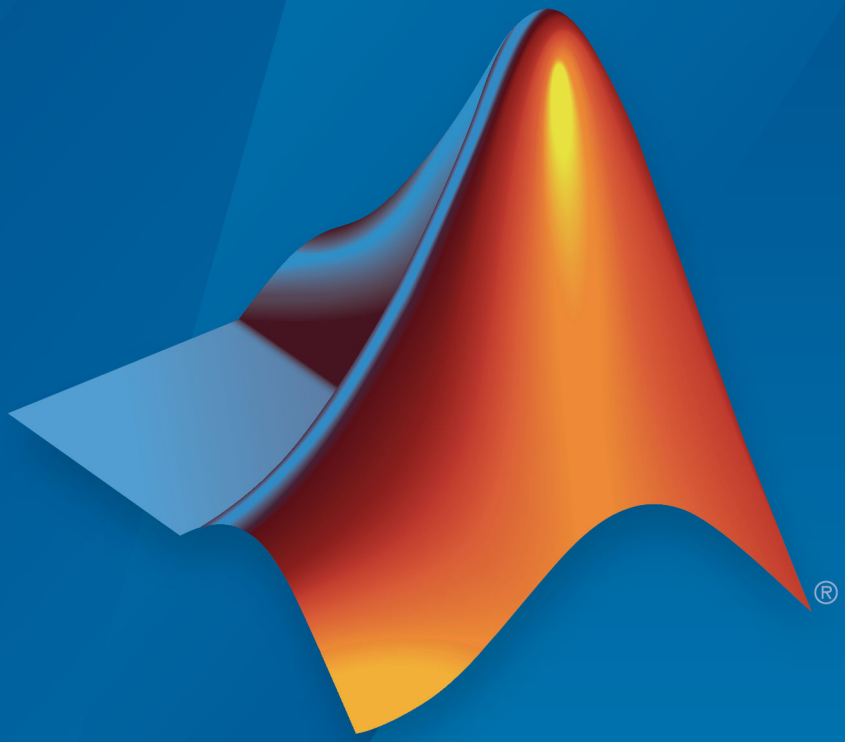
Sensor Fusion and Tracking Toolbox™

Reference



MATLAB® & SIMULINK®

R2019a

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Sensor Fusion and Tracking Toolbox™ Reference Guide

Revision History

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

Contents

1 Functions in Sensor Fusion and Tracking Toolbox

2 Classes in Sensor Fusion and Tracking Toolbox

3 System Objects in Sensor Fusion and Tracking Toolbox

4 Blocks in Sensor Fusion and Tracking Toolbox

5 Apps in Sensor Fusion and Tracking Toolbox

1

Functions in Sensor Fusion and Tracking Toolbox

allanvar

Allan variance

Syntax

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

Description

`[avar,tau] = allanvar(omega)` returns the Allan variance `avar` as a function of sample length `tau`. If `omega` is specified as a matrix, `allanvar` operates over the columns of `omega`.

`[avar,tau] = allanvar(omega,m)` returns the Allan variance for specific values of τ, defined by `m`.

`[avar,tau] = allanvar(omega,ptStr)` sets `tau` to the specified `ptStr`.

`[avar,tau] = allanvar(___,fs)` specifies the sample rate of `omega` at `fs` Hz. This input parameter can be used with any of the previous syntaxes.

Examples

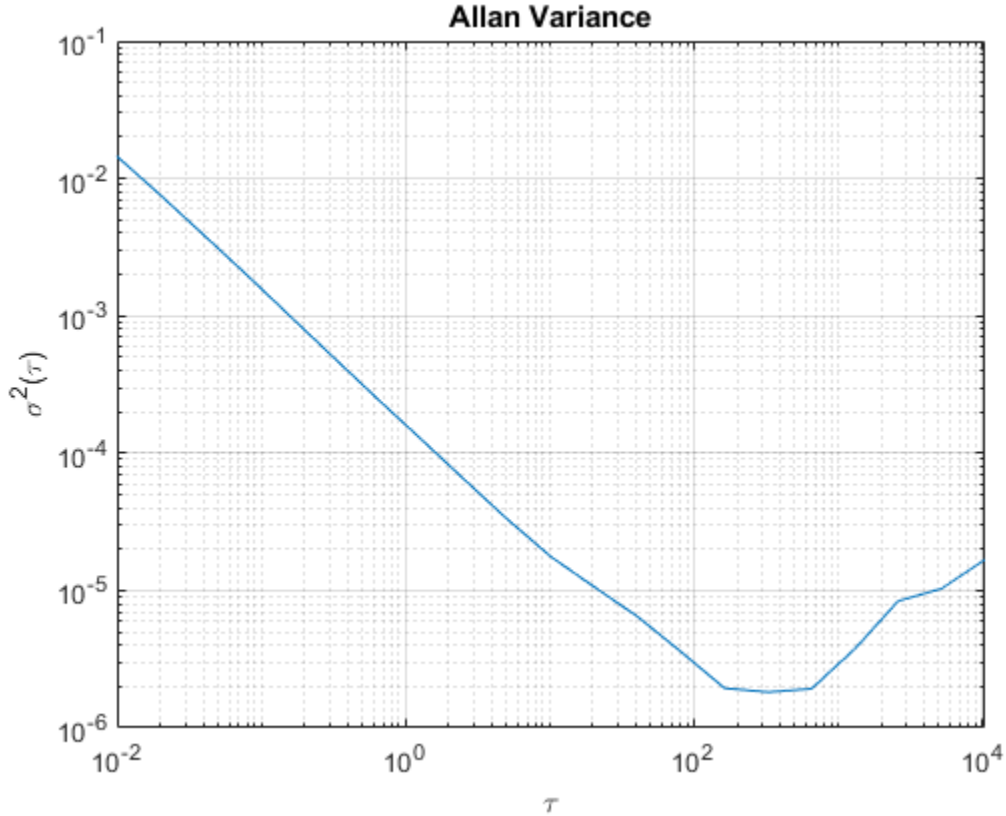
Determine Allan Variance of Single Axis Gyroscope

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 τ

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

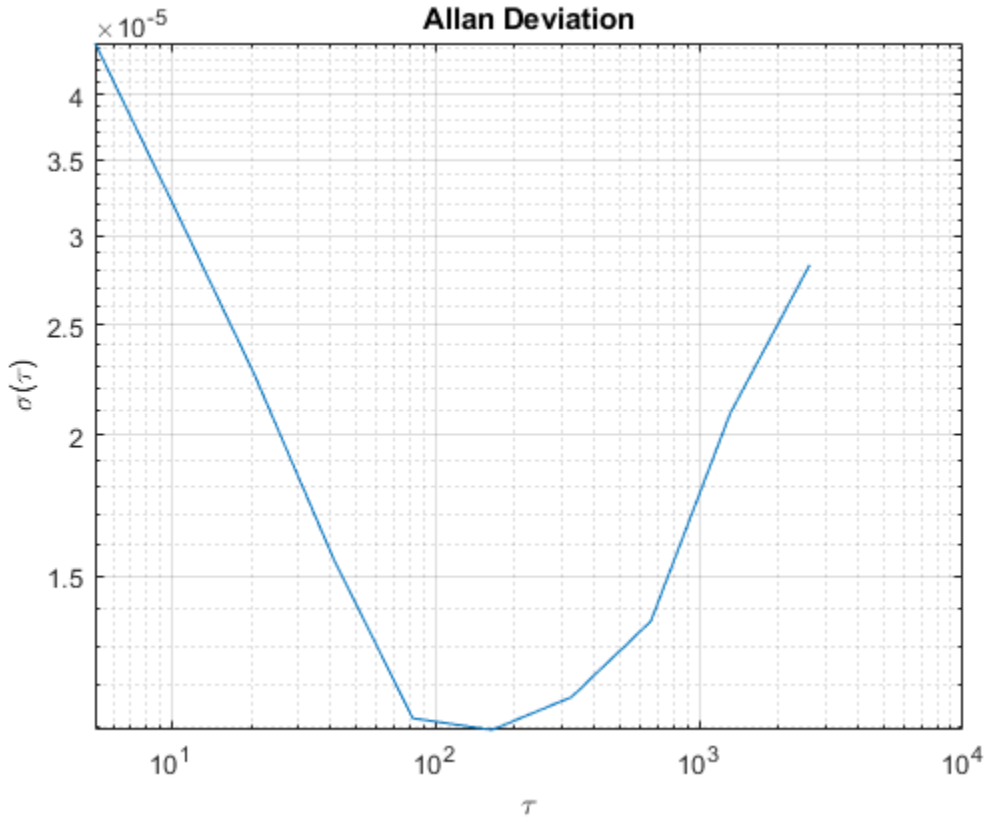
```
numSamples = 1e6;
Fs = 100;
nStd = 1e-3;
kStd = 1e-7;
nNoise = nStd.*randn(numSamples,1);
kNoise = kStd.*cumsum(randn(numSamples,1));
omega = nNoise+kNoise;
```

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 array
vector | matrix | multidimensional array

Input array specified as a vector, matrix, or multidimensional array. If specified as a matrix, `allanvar` operates over the columns of `omega`.

Data Types: `single` | `double`

`m` — Specific values of τ for calculating Allan variance
scalar | vector

Specific values of τ at which to calculate Allan variance, specified as a scalar or vector with ascending integer values less than *(N-1)/2*, where N is the number of elements in `omega`. `m` is not the index of `omega`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ptStr` — Point specification
`'octave'` (default) | `'decade'`

Point specification of `tau`, specified as:

- `'octave'`

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

- `'decade'`

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

N is the number of samples in `omega`.

`fs` — Sample rate in Hz
scalar

Sample rate of input array in Hz, specified as a positive scalar.

Data Types: `single` | `double`

Output Arguments

`avar` — Allan variance of input array
vector | matrix | multidimensional array

Allan variance of input array at `tau`, specified as a vector, matrix, or multidimensional array.

`tau` — Input values of Allan variance
vector | matrix | multidimensional array

Sample lengths corresponding to values of output. Specify this parameter by octave or decade using `ptStr`.

Algorithms

Allan variance can be used to measure the frequency stability of precision oscillators or identify sources of noise in gyroscope data. Consider N samples of data from a gyroscope with a sample time of τ_0. Form data clusters of durationsτ_0, $2\tau_0$, ..., $m\tau_0$, where m is less than half of N. Then, obtain the averages of the sum of the data points contained in each cluster over the length of the cluster. The Allan variance is defined as the two-sample variance of the data cluster averages as a function of cluster time. This function calculates overlapping Allan variance.

See Also

`gyroparams` | `imuSensor`

Introduced in R2019a

correctjpda

Correct state and state estimation error covariance using JPDA

Syntax

```
[x_corr,P_corr] = correctjpda(filterObj,z,jp)
[x_corr,P_corr] = correctjpda(filterObj,z,jp,varargin)
[x_corr,P_corr] = correctjpda(filterObj,z,jp,zcov)
[x_corr,P_corr,z_corr] = correctjpda(filterObj,z,jp)
```

Description

`[x_corr,P_corr] = correctjpda(filterObj,z,jp)` returns the correction of state, `x_corr`, and state estimate error covariance, `P_corr`, using a set of measurements `z` and their joint probabilistic data association coefficients `jp`. This syntax supports a filter object, `filterObj`, created by `trackingKF`, `trackingEKF`, `trackingMSCEKF`,`trackingUKF`, `trackingABF`, `trackingCKF`, `trackingGSF`, `trackingPF`,or `trackingIMM`.

`[x_corr,P_corr] = correctjpda(filterObj,z,jp,varargin)` specifies additional parameters used by the measurement function defined in the `MeasurementFcn` property of the tracking filter object. This syntax supports a filter object, `filterObj`, created by `trackingEKF`, `trackingMSCEKF`,`trackingUKF`, `trackingCKF`, `trackingGSF`, `trackingPF`, or `trackingIMM`.

`[x_corr,P_corr] = correctjpda(filterObj,z,jp,zcov)` specifies additional measurement covariance `zcov` used in the `MeasuremetNoise` property of a `trackingKF` filter object. This syntax only supports a filter object, `filterObj`, created by `trackingKF`.

`[x_corr,P_corr,z_corr] = correctjpda(filterObj,z,jp)` also returns the correction of measurements, `z_corr`. This syntax only supports a filter object, `filterObj`, created by `trackingABF`.

Input Arguments

`filterObj` — Tracking filter

object

Tracking filter, specified as an object. For example, you can create a `trackingEKF` object as

```
EKF = trackingEKF
```

and use `EKF` as the value of `filterObj`.

`z` — 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`

`jp` — 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 `jp` is the joint probability that the *i*th measurement in `z` is associated with the filter. The last element of `jp` corresponds to the probability that no measurement is associated with the filter. The sum of all elements of `jp` equals 1.

Data Types: `single` | `double`

`zcov` — Measurement covariance

M-by-*M* matrix

Measurement covariance matrix, 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 `z`.

Data Types: `single` | `double`

`varargin` — Measurement function arguments

comma-separated list of argument names

Measurement function arguments, specified as a comma-separated list. These arguments are the same ones that are passed into the measurement function specified by the

`MeasurementFcn` property of the tracking filter. For example, if you set `MeasurementFcn` to `@cameas`, and then call

```
[x_corr,P_corr] = correctjpda(filter,frame,sensorpos,sensorvel)
```

The `correctjpda` method will internally call

```
meas = cameas(state,frame,sensorpos,sensorvel)
```

Output Arguments

`x_corr` — 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 association probabilities.

`P_corr` — Corrected state error covariance matrix

positive-definite *P*-by-*P* matrix

Corrected state error covariance matrix, 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 association probabilities.

`z_corr` — 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.

Definitions

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 ($i = 1, \ldots, N$) with varied probabilities of association β_i ($i = 0, 1, \ldots, N$). Note that β_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

$$x_k{+} = x_k{-} + K_k \sum_{i=1}^{N} \beta_i (y_i - h(x_k{-}))$$
$$P_k{+} = P_k{-} - (1 - \beta_0) K_k S_k K_k{}^T + P_k$$

where

$$P_k = K_k \sum_{i=1}^{N} \left[\beta_i (y_i - h(x_k{-}))(y_i - h(x_k{-}))^T - (\delta y)(\delta y)^T \right] K_k{}^T$$

and

$$\delta y = \sum_{j=1}^{N} \beta_j \left(y_j - h(x_k^-) \right)$$

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

`trackerJPDA` | `trackingABF` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

Introduced in R2019a

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.'])
```

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

```
574 feasible joint event matrices were generated.
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 1-16.

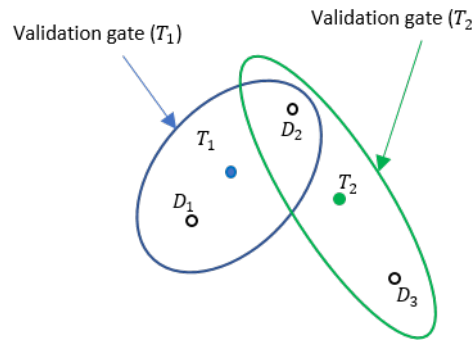
Data Types: `logical`

Definitions

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 Ω 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 Ω 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 Ω_p from the initial validation matrix Ω. For example, with the validation matrix Ω, eight FJE matrices can be obtained:

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

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

As a direct consequence of the two assumptions, the Ω_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.

