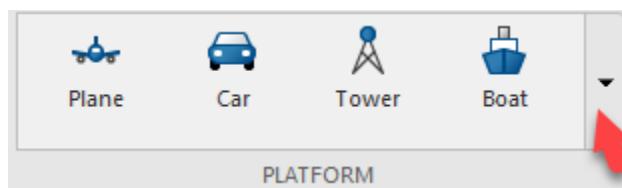
## Parameters

**Platform Properties — Platform properties including dimensions, pose, and RCS tab**

To enable the **Platform Properties** parameters, add at least one platform to the scenario. Then, select a platform from either the **Platform Canvas** or the **Platform Properties** parameter. The parameter values in the **Platform Properties** tab are based on the platform that you select.

Parameter	Description
<b>Current Platform</b>	Currently selected platform, specified as a list of platforms in the scenario.
<b>Name</b>	Name of platform, specified as a string.
<b>Class</b>	Platform class, specified as <b>Plane</b> , <b>Tower</b> , <b>Car</b> , or <b>Boat</b> .

You can change the default settings (such as **Speed**) of the four platform classes and add new platform classes using the **Platform Gallery Editor**. You can open the editor by clicking the drop-down arrow on the **PLATFORM** tab and selecting **Add/Edit Platform Gallery**.



**Dimensions — Platform dimensions**  
tab

Platform dimensions, specified as **Length**, **Width**, and **Height** in meters.

Parameter	Description
<b>Length (m)</b>	Length of platform, specified as a nonnegative scalar in meters.
<b>Width (m)</b>	Width of platform, specified as a nonnegative scalar in meters.
<b>Height (m)</b>	Height of platform, specified as a nonnegative scalar in meters.

You can also specify the **Platform Center Offset** using the **X**, **Y**, and **Z** offsets. The offset is measured from the geometric center of the platform to the specified center.

Parameter	Description
<b>X (m)</b>	Offset in the x-direction, specified as a scalar in meters.
<b>Y (m)</b>	Offset in the y-direction, specified as a scalar in meters.
<b>Z (m)</b>	Offset in the z-direction, specified as a scalar in meters.

#### **Initial Pose — Initial position and orientation of platform**

tab

Initial position and orientation of platform, specified by three position coordinates **X**, **Y**, and **Altitude** in meters and three rotational angles **Roll**, **Pitch**, and **Yaw** in degrees.

Parameter	Description
<b>X (m)</b>	Initial x coordinate of the platform center in the scenario frame, specified as a scalar in meters.
<b>Y (m)</b>	Initial y coordinate of the platform center in the scenario frame, specified as a scalar in meters.
<b>Altitude (m)</b>	Initial altitude of the platform center in the scenario frame, specified as a scalar in meters.
<b>Roll (°)</b>	Orientation angle of the platform about the x-axis of the scenario frame, specified as a scalar in degrees.
<b>Pitch (°)</b>	Orientation angle of the platform about the y-axis of the scenario frame, specified as a scalar in degrees.
<b>Yaw (°)</b>	Orientation angle of the platform about the z-axis of the scenario frame, specified as a scalar in degrees.

#### **Pose Estimation — Accuracy of platform pose estimation**

tab

Accuracy of the platform pose estimation, specified as standard deviations for three rotational angles : **Roll**, **Pitch**, and **Yaw**, and two translational motion quantities : **Position** and **Velocity**.

When the standard deviation value of any motion quantity is specified as nonzero, the platform pose contains errors corresponding to that motion quantity.

Parameter	Description
<b>Roll (°)</b>	Standard deviation of the roll angle of the platform, specified as a scalar in degrees.
<b>Pitch (°)</b>	Standard deviation of the pitch angle of the platform, specified as a scalar in degrees.
<b>Yaw (°)</b>	Standard deviation of the yaw angle of the platform, specified as a scalar in degrees.
<b>Position (m)</b>	Standard deviation of position coordinates of the platform, specified as a scalar in degrees.
<b>Velocity (m)</b>	Standard deviation of velocity coordinates of the platform, specified as a scalar in degrees.

#### Radar Cross Section — Radar cross section information tab

Radar cross section information, including RCS pattern information and RCS Viewer specifications. You can specify a constant RCS pattern as a scalar in dBsm, or you can import RCS information through the **Import Signature** window after selecting the **Import RCS** tab.