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

insfilter

Create inertial navigation filter

Syntax

```
filt = insfilter
filt = insfilter(type)
filt = insfilter(Name,Value)
```

Description

`filt = insfilter` returns an inertial navigation filter object that estimates pose based on accelerometer, gyroscope, GPS, and magnetometer measurements.

`filt = insfilter(type)` returns an inertial navigation filter based on `type`.

`filt = insfilter(Name,Value)` returns an inertial navigation filter configured by the name-value pair arguments.

Examples

Create INS Filter Using Name-Value Pairs

You can specify the type of INS filter to create by specifying name-value pairs as `true` or `false`. For example, to create the default `NHConstrainedIMUFuser` object, specify `NonholonomicHeading` as `true`, `Magnetometer` as `false`, `AsyncIMU` as `false`, and `ErrorState` as `false`.

```
nhFilter = insfilter('NonholonomicHeading',true, ...
                     'Magnetometer',false, ...
                     'AsyncIMU',false, ...
                     'ErrorState',false)


nhFilter =
```

```
  NHConstrainedIMUGPSFuser with properties:

        IMUSampleRate: 100        Hz
    ReferenceLocation: [0 0 0]    [deg deg m]
     DecimationFactor: 2

   Extended Kalman Filter Values
              State: [16x1 double]
     StateCovariance: [16x16 double]

   Process Noise Variances
                  GyroscopeNoise: [4.8e-06 4.8e-06 4.8e-06]    (rad/s)²
              AccelerometerNoise: [0.048 0.048 0.048]          (m/s²)²
              GyroscopeBiasNoise: [4e-14 4e-14 4e-14]          (rad/s)²
         GyroscopeBiasDecayFactor: 0.999
           AccelerometerBiasNoise: [4e-14 4e-14 4e-14]          (m/s²)²
    AccelerometerBiasDecayFactor: 0.9999

   Measurement Noise Variances
    ZeroVelocityConstraintNoise: 0.01    (m/s)²
```

You can also create a filter by specifying the minimum number of nondefault parameters required. By default, `AsyncIMU` and `ErrorState` are `false`, so you do not need to specify them to create an `NHConstrainedIMUGPSFuser`.

```
nhFilter = insfilter('NonholonomicHeading',true, ...
                     'Magnetometer',false)
```

```
nhFilter =

  NHConstrainedIMUGPSFuser with properties:

        IMUSampleRate: 100        Hz
    ReferenceLocation: [0 0 0]    [deg deg m]
     DecimationFactor: 2

   Extended Kalman Filter Values
              State: [16x1 double]
     StateCovariance: [16x16 double]

   Process Noise Variances
                  GyroscopeNoise: [4.8e-06 4.8e-06 4.8e-06]    (rad/s)²
```

```
            AccelerometerNoise: [0.048 0.048 0.048]          (m/s²)²
            GyroscopeBiasNoise: [4e-14 4e-14 4e-14]          (rad/s)²
        GyroscopeBiasDecayFactor: 0.999
          AccelerometerBiasNoise: [4e-14 4e-14 4e-14]          (m/s²)²
     AccelerometerBiasDecayFactor: 0.9999

   Measurement Noise Variances
     ZeroVelocityConstraintNoise: 0.01    (m/s)²
```

Create INS Filter by Specifying Filter Type

You can create a specific INS filter object by specifying a string for the `type` argument. Create an `AsyncMARGGPSFuser` by calling the `insfilter` function with "`asyncimu`".

```
filter = insfilter("asyncimu")
```

```
filter =
  AsyncMARGGPSFuser with properties:

    ReferenceLocation: [0 0 0]           [deg deg m]
                State: [28x1 double]
      StateCovariance: [28x28 double]

   Additive Process Noise Variances
           QuaternionNoise: [1x4 double]
      AngularVelocityNoise: [0.005 0.005 0.005]       (rad/s)²
             PositionNoise: [1e-06 1e-06 1e-06]       m²
             VelocityNoise: [1e-06 1e-06 1e-06]       (m/s)²
        AccelerationNoise: [50 50 50]                 (m/s²)²
       GyroscopeBiasNoise: [1e-10 1e-10 1e-10]       (rad/s)²
     AccelerometerBiasNoise: [0.0001 0.0001 0.0001]    (m/s²)²
     GeomagneticVectorNoise: [1e-06 1e-06 1e-06]       uT²
      MagnetometerBiasNoise: [0.1 0.1 0.1]             uT²
```

To specify an `ErrorStateIMUGPSFuser`, call `insfilter` with "`errorstate`".

```
filter = insfilter("errorstate")
```

```
filter =
  ErrorStateIMUGPSFuser with properties:
```

```
        IMUSampleRate: 100              Hz
     ReferenceLocation: [0 0 0]         [deg deg m]
                 State: [17x1 double]
       StateCovariance: [16x16 double]

   Process Noise Variances
            GyroscopeNoise: [1e-06 1e-06 1e-06]       (rad/s)²
        AccelerometerNoise: [0.0001 0.0001 0.0001]    (m/s²)²
        GyroscopeBiasNoise: [1e-09 1e-09 1e-09]       (rad/s)²
     AccelerometerBiasNoise: [0.0001 0.0001 0.0001]    (m/s²)²
```

Create Default INS Filter

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

```
filter = insfilter
```

```
filter =

  MARGGPSFuser 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
```

Input Arguments

You can create an INS filter object by specifying the filter type, or by specifying name-value pairs.

Supported Configurations

type	Name-Value Pair				Filter Object	Filter Description
	NonholonomicHeading	Magnetometer	AsyncIMU	ErrorState		
'marg'	false	true	false	false	MARGGPSFuser	Pose estimation using an extended Kalman filter based on accelerometer, gyroscope, GPS, and magnetometer input.
'asyncimu'	false	true	true	false	AsyncMARGGPSFuser	Pose estimation using an extended continuous-discrete Kalman filter based on accelerometer, gyroscope, GPS, and magnetometer input.
'nonholonomic'	true	false	false	false	NHConstrainedIMUGPSFuser	Pose estimation using an extended Kalman filter based on accelerometer, gyroscope, and GPS input with constraints on the velocity in the lateral and vertical body axes.
'errorstate'	false	false	false	true	ErrorStateIMUGPSFuser	Pose estimation using an error-state Kalman filter based on accelerometer, gyroscope, GPS, and monocular visual odometry input.

type — Type of INS filter to create
`"marg"` | `"asyncimu"` | `"nonholonomic"` | `"errorstate"`

Type of INS filter to create, specified as `"marg"`, `"asyncimu"`, `"nonholonomic"`, or `"errorstate"`.

Data Types: `char` | `string`

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: `"NonholonomicHeading",true`

`NonholonomicHeading` — Create filter with nonholonomic heading constraints
`false` (default) | `true`

Create a filter with nonholonomic heading constraints, specified as the comma-separated pair `'NonholonomicHeading'` and `true` or `false`.

Data Types: `logical`

`Magnetometer` — Create filter with magnetometer input
`true` (default) | `false`

Create a filter that uses magnetometer readings, specified as the comma-separated pair `'Magnetometer'` and `true` or `false`.

Data Types: `logical`

`AsyncIMU` — Create continuous-discrete filter
`false` (default) | `true`

Create a continuous-discrete filter, specified as the comma-separated pair `'Magnetometer'` and `true` or `false`.

Data Types: `logical`

`ErrorState` — Create error-state filter that uses visual odometry
`false` (default) | `true`

Create an error-state Kalman filter that uses visual odometry data, specified as the comma-separated pair `'ErrorState'` and `true` or `false`.

Data Types: `logical`

Output Arguments

`filt` — Inertial navigation filter
`MARGGPSFuser` (default) | `NHConstrainedIMUGPSFuser` | `AsyncMARGGPSFuser` | `ErrorStateIMUGPSFuser`

Inertial navigation filter, returned as a `MARGGPSFuser`, `NHConstrainedIMUGPSFuser`, `AsyncMARGGPSFuser`, or `ErrorStateIMUGPSFuser` object.

Extended Capabilities

C/C++ Code Generation

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

See Also

`AsyncMARGGPSFuser` | `ErrorStateIMUGPSFuser` | `MARGGPSFuser` | `NHConstrainedIMUGPSFuser` | `ahrsfilter` | `imufilter`

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)
```

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.

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 = ecompass(a,m);
quaterionEulerAngles = eulerd(q,'ZYX','frame')

r = ecompass(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])
```

```
quaterionEulerAngles =

   14.6563         0         0


rotmatEulerAngles =

   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 = ecompass(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 = 1×3

    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,a
```

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.

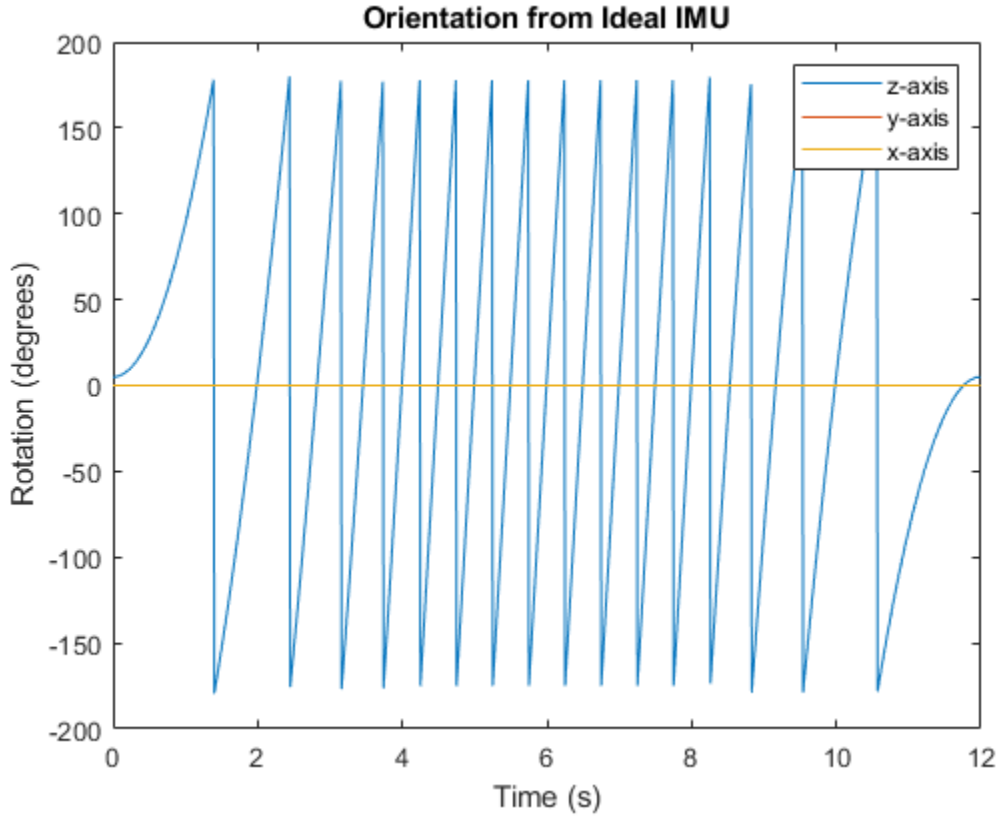
```
IMU = imuSensor('accel-mag','SampleRate',fs);
[accelerometerData,magnetometerData] = IMU(accelerationNED, ...
                                           angularVelocityNED, ...
                                           orientationNED);
```

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')
```

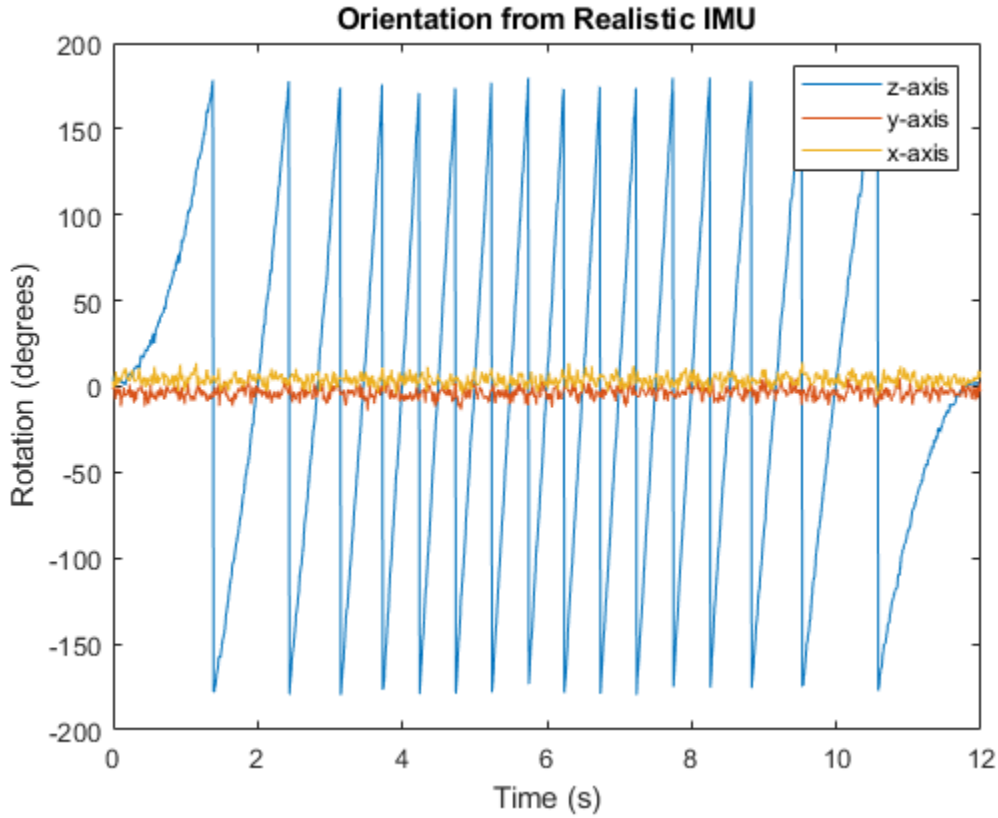


Repeat Experiment with Realistic IMU Sensor Model

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,orientati

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^2)

N-by-3 matrix

Accelerometer readings in sensor body coordinate system in m/s^2, 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

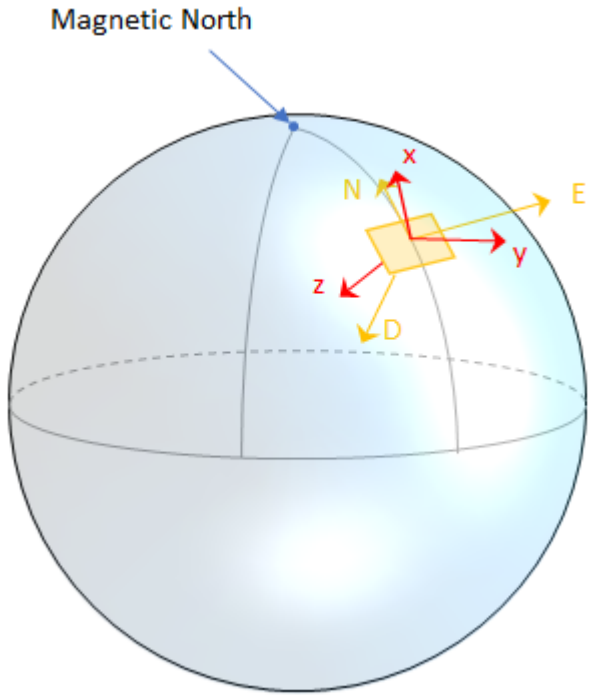
The `ecompass` function returns a quaternion or rotation matrix that can rotate quantities from a parent (NED) frame to a child (sensor) frame. For both output orientation formats, the rotation operator is determined by computing the rotation matrix.

The rotation matrix is first calculated with an intermediary:

$$R = \left[(a \times m) \times a \;\; a \times m \;\; a\right]$$

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.



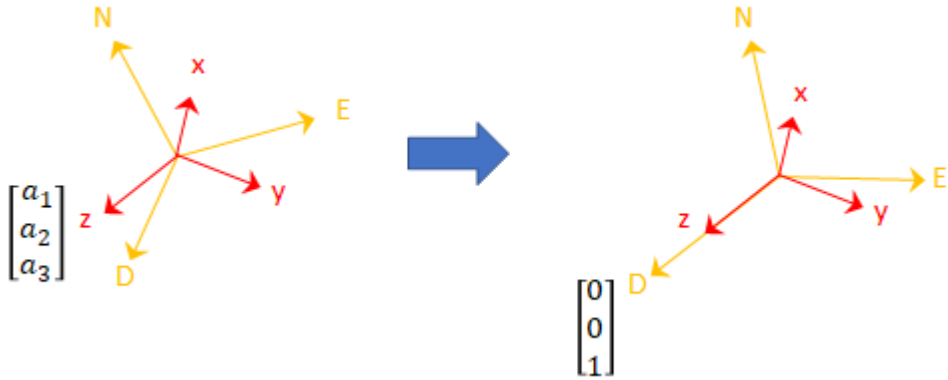
Recall that orientation of a sensor body is defined as the rotation operator (rotation matrix or quaternion) required to rotate a quantity from a parent (NED) frame to a child (sensor body) frame:

$$\left[\; R \;\right]\left[p_{\text{parent}}\right] = \left[p_{\text{child}}\right]$$

where

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

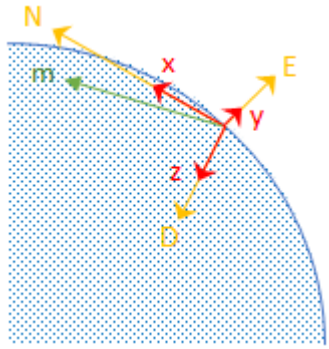


$$\begin{bmatrix} r_{11} & r_{21} & r_{31} \\ r_{12} & r_{22} & r_{32} \\ r_{13} & r_{23} & r_{33} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

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

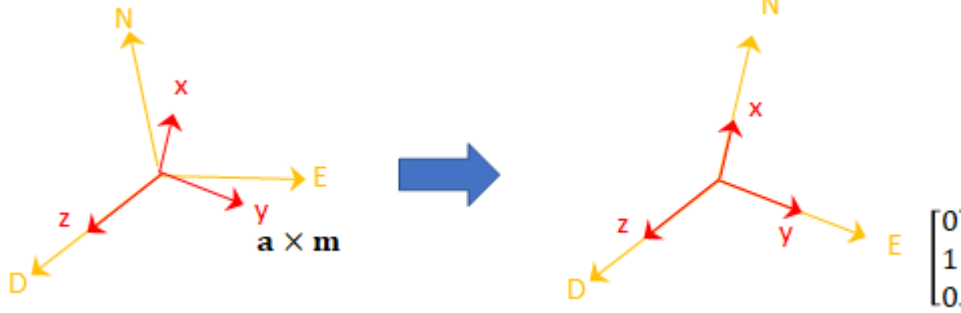
$$\begin{bmatrix} r_{31} \\ r_{32} \\ r_{33} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

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:

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

Finally, the rotation matrix is normalized column-wise:

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

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'.

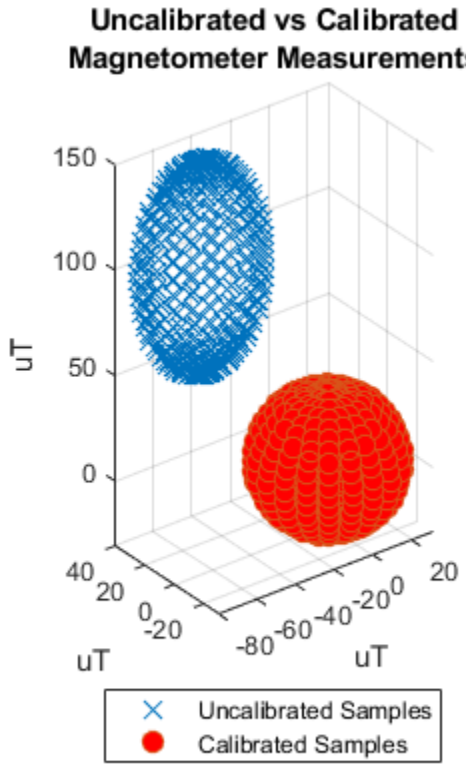
```
[A,b,expmfs] = magcal(D); % calibration coefficients
expmfs % Dipaly expected  magnetic field strength in uT
```

```
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

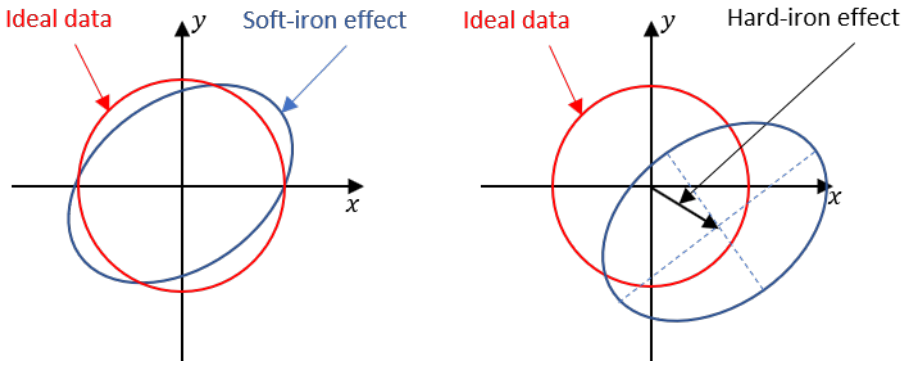
Expected magnetic filed strength, returned as a scalar.

Definitions

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 left.



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`

System Objects
`imuSensor`

Introduced in R2019a

partitionDetections

Partition detections based on Mahalanobis distance

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.

Syntax

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

Description

`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

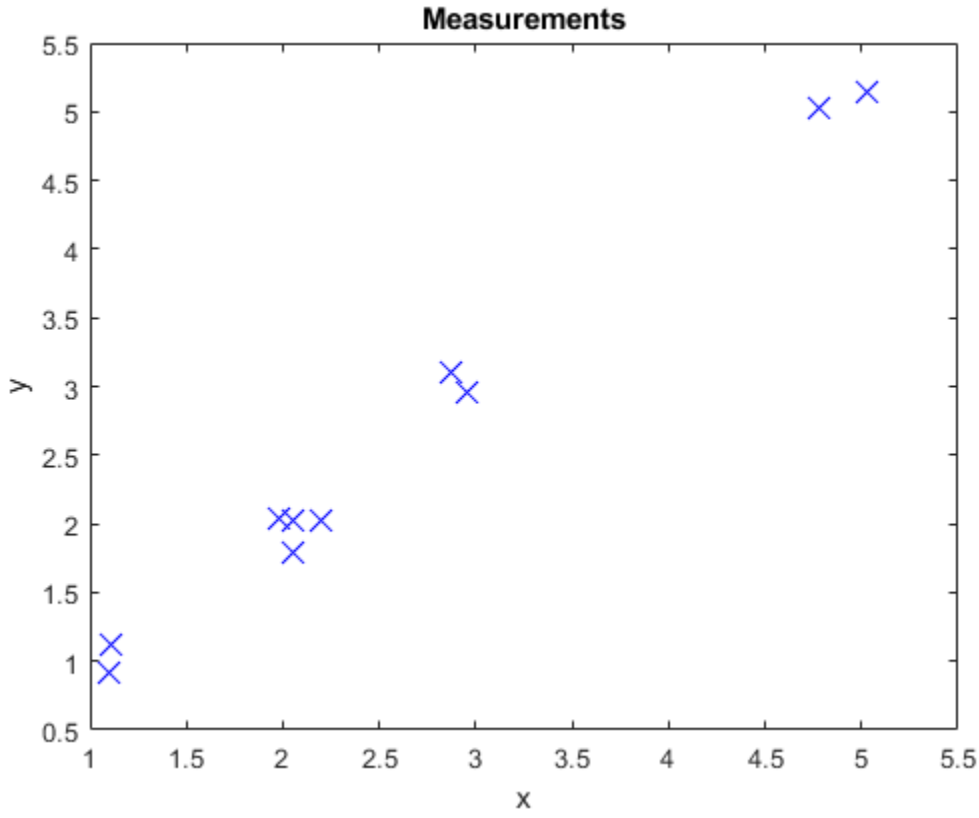
Generate Partition from Object Detection

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')
```



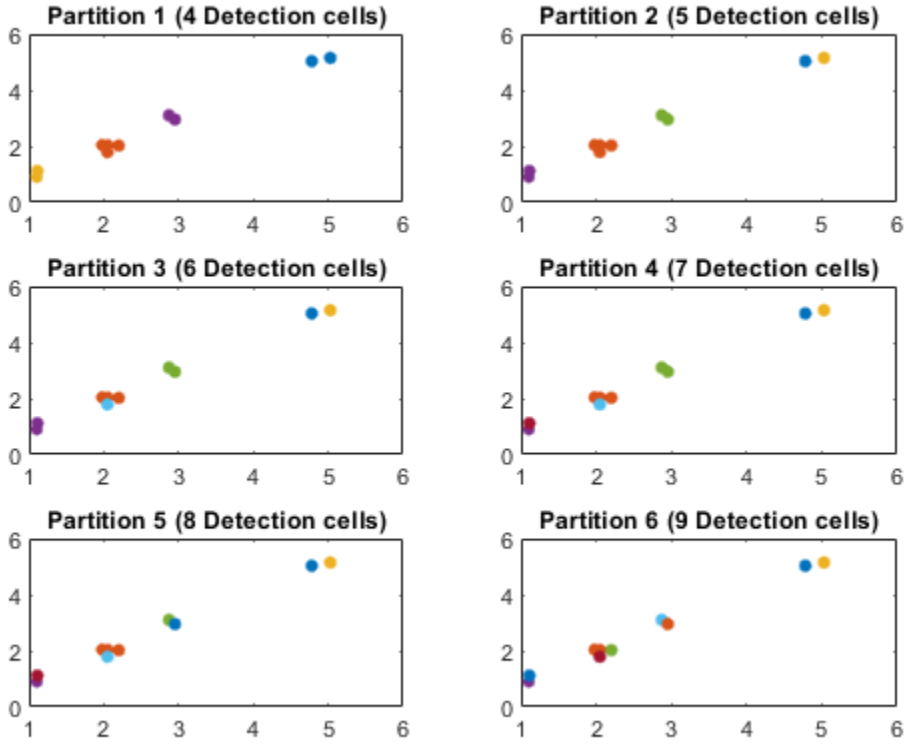
Generate partitions from the detections and count the number of partitions.

```
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] Granstorm, K., C. Lundquiest, 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 `m1`-by-...-by-`mN` array of random unit quaternions, where `m1`,..., `mN` indicate the size of each dimension. For example, `randrot(3,4)` returns a 3-by-4 matrix of random unit quaternions.

`R = randrot([m1,...,mN])` returns an `m1`-by-...-by-`mN` array of random unit quaternions, where `m1`,..., `mN` indicate the size of each dimension. For example, `randrot([3,4])` returns a 3-by-4 matrix of random unit quaternions.

Examples

Matrix of Random Rotations

Generate a 3-by-3 matrix of uniformly distributed random rotations.

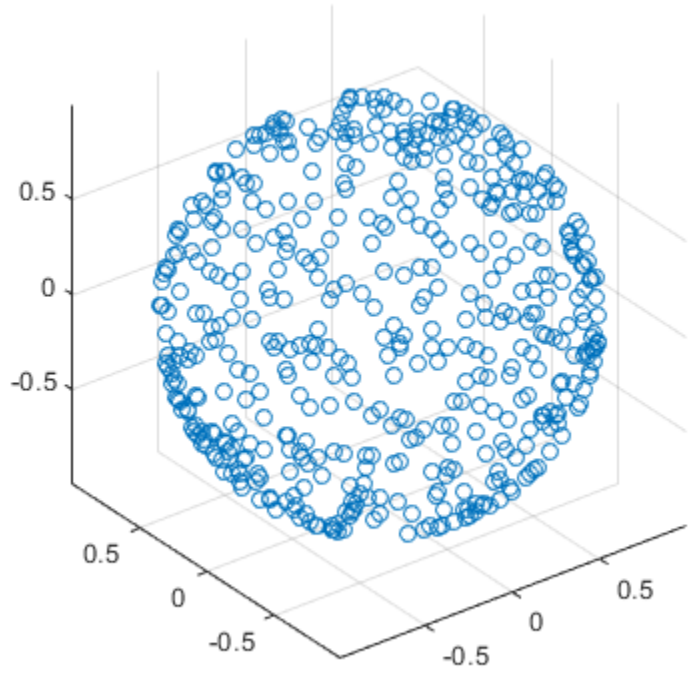
```
r = randrot(3)
```

```
r =

  3×3 quaternion array

      0.71601 - 0.048195i +  0.69548j + 0.036254k     -0.33542 -  0.39466i -  0.84503j
      0.31625 +  0.20986i +  0.29758j -  0.87601k      0.42409 - 0.047461i +  0.28419j
      0.16941 +  0.32961i -  0.74097j +  0.56002k      0.42141 +  0.88708i +  0.09635j
```

Create Uniform Distribution of Random Rotations

Create a vector of 500 random quaternions. Use `rotatepoint` to visualize the distribution of the random rotations applied to point (1, 0, 0).

```
q = randrot(500,1);

pt = rotatepoint(q, [1 0 0]);

figure
scatter3(pt(:,1), pt(:,2), pt(:,3))
axis equal
```



Input Arguments

`m` — Size of square matrix

integer

Size of square quaternion matrix, specified as an integer value. If `m` is `0` or negative, then `R` is returned as an empty matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`m1,...,mN` — Size of each dimension
two or more integer values

Size of each dimension, specified as two or more integer values. If the size of any dimension is `0` or negative, then `R` is returned as an empty array.

Example: `randrot(2,3)` returns a 2-by-3 matrix of random quaternions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`[m1,...,mN]` — Vector of size of each dimension
row vector of integer values

Vector of size of each dimension, specified as a row vector of two or more integer values. If the size of any dimension is `0` or negative, then `R` is returned as an empty array.

Example: `randrot([2,3])` returns a 2-by-3 matrix of random quaternions.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`R` — Random quaternions
scalar | vector | matrix | multidimensional array

Random quaternions, returned as a quaternion or array of quaternions.

Data Types: `quaternion`

References

[1] Shoemake, K. "Uniform Random Rotations." *Graphics Gems III* (K. David, ed.). New York: Academic Press, 1992.