Parameter	Description
<b>Constant RCS Pattern</b>	RCS pattern, specified as a positive constant in dBsm.
<b>Import RCS</b>	Import RCS pattern through an <b>Import Signature</b> window.

You can also specify the **RCS Viewer** by changing the **Elevation Cut** in degrees and the **Frequency Cut** in Hz.

Parameter	Description
<b>Elevation Cut</b>	Elevation cut of RCS viewer, specified as a scalar in degrees.
<b>Frequency Cut</b>	Frequency cut of RCS viewer, specified as a scalar in Hz.

#### Sensor Properties — Sensor properties including sensor mounting, scanning settings, and detection settings tab

To enable the **Sensor Properties** parameters, add at least one sensor to the platform. Then, select a sensor from either the **Sensor Canvas** or the **Sensor Properties** tab. The parameter values in the **Sensor Properties** tab are based on the platform and sensor that you select.

Parameter	Description
<b>Current Platform</b>	Current platform on which the sensor is mounted, specified as a list of platforms in the scenario.

Parameter	Description
<b>Current Sensor</b>	Currently selected sensor, specified as a list of sensors in the scenario.
<b>Name</b>	Name of sensor, specified as a string.
<b>Update Rate</b>	Sensor update rate, specified as a positive scalar in Hz.
<b>Type</b>	Sensor type, specified as: <ul style="list-style-type: none"><li>• Sector Monostatic Radar</li><li>• No Scanning Monostatic Radar</li><li>• Rotator Monostatic Radar</li><li>• Raster Monostatic Radar</li></ul>

#### Mounting Location & Angles — Sensor mounting location and angles tab

Sensor mounting location and angles on the platform, specified by three position coordinates **X**, **Y**, and **Z** in meters and three rotational angles **Roll**, **Pitch**, and **Yaw** in degrees.

Parameter	Description
<b>X (m)</b>	x coordinate of the sensor on the platform frame, specified as a scalar in meters.
<b>Y (m)</b>	y coordinate of the sensor on the platform frame, specified as a scalar in meters.
<b>Z (m)</b>	z coordinate of the sensor on the platform frame, specified as a scalar in meters.
<b>Roll (°)</b>	Orientation angle of the sensor about the x-axis of the platform frame, specified as a scalar in degrees.
<b>Pitch (°)</b>	Orientation angle of the sensor about the y-axis of the platform frame, specified as a scalar in degrees.
<b>Yaw (°)</b>	Orientation angle of the sensor about the z-axis of the platform frame, specified as a scalar in degrees.

#### Scanning & Field of view — Scanning and field of view of sensor tab

Parameter	Description
<b>Report Elevation</b>	Enable sensor reporting elevation information, specified as <b>on</b> or <b>off</b> .
<b>Scan Mode</b>	Mode of sensor scanning, selected as <b>Mechanical</b> , <b>Electric</b> , or <b>Mechanical and electric</b> .

Parameter	Description
<b>Field of View (°)</b>	Field of view of the sensor, specified as two nonnegative scalars representing <b>Azimuth</b> and <b>Elevation</b> in degrees.
<b>Mechanical scan limits (°)</b>	Upper and lower limits of mechanical scan, specified as two scalars for <b>Azimuth</b> in degrees. If <b>Report Elevation</b> is enabled, you can specify the scan limits for <b>Elevation</b> in degrees. To enable this parameter, set the <b>Scan Mode</b> to Mechanical or Mechanical and electric.
<b>Electronic scan limits (°)</b>	Upper and lower limits of electronic scan, specified as two scalars for <b>Azimuth</b> in degrees. If <b>Report Elevation</b> is enabled, you can specify the scan limits for <b>Elevation</b> in degrees . To enable this parameter, set the <b>Scan Mode</b> to Electric or Mechanical and electric.
<b>Max scan rate (°/s)</b>	Maximum scan rate, specified as a scalar for <b>Azimuth</b> in degrees per second. If <b>Report Elevation</b> is enabled, you can specify the maximum scan rate for <b>Elevation</b> in degrees per second. If the specified scan rate ( <b>Update Rate * Field of View</b> ) is larger than the <b>Max scan rate</b> , the sensor scan rate is truncated at the <b>Max scan rate</b> . To enable this parameter, set the <b>Scan Mode</b> to Mechanical or Mechanical and electric.