Extended Capabilities

C/C++ Code Generation

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

See Also

`quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2019a

rotvecd

Convert quaternion to rotation vector (degrees)

Syntax

```
rotationVector = rotvecd(quat)
```

Description

`rotationVector = rotvecd(quat)` converts the quaternion array, `quat`, to an *N*-by-3 matrix of equivalent rotation vectors in degrees. The elements of `quat` are normalized before conversion.

Examples

Convert Quaternion to Rotation Vector in Degrees

Convert a random quaternion scalar to a rotation vector in degrees.

```
quat = quaternion(randn(1,4));
rotvecd(quat)
```

```
ans = 1×3

   96.6345 -119.0274   45.4312
```

Input Arguments

`quat` — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

`rotationVector` — Rotation vector (degrees)
N-by-3 matrix

Rotation vector representation, returned as an *N*-by-3 matrix of rotation vectors, where each row represents the [*x* *y* *z*] angles of the rotation vectors in degrees. The *i*th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

All rotations in 3-D can be represented by four elements: a three-element axis of rotation and a rotation angle. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

$$q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(\mathrm{xi} + y\mathrm{j} + z\mathrm{k}),$$

where θ 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 θ over the parts b, c, and d. The rotation vector representation of q is

$$q_{\mathrm{rv}} = \frac{\theta}{\sin\left(\theta/2\right)}[b, c, d]\,.$$

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 R2018a

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'` | `'ZYZ'` | `'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.

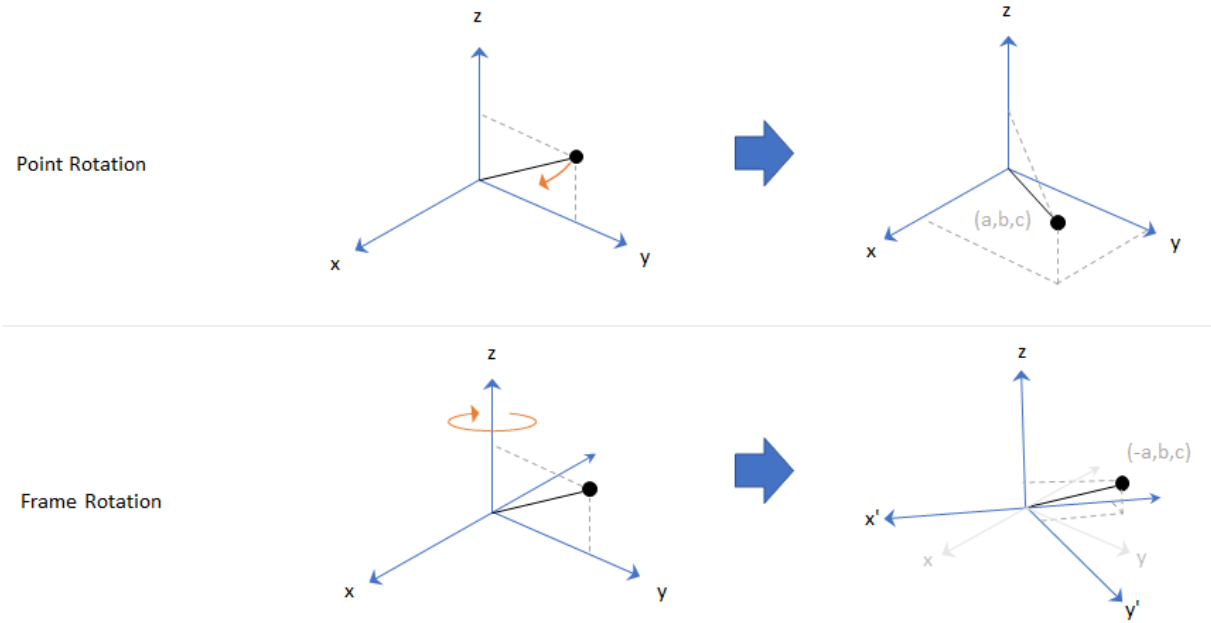
Data Types: `char` | `string`

`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 R2018a

meanrot

Quaternion mean rotation

Syntax

```
quatAverage = meanrot(quat)
quatAverage = meanrot(quat,dim)
quatAverage = meanrot( ___ ,nanflag)
```

Description

`quatAverage = meanrot(quat)` returns the average rotation of the elements of `quat` along the first array dimension whose size not does equal 1.

- If `quat` is a vector, `meanrot(quat)` returns the average rotation of the elements.
- If `quat` is a matrix, `meanrot(quat)` returns a row vector containing the average rotation of each column.
- If `quat` is a multidimensional array, then `mearot(quat)` operates along the first array dimension whose size does not equal 1, treating the elements as vectors. This dimension becomes 1 while the sizes of all other dimensions remain the same.

The `meanrot` function normalizes the input quaternions, `quat`, before calculating the mean.

`quatAverage = meanrot(quat,dim)` return the average rotation along dimension `dim`. For example, if `quat` is a matrix, then `meanrot(quat,2)` is a column vector containing the mean of each row.

`quatAverage = meanrot( ___ ,nanflag)` specifies whether to include or omit `NaN` values from the calculation for any of the previous syntaxes. `meanrot(quat,'includenan')` includes all `NaN` values in the calculation while `mean(quat,'omitnan')` ignores them.

Examples

Quaternion Mean Rotation

Create a matrix of quaternions corresponding to three sets of Euler angles.

```
eulerAngles = [40 20 10; ...
               50 10 5; ...
               45 70 1];

quat = quaternion(eulerAngles,'eulerd','ZYX','frame');
```

Determine the average rotation represented by the quaternions. Convert the average rotation to Euler angles in degrees for readability.

```
quatAverage = meanrot(quat)
eulerAverage = eulerd(quatAverage,'ZYX','frame')
```

```
quatAverage =

  quaternion

      0.88863 - 0.062598i +  0.27822j +  0.35918k


eulerAverage =

   45.7876   32.6452    6.0407
```

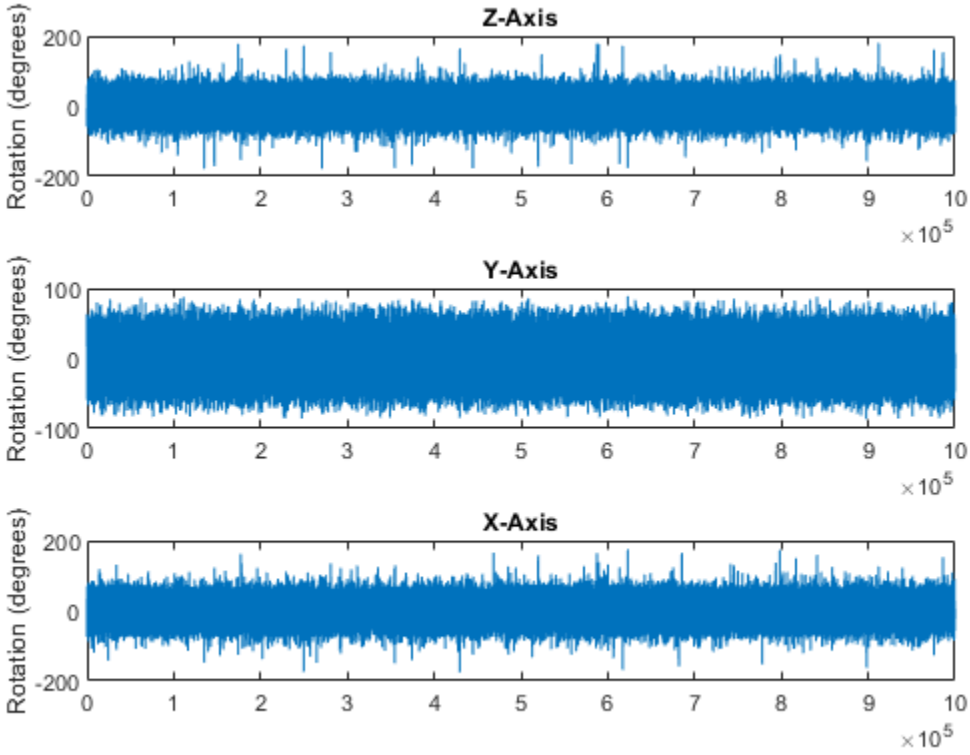
Average Out Rotational Noise

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;
```

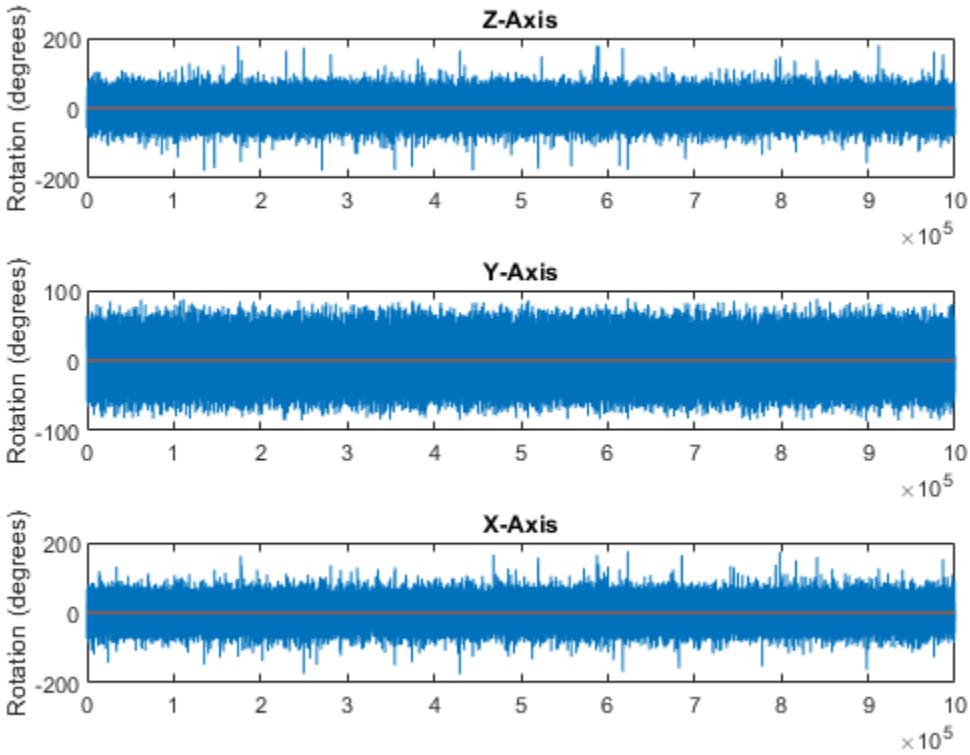
```
ax = ax./sqrt(sum(ax.^2,2));
ang = 0.5*randn(size(ax,1),1);
q = quaternion(ax.*ang ,'rotvec');

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');
```

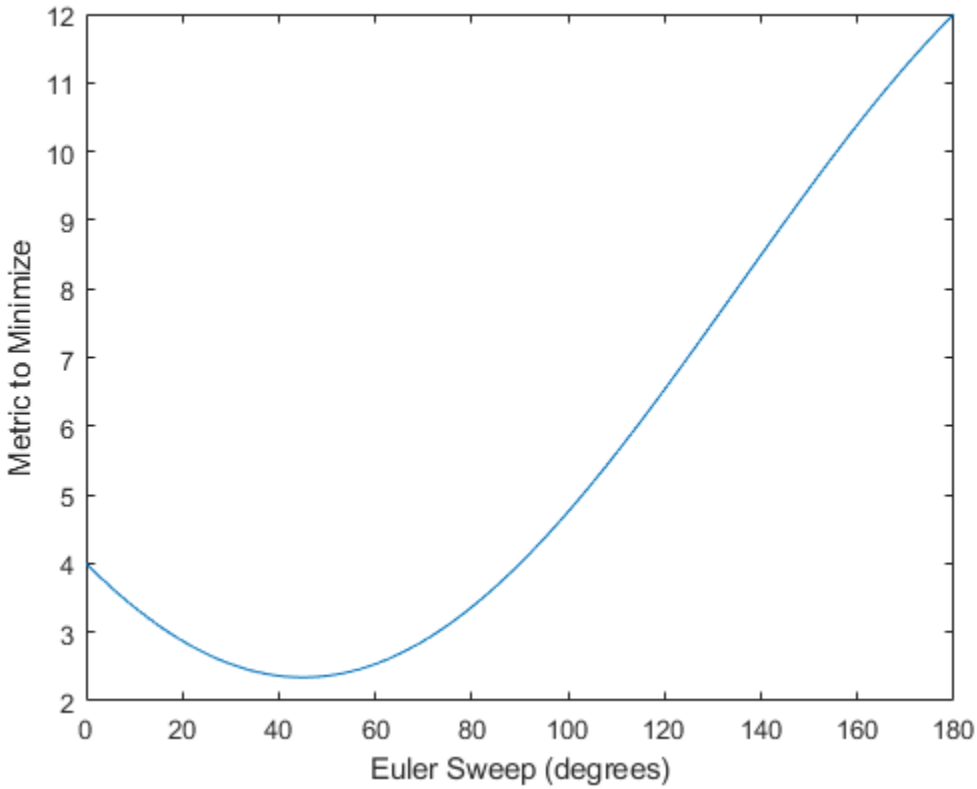
Create a quaternion sweep, `qSweep`, that represents rotations from 0 to 180 degrees about the *x*-axis.

```
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 `quaterion([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 =
```

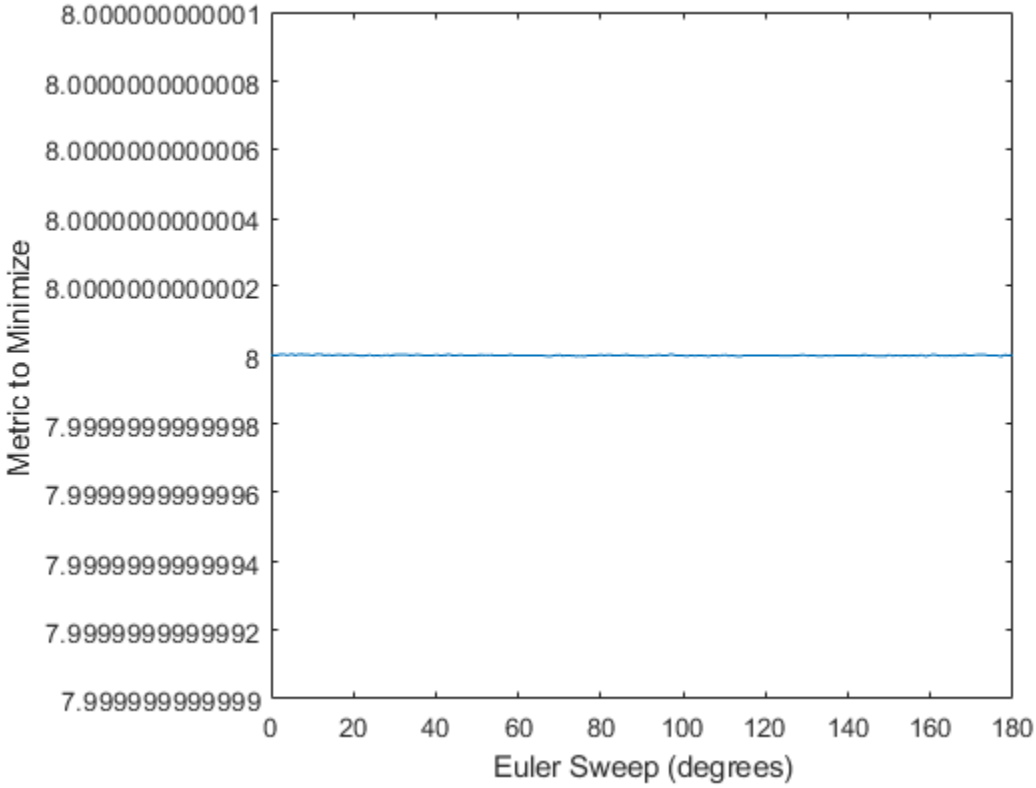
```
         0         0   45.0000
```

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 =
```

```
         0         0   90.0000
```

Input Arguments

`quat` — Quaternion

scalar | vector | matrix | multidimensional array

Quaternion for which to calculate the mean, specified as a scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

`dim` — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Dimension `dim` indicates the dimension whose length reduces to 1. The `size(quatAverage,dim)` is `1`, while the sizes of all other dimensions remain the same.

Data Types: `double` | `single`

`nanflag` — `NaN` condition

`'includenan'` (default) | `'omitnan'`

`NaN` condition, specified as one of these values:

- `'includenan'` –– Include `NaN` values when computing the mean rotation, resulting in `NaN`.
- `'omitnan'` –– Ignore all `NaN` values in the input.

Data Types: `char` | `string`

Output Arguments

`quatAverage` — Quaternion average rotation

scalar | vector | matrix | multidimensional array

Quaternion average rotation, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double`

Algorithms

`meanrot` determines a quaternion mean, $\bar{q}$, according to [1]. $\bar{q}$ is the quaternion that minimizes the squared Frobenius norm of the difference between rotation matrices:

$$\bar{q} = \arg \min_{q \in \mathrm{S}^3} \sum_{i=1}^{n} \|A(q) - A(q_i)\|_F^2$$

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

```
q0 = slerp(q1,q2,T)
```

Description

`q0 = 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 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 =

     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 =

   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 =

    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
```

```
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
    9.0000         0         0
```

Alternatively, you can use the `dist` function to verify that the distance between the interpolated quaternions is consistent. The `dist` function returns angular distance in radians; convert to degrees for easy comparison.

```
def = rad2deg(dist(interpolatedQuaternions(2:end),interpolatedQuaternions(1:end-1)))
```

```
def =

  Columns 1 through 7

    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000    9.0000

  Columns 8 through 10

    9.0000    9.0000    9.0000
```

SLERP Minimizes Great Circle Path

The SLERP algorithm interpolates along a great circle path connecting two quaternions. This example shows how the SLERP algorithm minimizes the great circle path.

Define three quaternions:

1. `q0` - quaternion indicating no rotation from the global frame
2. `q179` - quaternion indicating a 179 degree rotation about the *z*-axis
3. `q180` - quaternion indicating a 180 degree rotation about the *z*-axis
4. `q181` - quaternion indicating a 181 degree rotation about the *z*-axis

```
q0 = ones(1,'quaternion');
```

```
q179 = quaternion([179,0,0],'eulerd','ZYX','frame');
```

```
q180 = quaternion([180,0,0],'eulerd','ZYX','frame');
```

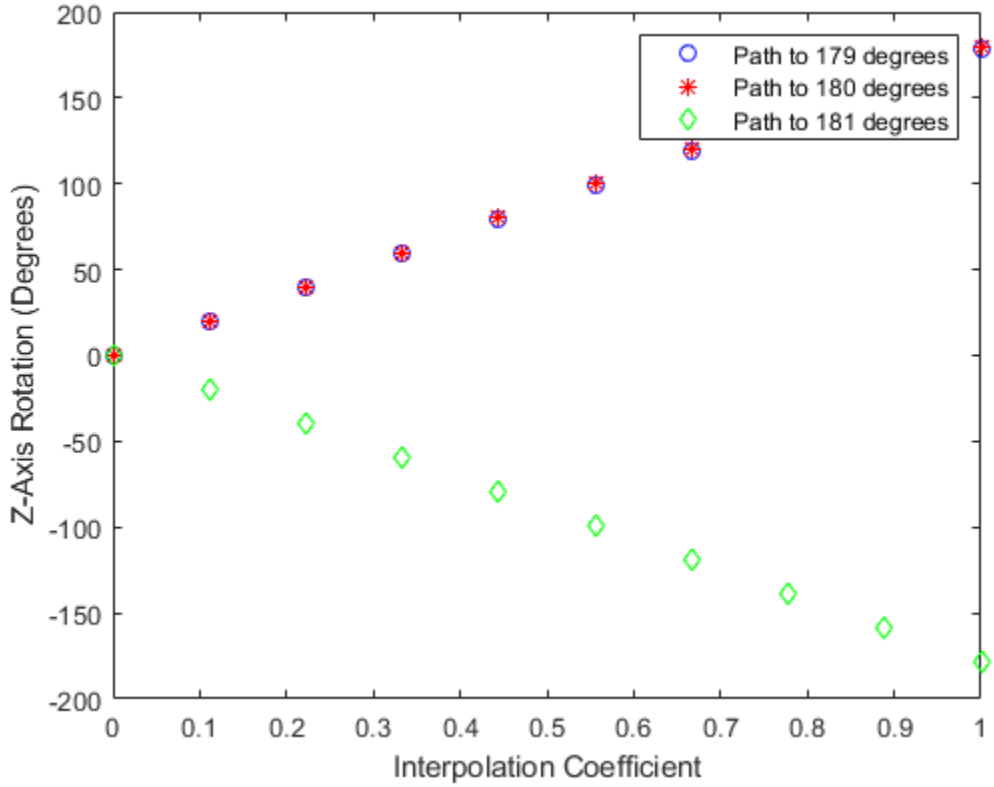
```
q181 = quaternion([181,0,0],'eulerd','ZYX','frame');
```

Use `slerp` to interpolate between `q0` and the three quaternion rotations. Specify that the paths are traveled in 10 steps.

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

q179path = slerp(q0,q179,T);
q180path = slerp(q0,q180,T);
q181path = slerp(q0,q181,T);
```

Plot each path in terms of Euler angles in degrees.

```
q179pathEuler = eulerd(q179path,'ZYX','frame');
q180pathEuler = eulerd(q180path,'ZYX','frame');
q181pathEuler = eulerd(q181path,'ZYX','frame');

plot(T,q179pathEuler(:,1),'bo', ...
     T,q180pathEuler(:,1),'r*', ...
     T,q181pathEuler(:,1),'gd');
legend('Path to 179 degrees', ...
       'Path to 180 degrees', ...
       'Path to 181 degrees')
xlabel('Interpolation Coefficient')
ylabel('Z-Axis Rotation (Degrees)')
```



The path between `q0` and `q179` is clockwise to minimize the great circle distance. The path between `q0` and `q181` is counterclockwise to minimize the great circle distance. The path between `q0` and `q180` 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.

`q1`, `q2`, 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.

`q1`, `q2`, 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].

`q1`, `q2`, and `T` must have compatible sizes. In the simplest cases, they can be the same size or any one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are either the same or one of the dimension sizes is 1.

Data Types: `single` | `double`

Output Arguments

`q0` — Interpolated quaternion

scalar | vector | matrix | multidimensional array

Interpolated quaternion, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Quaternion **s**pherical **l**inear int**erp**olation (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 θ 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 R2018a

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^{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 = 2x2 quaternion array
      1 +  2i +  3j +  4k      9 + 10i + 11j + 12k
      5 +  6i +  7j +  8k     13 + 14i + 15j + 16k


quatArrayParts = compact(quatArray)

quatArrayParts = 4×4

     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

Input Arguments

quat — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

matrix — Quaternion in matrix form

N-by-4 matrix

Quaternion in matrix form, returned as an N-by-4 matrix, where N = `numel(quat)`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`classUnderlying` | `parts`

Objects

`quaternion`

Topics

"Rotations, Orientation, and Quaternions"

Introduced in R2018a

conj

Complex conjugate of quaternion

Syntax

```
quatConjugate = conj(quat)
```

Description

`quatConjugate = conj(quat)` returns the complex conjugate of the quaternion, `quat`.

If $q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, the complex conjugate of q is $q^* = a - b\mathrm{i} - c\mathrm{j} - d\mathrm{k}$. 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 R2018a

ctranspose, '

Complex conjugate transpose of quaternion array

Syntax

```
quatTransposed = quat'
```

Description

`quatTransposed = quat'` returns the complex conjugate transpose of the quaternion, `quat`.

Examples

Vector Complex Conjugate Transpose

Create a vector of quaternions and compute its complex conjugate transpose.

```
quat = quaternion(randn(4,4))
```

```
quat = 4x1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
quatTransposed = quat'
```

```
quatTransposed = 1x4 quaternion array
      0.53767 -  0.31877i -   3.5784j -   0.7254k       1.8339 +   1.3077i -   2.7694j
```

Matrix Complex Conjugate Transpose

Create a matrix of quaternions and compute its complex conjugate transpose.

```
quat = [quaternion(randn(2,4)),quaternion(randn(2,4))]
```

```
quat = 2x2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       3.5784 -   1.3499i +   0.7254j
       1.8339 +  0.86217i -   1.3077j +  0.34262k       2.7694 +   3.0349i - 0.063055j


```

```
quatTransposed = quat'
```

```
quatTransposed = 2x2 quaternion array
      0.53767 +   2.2588i -  0.31877j +  0.43359k       1.8339 -  0.86217i +   1.3077j
       3.5784 +   1.3499i -   0.7254j -  0.71474k       2.7694 -   3.0349i + 0.063055j
```

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

`quaternion` | `transpose, .'`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

dist

Angular distance in radians

Syntax

```
distance = dist(quatA,quatB)
```

Description

`distance = dist(quatA,quatB)` returns the angular distance in radians between the quaternion rotation operators for `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,'eulerd','zyx','frame');
qVector2 = quaternion(angles2,'eulerd','zyx','frame');

rad2deg(dist(qVector1,qVector2))
```

```
ans = 4×1

    6.0000
    6.0827
    6.0827
    6.0287
```

Note that a quaternion represents the same rotation as its negative. Calculate a quaternion and its negative.

```
qPositive = quaternion([30,45,-60],'eulerd','zyx','frame')
```

```
qPositive = quaternion
     0.72332 - 0.53198i + 0.20056j +  0.3919k
```

```
qNegative = -qPositive
```

```
qNegative = quaternion
    -0.72332 + 0.53198i - 0.20056j -  0.3919k
```

Find the distance between the quaternion and its negative.

```
dist(qPositive,qNegative)
```

```
ans = 0
```

The components of a quaternion may look different from the components of its negative, but both expressions represent the same rotation.

Input Arguments

`quatA,quatB` — Quaternions to calculate distance between

scalar | vector | matrix | multidimensional array

Quaternions to calculate distance between, specified as comma-separated quaternions or arrays of quaternions. `quatA` and `quatB` must have compatible sizes:

- `size(quatA) == size(quatB)`, or
- `numel(quatA) == 1`, or
- `numel(quatB) == 1`, or
- if `[Adim1,…,AdimN] = size(quatA)` and `[Bdim1,…,BdimN] = size(quatB)`, then for `i = 1:N`, either `Adimi==Bdimi` or `Adim==1` or `Bdim==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

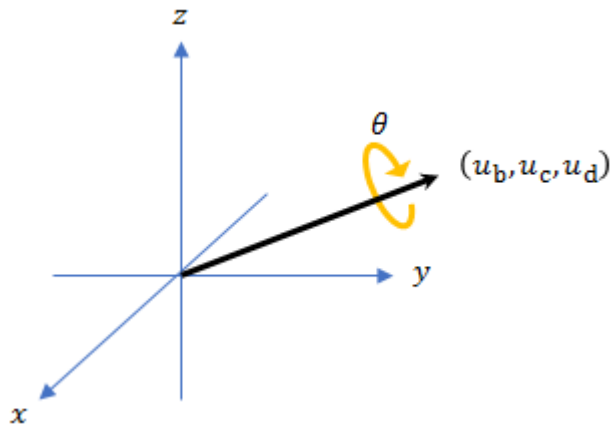
Angular distance in radians, returned as an array. The dimensions are the maximum of the union of `size(quatA)` and `size(quatB)`.

Data Types: `single` | `double`

Algorithms

The `dist` function returns the angular distance between two quaternion rotation operators.

A quaternion may be defined by an axis (u_b,u_c,u_d) and angle of rotation θ_q:
$q = \cos\left(\theta_q/2\right) + \sin\left(\theta_q/2\right)(u_b\mathrm{i} + u_c\mathrm{j} + u_d\mathrm{k})$.



Given a quaternion in the form, $q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, where a is the real part, you can solve for θ_q: $\theta_q = 2\cos^{-1}(a)$.

Consider two quaternions, p and q, and the product $z = p*\text{conjugate}(q)$. In a rotation operator, z rotates by p and derotates by q. As p approaches q, the angle of z goes to 0, and the product 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, angular distance is calculated as:

```
angularDistance = 2*acos(parts(p*conj(q)));
```

Extended Capabilities

C/C++ Code Generation

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

See Also

`conj` | `parts` | `quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

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'` | `'ZYZ'` | `'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.

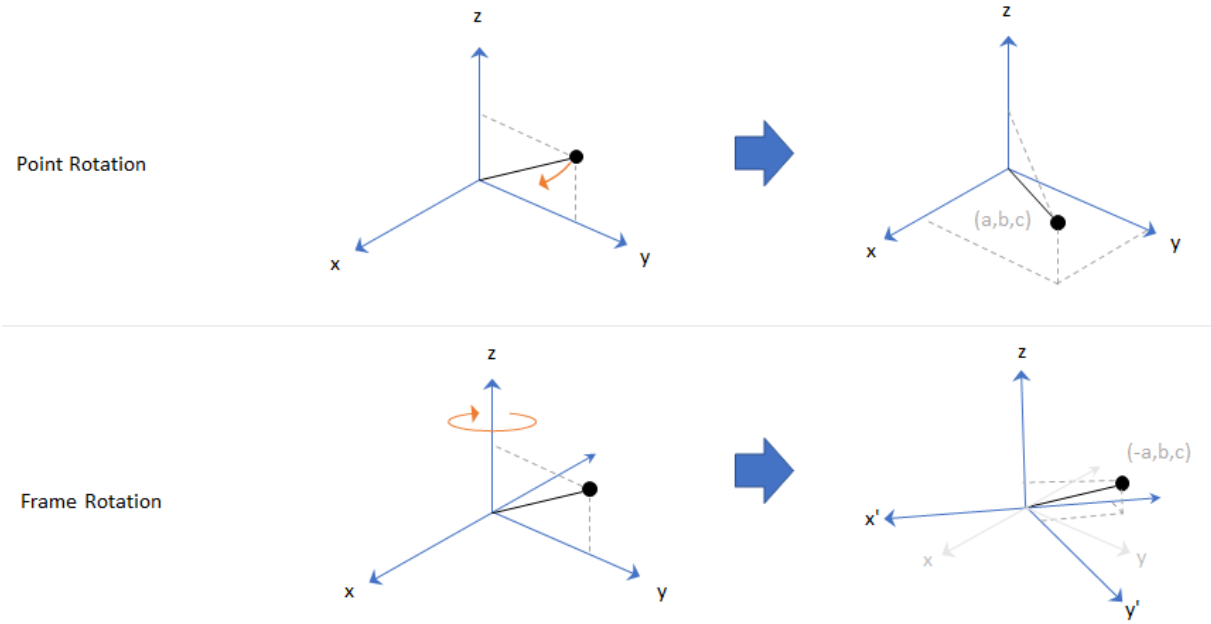
Data Types: `char` | `string`

`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

`eulerd` | `quaternion` | `rotateframe` | `rotatepoint`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

exp

Exponential of quaternion array

Syntax

```
B = exp(A)
```

Description

`B = exp(A)` computes the exponential of the elements of the quaternion array `A`.

Examples

Exponential of Quaternion Array

Create a 4-by-1 quaternion array A.