#### Detections Settings — Detections Settings tab

Detection settings of the sensor, specified by using detections probability, false alarm rate, reference range, and reference RCS.

Parameter	Description
<b>Detection Probability</b>	Probability of sensor successfully detecting a target, specified as a scalar in [0,1]. This quantity defines the probability of detecting a target with a radar cross-section larger than the <b>Reference RCS</b> and within the <b>Reference Range</b> of the sensor.
<b>False Alarm Rate</b>	Probability of sensor making a false detection in each sensor resolution cell, specified as a scalar in [1e-7,1e-3].

Parameter	Description
<b>Reference Range (m)</b>	Reference range for the given <b>Detection Probability</b> and the given <b>Reference RCS</b> , specified as a positive scalar in meters.
<b>Reference RCS (dBsm)</b>	Reference radar cross-section (RCS) for the given <b>Detection Probability</b> and the given <b>Reference Range</b> , specified as a scalar in dBsm.

### Advanced Settings – Advance settings

tab

Advanced settings of the sensor are listed in this table.

Parameter	Description
<b>Max Number of Detections</b>	Maximum number of detections reported by the sensor, specified as a positive integer.
<b>Report False Alarm</b>	Enable the sensor to model and report false alarms, specified as <b>on</b> or <b>off</b> . When specified as <b>off</b> , the sensor does not generate any false detection.
<b>Report Range Rate</b>	Enable the radar to measure and report target range rates, specified as <b>on</b> or <b>off</b> .
<b>Model Target Occlusion</b>	Enable occlusion of objects from extended objects, specified as <b>on</b> or <b>off</b> . Turn off this option to disable occlusion of extended objects.
<b>Model Range Ambiguity</b>	Enable range ambiguities, specified as <b>on</b> or <b>off</b> . When specified as <b>off</b> , the sensor cannot resolve range ambiguities and target ranges beyond the <b>Max Unambiguous Range</b> are wrapped into the interval [0, <b>MaxUnambiguousRange</b> ]. When <b>false</b> , targets are reported at their unambiguous range.
<b>Model Range Rate Ambiguity</b>	Enable range-rate ambiguities, specified as <b>on</b> or <b>off</b> . Turn on this option to enable range-rate ambiguities by the sensor. When true, the sensor does not resolve range rate ambiguities and target range rates beyond the <b>Max Unambiguous Radial Speed</b> are wrapped into the interval [- <b>MaxUnambiguousRadialSpeed</b> , <b>MaxUnambiguousRadialSpeed</b> ]. When <b>false</b> , targets are reported at their unambiguous range rate.  To enable this parameter, set <b>Report Range Rate</b> to <b>on</b> .

Parameter	Description
<b>Max Unambiguous Range (m)</b>	Maximum unambiguous range, specified as a positive scalar. Maximum unambiguous range defines the maximum range for which the radar can unambiguously resolve the range of a target.
<b>Max Unambiguous Radial Speed (m/s)</b>	Maximum unambiguous radial speed, specified as a positive scalar. 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.  To enable this parameter, set <b>Report Range Rate</b> to on.

#### Accuracy & Noise — Accuracy and noise settings

tab

The accuracy and noise setting of the sensor are listed in this table.

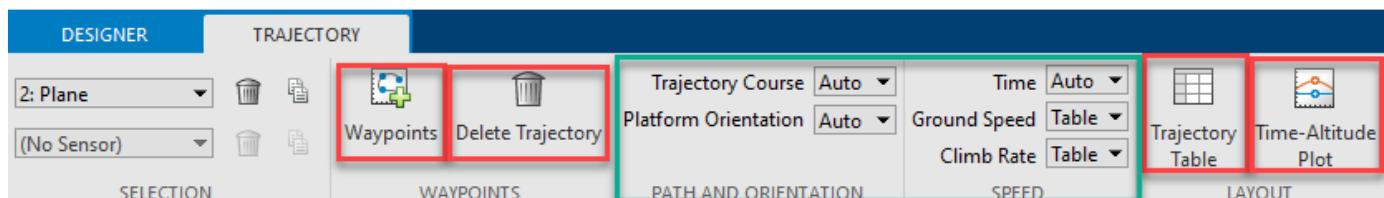
Parameter	Description
<b>Azimuth (°)</b>	Azimuth resolution and bias, specified as two nonnegative scalars: <ul style="list-style-type: none"> <li>• Azimuth resolution defines the minimum separation in azimuth angle at which the radar can distinguish two targets.</li> <li>• Azimuth bias is expressed as a fraction of the azimuth resolution. This value sets a lower bound on the azimuthal accuracy of the sensor.</li> </ul>
<b>Elevation (°)</b>	Elevation resolution and bias, specified as two nonnegative scalars: <ul style="list-style-type: none"> <li>• Elevation resolution defines the minimum separation in elevation angle at which the radar can distinguish two targets.</li> <li>• Elevation bias is expressed as a fraction of the azimuth resolution. This value sets a lower bound on the elevation accuracy of the sensor.</li> </ul> To enable this parameter, turn on <b>Report Elevation</b> .