```
A = quaternion(magic(4))
```

```
A = 4x1 quaternion array
     16 +  2i +  3j + 13k
      5 + 11i + 10j +  8k
      9 +  7i +  6j + 12k
      4 + 14i + 15j +  1k
```

Compute the exponential of A.

```
B = exp(A)
```

```
B = 4x1 quaternion array
     5.3525e+06 + 1.0516e+06i + 1.5774e+06j + 6.8352e+06k
        -57.359 -     89.189i -     81.081j -     64.865k
        -6799.1 +     2039.1i +     1747.8j +     3495.6k
          -6.66 +     36.931i +     39.569j +     2.6379k
```

Input Arguments

A — Input quaternion

scalar | vector | matrix | multidimensional array

Input quaternion, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

B — Result

scalar | vector | matrix | multidimensional array

Result of quaternion exponential, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Given a quaternion $A = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k} = a + \bar{v}$, 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 = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

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

```
C = 2x1 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 = 2x2 quaternion array
     1 + 2i + 3j + 4k     4 + 5i + 6j + 7k
     2 + 3i + 4j + 5k     5 + 6i + 7j + 8k
```

```
q2 = quaternion(magic(4));
B = reshape(q2,2,2)
```

```
B = 2x2 quaternion array
     16 +  2i +  3j + 13k      9 +  7i +  6j + 12k
      5 + 11i + 10j +  8k      4 + 14i + 15j +  1k
```

```
C = A.\B
```

```
C = 2x2 quaternion array
          2.7 -      1.9i -      0.9j -      1.7k       1.5159 -  0.37302i -  0.15079j
       2.2778 +  0.46296i -  0.57407j + 0.092593k       1.2471 +  0.91379i -  0.33908j
```

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 $A = a_1 + a_2\mathrm{i} + a_3\mathrm{j} + a_4\mathrm{k}$ and a real scalar p,

$$C = p\,.\backslash A = \frac{a_1}{p} + \frac{a_2}{p}\mathrm{i} + \frac{a_3}{p}\mathrm{j} + \frac{a_4}{p}\mathrm{k}$$

Note For a real scalar p, $A./p = A.\backslash p$.

Quaternion Division by a Quaternion Scalar

Given two quaternions A and B of compatible sizes, then

$$C = A.\backslash B = A^{-1}.*B = \left(\frac{conj(A)}{norm(A)^2}\right).*B$$

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 = 3x1 quaternion array
     0.53767 + 0.86217i - 0.43359j +  2.7694k
      1.8339 + 0.31877i + 0.34262j -  1.3499k
     -2.2588 -  1.3077i +  3.5784j +  3.0349k
```

Compute the logarithmic values of A.

```
B = log(A)
```

```
B = 3x1 quaternion array
      1.0925 + 0.40848i - 0.20543j +  1.3121k
      0.8436 + 0.14767i + 0.15872j - 0.62533k
      1.6807 - 0.53829i +   1.473j +  1.2493k
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Output Arguments

B — Logarithm values
scalar | vector | matrix | multidimensional array

Quaternion natural logarithm values, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Given a quaternion $A = a + \bar{v} = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, 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 R2018a

mtimes, *

Quaternion multiplication

Syntax

```
quatC = A*B
```

Description

`quatC = A*B` implements quaternion multiplication if either `A` or `B` is a quaternion. Either `A` or `B` must be a scalar.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the order of the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the order pq. The rotation operator becomes $(pq)^{*}v(pq)$, where v represents the object to rotate specified in quaternion form. * represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the reverse order, qp. The rotation operator becomes $(qp)v(qp)^{*}$.

Examples

Multiply Quaternion Scalar and Quaternion Vector

Create a 4-by-1 column vector, `A`, and a scalar, `b`. Multiply `A` times `b`.

```
A = quaternion(randn(4,4))
```

```
A = 4x1 quaternion array
      0.53767 +  0.31877i +   3.5784j +   0.7254k
```

```
       1.8339 -   1.3077i +   2.7694j - 0.063055k
      -2.2588 -  0.43359i -   1.3499j +  0.71474k
      0.86217 +  0.34262i +   3.0349j -  0.20497k
```

```
b = quaternion(randn(1,4))
```

```
b = quaternion
    -0.12414 +  1.4897i +   1.409j +  1.4172k
```

```
C = A*b
```

```
C = 4x1 quaternion array
      -6.6117 +   4.8105i +  0.94224j -   4.2097k
      -2.0925 +   6.9079i +   3.9995j -   3.3614k
       1.8155 -   6.2313i -    1.336j -     1.89k
      -4.6033 +   5.8317i + 0.047161j -    2.791k
```

Input Arguments

A — Input

scalar | vector | matrix | multidimensional array

Input to multiply, specified as a quaternion, array of quaternions, real scalar, or array of real scalars.

If B is nonscalar, then A must be scalar.

Data Types: `quaternion` | `single` | `double`

B — Input

scalar | vector | matrix | multidimensional array

Input to multiply, specified as a quaternion, array of quaternions, real scalar, or array of real scalars.

If A is nonscalar, then B must be scalar.

Data Types: `quaternion` | `single` | `double`

Output Arguments

`quatC` — Quaternion product
scalar | vector | matrix | multidimensional array

Quaternion product, returned as a quaternion or array of quaternions.

Data Types: `quaternion`

Algorithms

Quaternion Multiplication by a Real Scalar

Given a quaternion

$$q = a_q + b_q i + c_q j + d_q k,$$

the product of q and a real scalar β is

$$\beta q = \beta a_q + \beta b_q i + \beta c_q j + \beta d_q k$$

Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$i^2 = j^2 = k^2 = ijk = -1,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	1	**i**	**j**	**k**
1	1	i	j	k
i	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, $q = a_q + b_q i + c_q j + d_q k$, and $p = a_p + b_p i + c_p j + d_p k$, the multiplication can be expanded as:

$$\begin{aligned} z = pq &= \left(a_p + b_p i + c_p j + d_p k\right)\left(a_q + b_q i + c_q j + d_q k\right) \\ &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\ &+ b_p a_q i + b_p b_q i^2 + b_p c_q ij + b_p d_q ik \\ &+ c_p a_q j + c_p b_q ji + c_p c_q j^2 + c_p d_q jk \\ &+ d_p a_q k + d_p b_q ki + d_p c_q kj + d_p d_q k^2 \end{aligned}$$

You can simplify the equation using the quaternion multiplication table:

$$\begin{aligned} z = pq &= a_p a_q + a_p b_q i + a_p c_q j + a_p d_q k \\ &+ b_p a_q i - b_p b_q + b_p c_q k - b_p d_q j \\ &+ c_p a_q j - c_p b_q k - c_p c_q + c_p d_q i \\ &+ d_p a_q k + d_p b_q j - d_p c_q i - d_p d_q \end{aligned}$$

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 R2018a

norm

Quaternion norm

Syntax

```
N = norm(quat)
```

Description

`N = norm(quat)` returns the norm of the quaternion, `quat`.

Given a quaternion of the form $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, the norm of the quaternion is defined as $\mathrm{norm}(Q) = \sqrt{a^2 + b^2 + c^2 + d^2}$.

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

`conj` | `normalize` | `parts` | `quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

normalize

Quaternion normalization

Syntax

```
quatNormalized = normalize(quat)
```

Description

`quatNormalized = normalize(quat)` normalizes the quaternion.

Given a quaternion of the form $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, 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
`conj` | `norm` | `quaternion` | `times, .*`

Topics
“Rotations, Orientation, and Quaternions”

Introduced in R2018a

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 $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, a quaternion one is defined as $Q = 1 + 0\mathrm{i} + 0\mathrm{j} + 0\mathrm{k}$.

`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(qOnes)`.

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

`quat0nes` — Quaternion ones
scalar | vector | matrix | multidimensional array

Quaternion ones, returned as a scalar, vector, matrix, or multidimensional array of quaternions.

Given a quaternion of the form $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, a quaternion one is defined as $Q = 1 + 0\mathrm{i} + 0\mathrm{j} + 0\mathrm{k}$.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`quaternion` | `zeros`

Topics
“Rotations, Orientation, and Quaternions”

Introduced in R2018a

parts

Extract quaternion parts

Syntax

```
[a,b,c,d] = parts(quat)
```

Description

`[a,b,c,d] = parts(quat)` returns the parts of the quaternion array as arrays, each the same size as `quat`.

Examples

Convert Quaternion to Matrix of Quaternion Parts

Convert a quaternion representation to parts using the `parts` function.

Create a two-element column vector of quaternions by specifying the parts.

```
quat = quaternion([1:4;5:8])
```

```
quat = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

Recover the parts from the quaternion matrix using the `parts` function. The parts are returned as separate output arguments, each the same size as the input 2-by-1 column vector of quaternions.

```
[qA,qB,qC,qD] = parts(quat)
```

```
qA = 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

`classUnderlying` | `quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

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 = 2×3

     1     0     2
     3     2     1
```

```
C = A.^b
```

```
C = 2x3 quaternion array
         1 +    2i +    3j +    4k         1 +    0i +    0j +    0k       -28 +    4i +
     -2110 -  444i -  518j -  592k      -124 +   60i +   70j +   80k         5 +    6i +
```

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 $A = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$ is given by

$$A = \|A\|(\cos\theta + \hat{u}\sin\theta)$$

where θ is the angle of rotation, and $\hat{u}$ is the unit quaternion.

Quaternion A raised by a real exponent b is given by

$$P = A.\hat{}\,b = \|A\|^{b}(\cos(b\theta) + \hat{u}\sin(b\theta))$$

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 = 3x3 quaternion array
      0.53767 +   2.7694i +    1.409j -  0.30344k      0.86217 +   0.7254i -   1.2075j
       1.8339 -   1.3499i +   1.4172j +  0.29387k      0.31877 - 0.063055i +  0.71724j
      -2.2588 +   3.0349i +   0.6715j -  0.78728k      -1.3077 +  0.71474i +   1.6302j
```

Find the product of the quaternions in each column. The length of the first dimension is `1`, and the length of the second dimension matches `size(A,2)`.

```
B = prod(A)
```

```
B = 1x3 quaternion array
     -19.837 -  9.1521i +  15.813j -  19.918k     -5.4708 - 0.28535i +   3.077j -  1.22
```

Product of Specified Dimension of Quaternion Array

You can specify which dimension of a quaternion array to take the product of.

Create a 2-by-2-by-2 quaternion array.

```
A = reshape(quaternion(randn(8,4)),2,2,2);
```

Find the product of the elements in each page of the array. The length of the first dimension matches `size(A,1)`, the length of the second dimension matches `size(A,2)`, and the length of the third dimension is `1`.

```
dim = 3;
B = prod(A,dim)
```

```
B = 2x2 quaternion array
     -2.4847 +  1.1659i - 0.37547j +  2.8068k     0.28786 - 0.29876i - 0.51231j -  4.29
     0.38986 -  3.6606i -  2.0474j -   6.047k      -1.741 - 0.26782i +  5.4346j +  4.14
```

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

`mtimes, *` | `quaternion` | `times, .*`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

rdivide, ./

Element-wise quaternion right division

Syntax

```
C = A./B
```

Description

`C = A./B` performs quaternion element-wise division by dividing each element of quaternion `A` by the corresponding element of quaternion `B`.

Examples

Divide a Quaternion Array by a Real Scalar

Create a 2-by-1 quaternion array, and divide it element-by-element by a real scalar.

```
A = quaternion([1:4;5:8])
```

```
A = 2x1 quaternion array
     1 + 2i + 3j + 4k
     5 + 6i + 7j + 8k
```

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

```
C = 2x1 quaternion array
     0.5 +   1i + 1.5j +   2k
     2.5 +   3i + 3.5j +   4k
```

Divide a Quaternion Array by Another Quaternion Array

Create a 2-by-2 quaternion array, and divide it element-by-element by another 2-by-2 quaternion array.

```
q1 = quaternion(magic(4));
A = reshape(q1,2,2)
```

```
A = 2x2 quaternion array
     16 +  2i +  3j + 13k      9 +  7i +  6j + 12k
      5 + 11i + 10j +  8k      4 + 14i + 15j +  1k
```

```
q2 = quaternion([1:4;3:6;2:5;4:7]);
B = reshape(q2,2,2)
```

```
B = 2x2 quaternion array
     1 + 2i + 3j + 4k     2 + 3i + 4j + 5k
     3 + 4i + 5j + 6k     4 + 5i + 6j + 7k
```

```
C = A./B
```

```
C = 2x2 quaternion array
          2.7 -      0.1i -      2.1j -      1.7k       2.2778 + 0.092593i -  0.46296j
       1.8256 - 0.081395i +  0.45349j -  0.24419k       1.4524 -      0.5i +   1.0238j
```

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 $A = a_1 + a_2\mathrm{i} + a_3\mathrm{j} + a_4\mathrm{k}$ and a real scalar p,

$$C = A\,./p = \frac{a_1}{p} + \frac{a_2}{p}\mathrm{i} + \frac{a_3}{p}\mathrm{j} + \frac{a_4}{p}\mathrm{k}$$

Note For a real scalar p, $A./p = A.\backslash p$.

Quaternion Division by a Quaternion Scalar

Given two quaternions A and B of compatible sizes,

$$C = A./B = A.*B^{-1} = A.*\left(\frac{conj(B)}{norm(B)^2}\right)$$

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

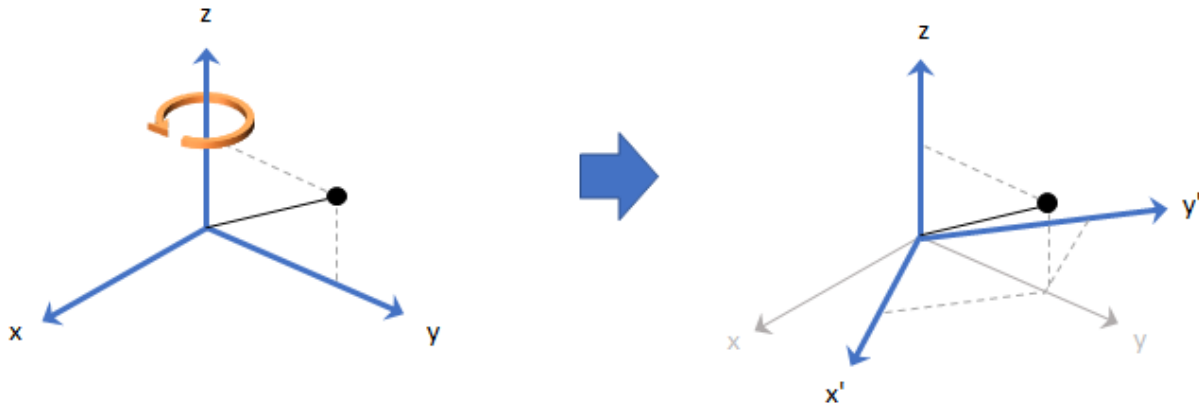
Quaternion frame rotation

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.

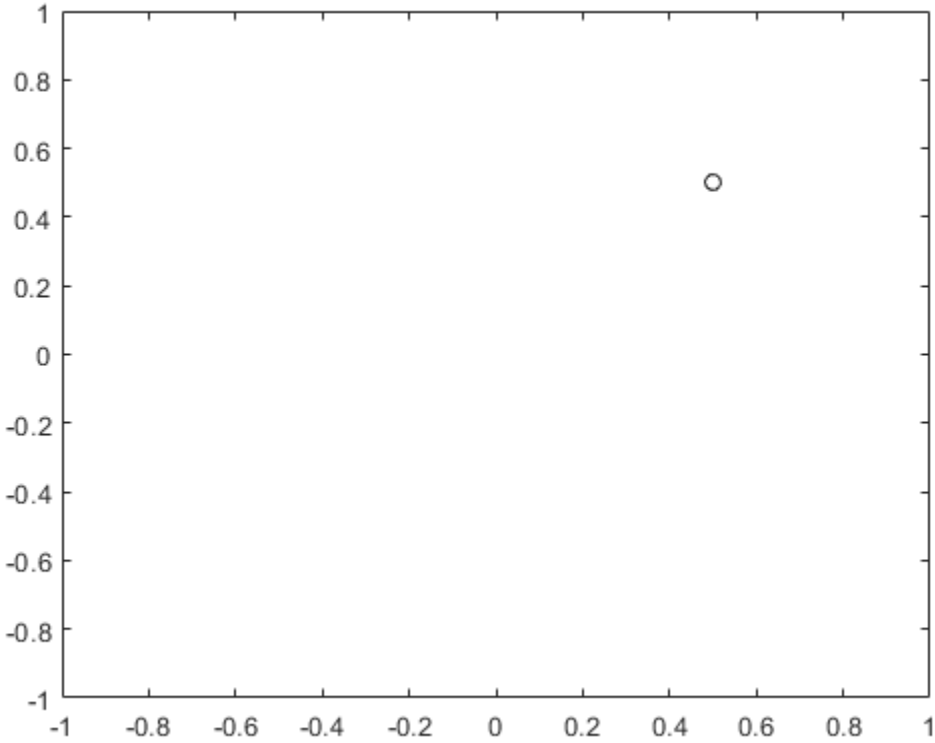


Examples

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])
```



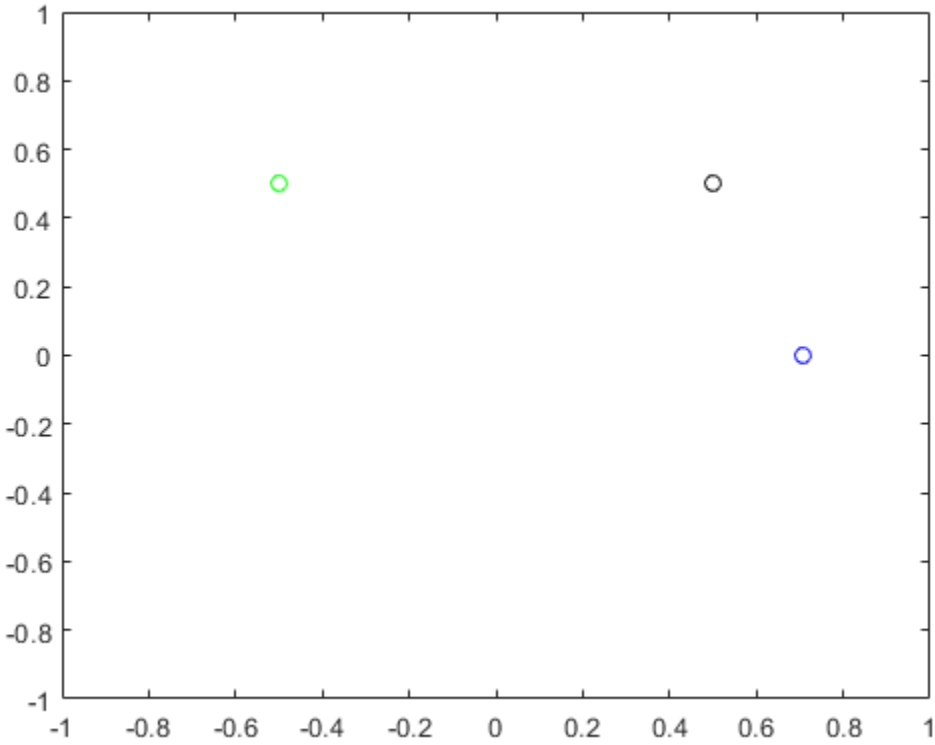
Create a quaternion vector specifying two separate rotations, one to rotate the frame 45 degrees and another to rotate the point -90 degrees about the *z*-axis. Use `rotateframe` to perform the rotations.

```
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');
```

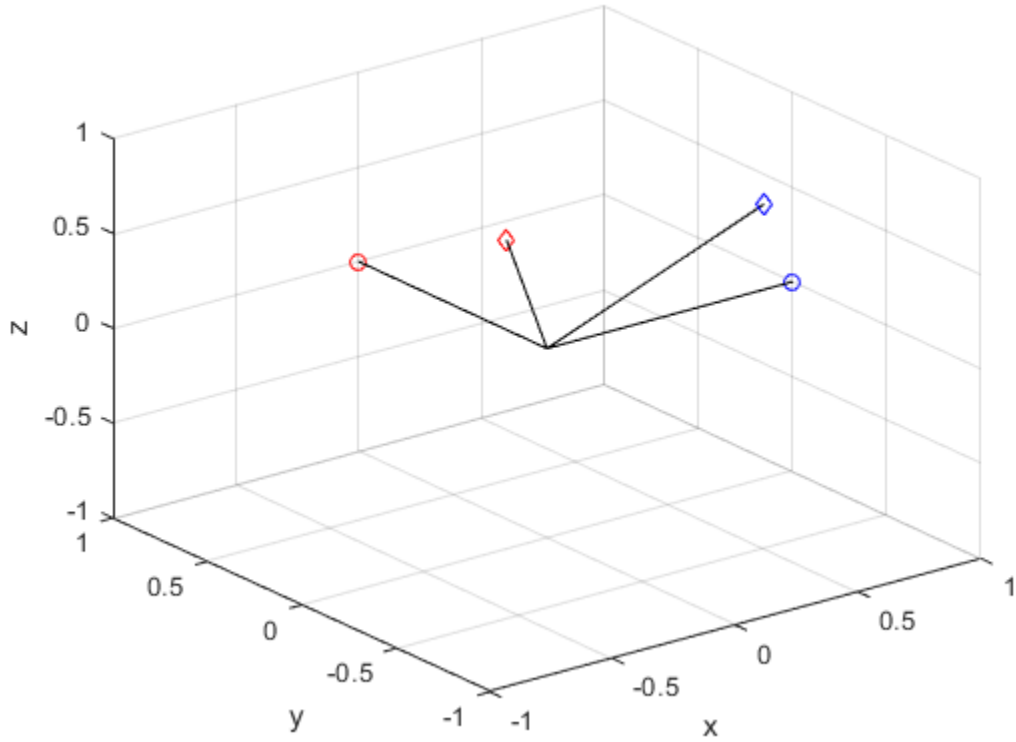
Use `rotateframe` to reference both points using the quaternion rotation operator. Display the result.

```
rP = rotateframe(quat,[a;b])
```

```
rP = 2×3

    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:

$$L_q(u) = q^* u q$$

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, $q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, 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:

$$v_q = q^* u_q q$$

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

`quaternion` | `rotatepoint`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

rotatepoint

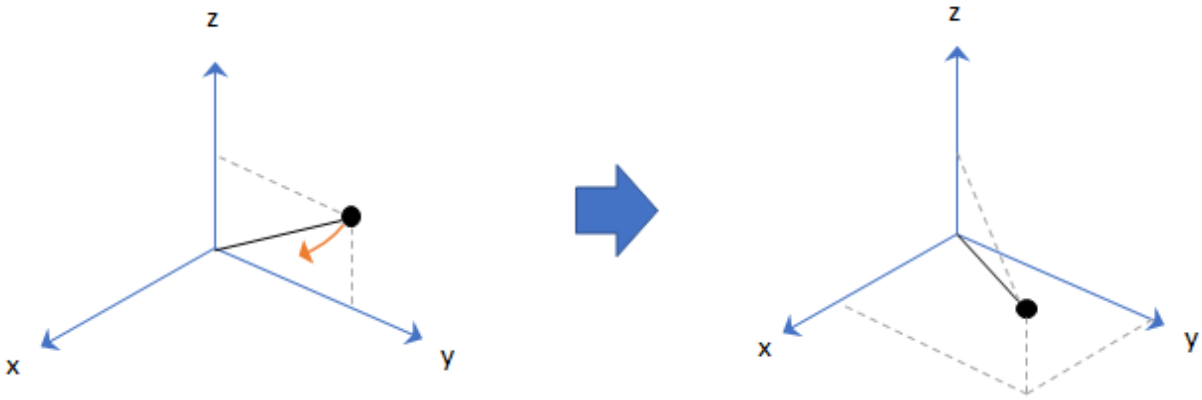
Quaternion point rotation

Syntax

```
rotationResult = rotatepoint(quat,cartesianPoints)
```

Description

`rotationResult = rotatepoint(quat,cartesianPoints)` rotates the Cartesian points using the quaternion, `quat`. The elements of the quaternion are normalized before use in the rotation.

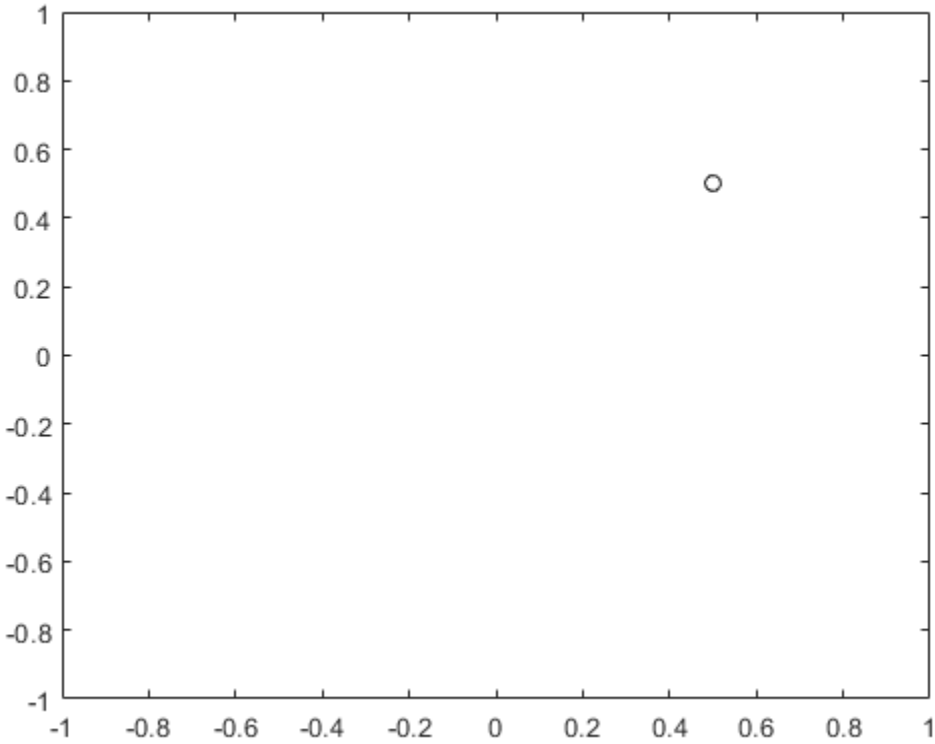


Examples

Rotate Point Using Quaternion Vector

Define a point in three dimensions. The coordinates of a point are always specified in order *x*, *y*, *z*. For convenient visualization, define the point on the *x*-*y* plane.

```
x = 0.5;
y = 0.5;
z = 0;

plot(x,y,'ko')
hold on
axis([-1 1 -1 1])
```



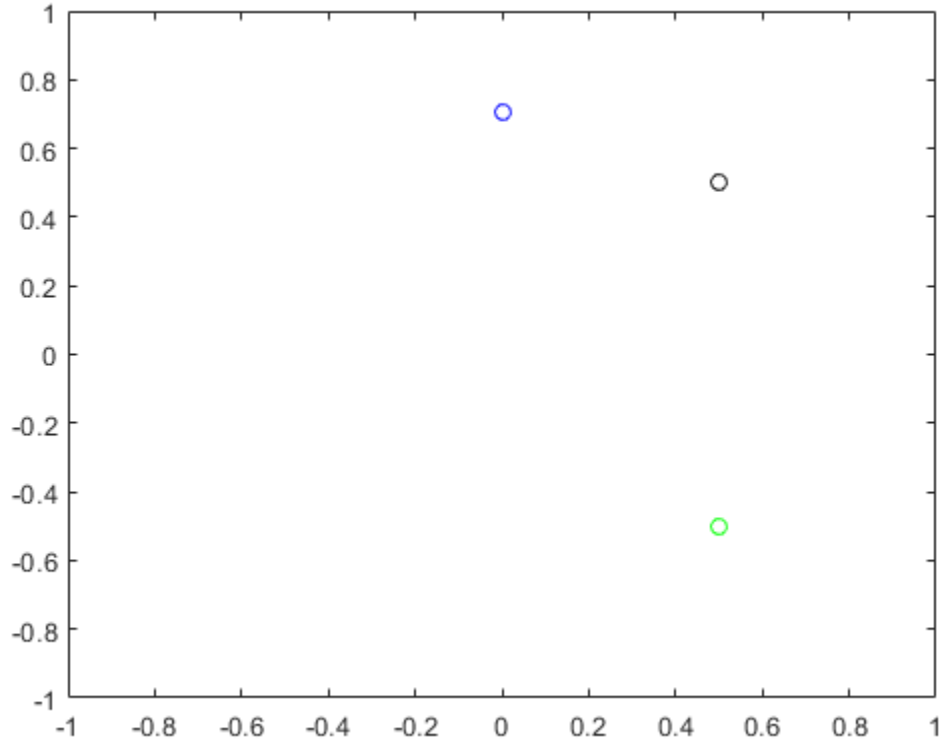
Create a quaternion vector specifying two separate rotations, one to rotate the point 45 and another to rotate the point -90 degrees about the *z*-axis. Use `rotatepoint` to perform the rotation.

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

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

rotatedPoint = 2×3

   -0.0000    0.7071         0
    0.5000   -0.5000         0
```

Plot the rotated points.

```
plot(rotatedPoint(1,1),rotatedPoint(1,2),'bo')
plot(rotatedPoint(2,1),rotatedPoint(2,2),'go')
```



Rotate Group of Points Using Quaternion

Define two points in three-dimensional space. Define a quaternion to rotate the point by first rotating about the *z*-axis 30 degrees and then about the new *y*-axis 45 degrees.

```
a = [1,0,0];
b = [0,1,0];
quat = quaternion([30,45,0],'eulerd','ZYX','point');
```