Parameter	Description
<b>Range (m)</b>	Range resolution and bias, specified as two nonnegative scalars: <ul style="list-style-type: none"> <li>• Range resolution defines the minimum separation in range at which the radar can distinguish between two targets.</li> <li>• Range bias is expressed as a fraction of the range resolution. This value sets a lower bound on the range accuracy of the radar.</li> </ul>
<b>Range Rate (m/s)</b>	Range rate resolution and bias, specified as two nonnegative scalars: <ul style="list-style-type: none"> <li>• Range rate resolution defines the minimum separation in range rate at which the radar can distinguish between two targets.</li> <li>• Range rate bias is expressed as a fraction of the range rate resolution. This value sets a lower bound on the range rate accuracy of the radar.</li> </ul>
<b>Add noise to measurements</b>	Add measurement noise in the detections, specified as on or off.

### TRAJECTORY — Trajectory settings

tab on toolbar

To edit the trajectory and control the trajectory generation, use the trajectory settings.



- Click **Waypoints** to add waypoints to a selected platform.
- Click **Delete Trajectory** to delete an existing trajectory.
- Click **Trajectory Table** to display the trajectory table.
- Click **Time-Altitude plot** to display the time vs altitude plot.

You can also choose to automatically generate the waypoint trajectory or manually input waypoints by changing the selections of the **PATH AND ORIENTATION** and the **SPEED** parameters.

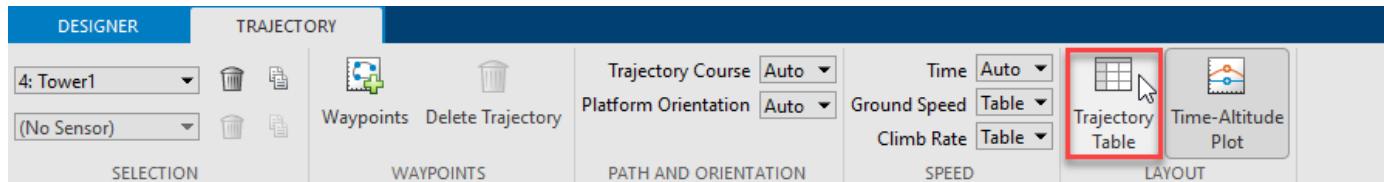
Parameter	Selection
<b>Trajectory Course</b>	<ul style="list-style-type: none"> <li>• <b>Auto:</b> When selected, the app generates the course by fitting all the waypoints with a smooth curve.</li> <li>• <b>Table:</b> When selected, you can manually edit the trajectory course at each waypoint using the <b>Trajectory Table</b>.</li> </ul>

Parameter	Selection
<b>Platform Orientation</b>	<ul style="list-style-type: none"> <li><b>Auto:</b> When selected, the app calculates the yaw and pitch angles of the platform to align the platform with the trajectory and calculates the roll angle to cancel the centripetal acceleration.</li> <li><b>Table:</b> When selected, you can manually edit the yaw, pitch, and roll angles at each waypoint using the <b>Trajectory Table</b>.</li> </ul>
<b>Time</b>	<ul style="list-style-type: none"> <li><b>Auto:</b> When selected, the app calculates the visiting time at all the waypoints.</li> <li><b>Table:</b> When selected, you can manually edit the visiting time at each waypoint using the <b>Trajectory Table</b>.</li> </ul>
<b>Ground speed</b>	<ul style="list-style-type: none"> <li><b>Auto:</b> When selected, the app uses the default ground speed for each platform class at each waypoint.</li> <li><b>Table:</b> When selected, you can manually edit the ground speed at each waypoint using the <b>Trajectory Table</b>.</li> </ul>
<b>Climb Rate</b>	<ul style="list-style-type: none"> <li><b>Auto:</b> When selected, the app calculates the climb rate at each waypoint to smoothly fit all the waypoints.</li> <li><b>Table:</b> When selected, you can manually edit the climb rate at each waypoint using the <b>Trajectory Table</b>.</li> </ul>

### Trajectory Table – Trajectory information

table

Trajectory information for each waypoint, specified as a table of scalars. When you insert waypoints on the platform canvas, the table is automatically generated . Click **Trajectory Table** under the **Trajectory** tab to display the table.



After you change the parameter values in the table, the platform trajectory changes accordingly on the canvas. The table includes these trajectory parameters.

Parameter	Description
<b>Times (s)</b>	Time at which the platform visits the waypoint, specified as a scalar in seconds.
<b>X (m)</b>	x coordinate of the waypoint in the scenario navigation frame.
<b>Y (m)</b>	y coordinate of the waypoint in the scenario navigation frame.
<b>Altitude (m)</b>	Altitude of the platform waypoint in the scenario navigation frame.

Parameter	Description
<b>Course (°)</b>	The direction of motion on the x-y plane, specified as an angle measurement from the x direction.
<b>Ground speed (m/s)</b>	Magnitude of the projected velocity on the x-y plane, specified as a scalar in meters.
<b>Climb Rate (m/s)</b>	Climb rate of the waypoint, which is the projection of the platform velocity in the z direction.
<b>Roll (°)</b>	Orientation angle of the platform about the x-axis of the scenario frame, in degrees, specified as a scalar.
<b>Pitch (°)</b>	Orientation angle of the platform about the y-axis of the scenario frame, in degrees, specified as a scalar.
<b>Yaw (°)</b>	Orientation angle of the platform about the z-axis of the scenario frame, in degrees, specified as a scalar.

## Programmatic Use

The `trackingScenarioDesigner` command opens the **Tracking Scenario Designer** app.

The `trackingScenarioDesigner(scenarioFileName)` command opens the app and loads the specified scenario MAT-file into the app. This file must be a tracking scenario file saved from the app.

If the scenario file is not in the current folder or not in a folder on the MATLAB path, specify the full path name. For example:

```
drivingScenarioDesigner('C:\Desktop\myTrackingScenario.mat');
```

You can also load prebuilt scenario files. Before loading a prebuilt scenario, add the folder containing the scenario to the MATLAB path.

The `trackingScenarioDesigner(scenario)` command opens the app and loads the specified `trackingScenario` object, `scenario`, into the app with the following limitations:

- The `IsEarthCentered` property of the scenario must set to `false`.
- The app ignores the `StopTime` and `UpdateRate` properties of the scenario.
- The `ClassID` property value of any platform in the scenario must be equal to one of the default Class ID values in the app.
- The `PlatformID` property values of all platforms in the scenario are renumbered in a numeric sequence in the app.
- The app only supports the `monostaticRadarSensor` object and ignores other sensor objects (such as the `irSensor` object and the `sonarSensor` object) in the scenario. The app also ignores all emitter objects (such as the `radarEmitter` object).

## Tips

- The app uses the NED frame as the default coordinate frame, in which a platform with positive altitude has negative z coordinate.
- You can undo (press **Ctrl+Z**) and redo (press **Ctrl+Y**) changes you make on the scenario and sensor canvases. For example, you can use these keyboard shortcuts to delete a recently placed road center or redo the movement of a radar sensor.
- You can use the **Space** bar on the keyboard to reset the Platform Canvas to a view containing all platforms and trajectories.
- You can use the **Enter** and **Esc** keys on the keyboard to accept and cancel a waypoint, respectively.

## See Also

### Objects

`Platform` | `monostaticRadarSensor` | `radarSensor` | `rcsSignature` | `trackingScenario` | `waypointTrajectory`

### Topics

“Design and Simulate Tracking Scenario with Tracking Scenario Designer”

### Introduced in R2020a