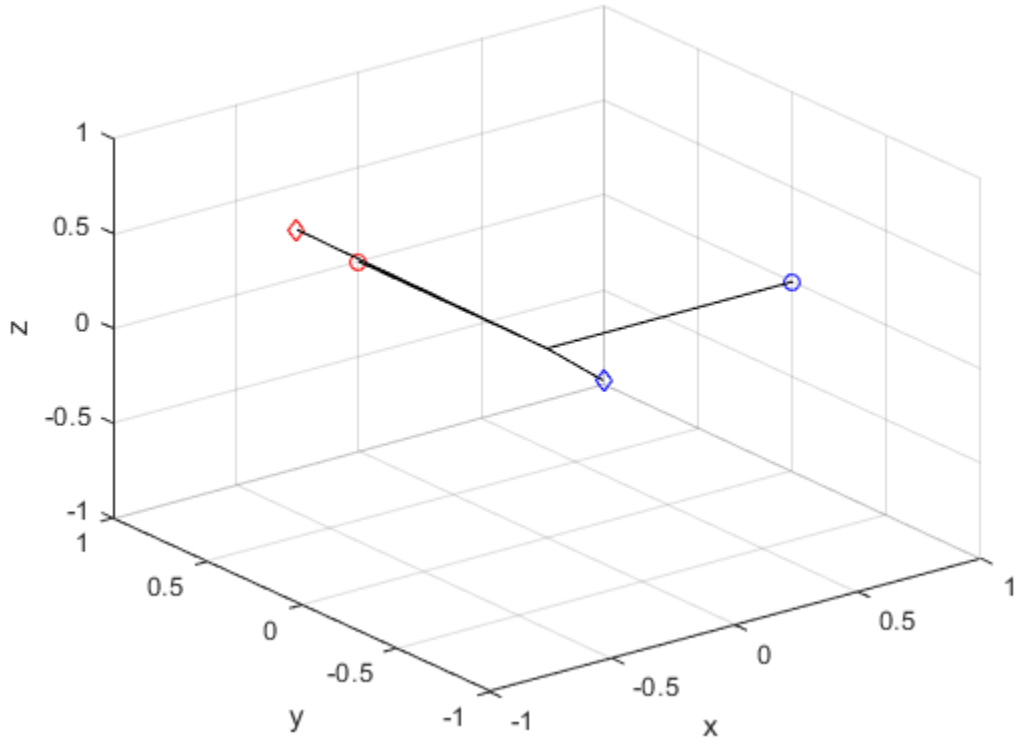
Use `rotatepoint` to rotate both points using the quaternion rotation operator. Display the result.

```
rP = rotatepoint(quat,[a;b])
```

```
rP = 2×3

    0.6124    0.5000   -0.6124
   -0.3536    0.8660    0.3536
```

Visualize the original orientation and the rotated orientation of the points. Draw lines from the origin to each of the points for visualization purposes.

```
plot3(a(1),a(2),a(3),'bo');

hold on
grid on
axis([-1 1 -1 1 -1 1])
xlabel('x')
ylabel('y')
zlabel('z')

plot3(b(1),b(2),b(3),'ro');
plot3(rP(1,1),rP(1,2),rP(1,3),'bd')
plot3(rP(2,1),rP(2,2),rP(2,3),'rd')

plot3([0;rP(1,1)],[0;rP(1,2)],[0;rP(1,3)],'k')
plot3([0;rP(2,1)],[0;rP(2,2)],[0;rP(2,3)],'k')
plot3([0;a(1)],[0;a(2)],[0;a(3)],'k')
plot3([0;b(1)],[0;b(2)],[0;b(3)],'k')
```



Input Arguments

`quat` — Quaternion that defines rotation

scalar | vector

Quaternion that defines rotation, specified as a scalar quaternion, row vector of quaternions, or column vector of quaternions.

Data Types: `quaternion`

`cartesianPoints` — Three-dimensional Cartesian points

1-by-3 vector | *N*-by-3 matrix

Three-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, $q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, 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:

$v_q = qu_q q^*$

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

`quaternion` | `rotateframe`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

rotmat

Convert quaternion to rotation matrix

Syntax

```
rotationMatrix = rotmat(quat,rotationType)
```

Description

`rotationMatrix = rotmat(quat,rotationType)` converts the quaternion, `quat`, to an equivalent rotation matrix representation.

Examples

Convert Quaternion to Rotation Matrix for Point Rotation

Define a quaternion for use in point rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma],'eulerd','ZYX','point')
```

```
quat = quaternion
       0.8924 +  0.23912i +  0.36964j + 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat,'point')
```

```
rotationMatrix = 3×3

    0.7071   -0.0000    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the *y*- and *x*-axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)   0           sind(theta) ; ...
      0             1           0           ; ...
     -sind(theta)   0           cosd(theta)];

rx = [1             0           0           ;      ...
      0             cosd(gamma) -sind(gamma) ;     ...
      0             sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry
```

```
rotationMatrixVerification = 3×3

    0.7071         0    0.7071
    0.3536    0.8660   -0.3536
   -0.6124    0.5000    0.6124
```

Convert Quaternion to Rotation Matrix for Frame Rotation

Define a quaternion for use in frame rotation.

```
theta = 45;
gamma = 30;
quat = quaternion([0,theta,gamma],'eulerd','ZYX','frame')
```

```
quat = quaternion
       0.8924 +  0.23912i +  0.36964j - 0.099046k
```

Convert the quaternion to a rotation matrix.

```
rotationMatrix = rotmat(quat,'frame')
```

```
rotationMatrix = 3×3
```

```
    0.7071   -0.0000   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

To verify the rotation matrix, directly create two rotation matrices corresponding to the rotations about the *y*- and *x*-axes. Multiply the rotation matrices and compare to the output of `rotmat`.

```
theta = 45;
gamma = 30;

ry = [cosd(theta)   0           -sind(theta) ; ...
      0             1           0            ; ...
     sind(theta)   0           cosd(theta)];

rx = [1             0           0           ;      ...
      0             cosd(gamma) sind(gamma) ;     ...
      0             -sind(gamma) cosd(gamma)];

rotationMatrixVerification = rx*ry
```

```
rotationMatrixVerification = 3×3

    0.7071         0   -0.7071
    0.3536    0.8660    0.3536
    0.6124   -0.5000    0.6124
```

Convert Quaternion Vector to Rotation Matrices

Create a 3-by-1 normalized quaternion vector.

```
qVec = normalize(quaternion(randn(3,4)));
```

Convert the quaternion array to rotation matrices. The pages of `rotmatArray` correspond to the linear index of `qVec`.

```
rotmatArray = rotmat(qVec,'frame');
```

Assume `qVec` and `rotmatArray` correspond to a sequence of rotations. Combine the quaternion rotations into a single representation, then apply the quaternion rotation to arbitrarily initialized Cartesian points.

```
loc = normalize(randn(1,3));
quat = prod(qVec);
rotateframe(quat,loc)
```

```
ans = 1×3

    0.9524    0.5297    0.9013
```

Combine the rotation matrices into a single representation, then apply the rotation matrix to the same initial Cartesian points. Verify the quaternion rotation and rotation matrix result in the same orientation.

```
totalRotMat = eye(3);
for i = 1:size(rotmatArray,3)
    totalRotMat = rotmatArray(:,:,i)*totalRotMat;
end
totalRotMat*loc'
```

```
ans = 3×1

    0.9524
    0.5297
    0.9013
```

Input Arguments

`quat` — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

`rotationType` — Type or rotation

`'frame'` | `'point'`

Type of rotation represented by the `rotationMatrix` output, specified as `'frame'` or `'point'`.

Data Types: `char` | `string`

Output Arguments

`rotationMatrix` — Rotation matrix representation

3-by-3 matrix | 3-by-3-by-*N* multidimensional array

Rotation matrix representation, returned as a 3-by-3 matrix or 3-by-3-by-*N* multidimensional array.

- If `quat` is a scalar, `rotationMatrix` is returned as a 3-by-3 matrix.
- If `quat` is non-scalar, `rotationMatrix` is returned as a 3-by-3-by-*N* multidimensional array, where `rotationMatrix(:,:,i)` is the rotation matrix corresponding to `quat(i)`.

The data type of the rotation matrix is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

Given a quaternion of the form

$$q = a + bi + cj + dk,$$

the equivalent rotation matrix for frame rotation is defined as

$$\begin{bmatrix} 2a^2 - 1 + 2b^2 & 2bc + 2ad & 2bd - 2ac \\ 2bc - 2ad & 2a^2 - 1 + 2c^2 & 2cd + 2ab \\ 2bd + 2ac & 2cd - 2ab & 2a^2 - 1 + 2d^2 \end{bmatrix}.$$

The equivalent rotation matrix for point rotation is the transpose of the frame rotation matrix:

$$\begin{bmatrix} 2a^2-1+2b^2 & 2bc-2ad & 2bd+2ac \\ 2bc+2ad & 2a^2-1+2c^2 & 2cd-2ab \\ 2bd-2ac & 2cd+2ab & 2a^2-1+2d^2 \end{bmatrix}.$$

References

[1] Kuipers, Jack B. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton, NJ: Princeton University Press, 2007.

Extended Capabilities

C/C++ Code Generation

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

See Also

`euler` | `eulerd` | `quaternion` | `rotvec` | `rotvecd`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

rotvec

Convert quaternion to rotation vector (radians)

Syntax

```
rotationVector = rotvec(quat)
```

Description

`rotationVector = rotvec(quat)` converts the quaternion array, `quat`, to an *N*-by-3 matrix of equivalent rotation vectors in radians. The elements of `quat` are normalized before conversion.

Examples

Convert Quaternion to Rotation Vector in Radians

Convert a random quaternion scalar to a rotation vector in radians

```
quat = quaternion(randn(1,4));
rotvec(quat)
```

```
ans = 1×3

    1.6866   -2.0774    0.7929
```

Input Arguments

quat — Quaternion to convert

scalar | vector | matrix | multidimensional array

Quaternion to convert, specified as scalar quaternion, vector, matrix, or multidimensional array of quaternions.

Data Types: `quaternion`

Output Arguments

`rotationVector` — Rotation vector (radians)
N-by-3 matrix

Rotation vector representation, returned as an *N*-by-3 matrix of rotations vectors, where each row represents the [X Y Z] angles of the rotation vectors in radians. The *i*th row of `rotationVector` corresponds to the element `quat(i)`.

The data type of the rotation vector is the same as the underlying data type of `quat`.

Data Types: `single` | `double`

Algorithms

All rotations in 3-D can be represented by a three-element axis of rotation and a rotation angle, for a total of four elements. If the rotation axis is constrained to be unit length, the rotation angle can be distributed over the vector elements to reduce the representation to three elements.

Recall that a quaternion can be represented in axis-angle form

$$q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(\mathrm{xi} + y\mathrm{j} + z\mathrm{k}),$$

where θ 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 θ over the parts b, c, and d. The rotation vector representation of q is

$$q_{\text{rv}} = \frac{\theta}{\sin\left(\theta/2\right)}[b, c, d]\,.$$

Extended Capabilities

C/C++ Code Generation

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

See Also

`euler` | `eulerd` | `quaternion` | `rotvecd`

Topics

"Rotations, Orientation, and Quaternions"

Introduced in R2018a

times, .*

Element-wise quaternion multiplication

Syntax

```
quatC = A.*B
```

Description

`quatC = A.*B` returns the element-by-element quaternion multiplication of quaternion arrays.

You can use quaternion multiplication to compose rotation operators:

- To compose a sequence of frame rotations, multiply the quaternions in the same order as the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the order pq. The rotation operator becomes $(pq)^{*}v(pq)$, where v represents the object to rotate in quaternion form. * represents conjugation.
- To compose a sequence of point rotations, multiply the quaternions in the reverse order of the desired sequence of rotations. For example, to apply a p quaternion followed by a q quaternion, multiply in the reverse order, qp. The rotation operator becomes $(qp)v(qp)^{*}$.

Examples

Multiply Two Quaternion Vectors

Create two vectors, `A` and `B`, and multiply them element by element.

```
A = quaternion([1:4;5:8]);
B = A;
C = A.*B
```

```
C = 2x1 quaternion array
     -28 +   4i +   6j +   8k
    -124 +  60i +  70j +  80k
```

Multiply Two Quaternion Arrays

Create two 3-by-3 arrays, A and B, and multiply them element by element.

```
A = reshape(quaternion(randn(9,4)),3,3);
B = reshape(quaternion(randn(9,4)),3,3);
C = A.*B
```

```
C = 3x3 quaternion array
     0.60169 +  2.4332i -  2.5844j + 0.51646k    -0.49513 +  1.1722i +  4.4401j -   1.2
     -4.2329 +  2.4547i +  3.7768j + 0.77484k    -0.65232 - 0.43112i -  1.4645j - 0.900
     -4.4159 +  2.1926i +  1.9037j -  4.0303k     -2.0232 +  0.4205i - 0.17288j +  3.85
```

Note that quaternion multiplication is not commutative:

```
isequal(C,B.*A)
```

```
ans = logical
   0
```

Multiply Quaternion Row and Column Vectors

Create a row vector a and a column vector b, then multiply them. The 1-by-3 row vector and 4-by-1 column vector combine to produce a 4-by-3 matrix with all combinations of elements multiplied.

```
a = [zeros('quaternion'),ones('quaternion'),quaternion(randn(1,4))]
```

```
a = 1x3 quaternion array
           0 +       0i +       0j +       0k           1 +       0i +       0j +
```

```
b = quaternion(randn(4,4))
```

```
b = 4x1 quaternion array
      0.31877 +   3.5784i +   0.7254j -  0.12414k
      -1.3077 +   2.7694i - 0.063055j +   1.4897k
     -0.43359 -   1.3499i +  0.71474j +    1.409k
      0.34262 +   3.0349i -  0.20497j +   1.4172k


a.*b

ans = 4x3 quaternion array
            0 +        0i +        0j +        0k      0.31877 +   3.5784i +   0.7254j
            0 +        0i +        0j +        0k      -1.3077 +   2.7694i - 0.063055j
            0 +        0i +        0j +        0k     -0.43359 -   1.3499i +  0.71474j
            0 +        0i +        0j +        0k      0.34262 +   3.0349i -  0.20497j
```

Input Arguments

A — Array to multiply

scalar | vector | matrix | multidimensional array

Array to multiply, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of them is 1.

Data Types: `quaternion` | `single` | `double`

B — Array to multiply

scalar | vector | matrix | multidimensional array

Array to multiply, specified as a quaternion, an array of quaternions, a real scalar, or an array of real numbers.

A and B must have compatible sizes. In the simplest cases, they can be the same size or one can be a scalar. Two inputs have compatible sizes if, for every dimension, the dimension sizes of the inputs are the same or one of them is 1.

Data Types: `quaternion` | `single` | `double`

Output Arguments

quatC — Quaternion product
scalar | vector | matrix | multidimensional array

Quaternion product, returned as a scalar, vector, matrix, or multidimensional array.

Data Types: `quaternion`

Algorithms

Quaternion Multiplication by a Real Scalar

Given a quaternion,

$$q = a_q + b_q i + c_q j + d_q k,$$

the product of q and a real scalar β is

$$\beta q = \beta a_q + \beta b_q i + \beta c_q j + \beta d_q k$$

Quaternion Multiplication by a Quaternion Scalar

The definition of the basis elements for quaternions,

$$i^2 = j^2 = k^2 = ijk = -1,$$

can be expanded to populate a table summarizing quaternion basis element multiplication:

	1	**i**	**j**	**k**
1	1	i	j	k
i	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, $q = a_q + b_q\mathrm{i} + c_q\mathrm{j} + d_q\mathrm{k}$, and $p = a_p + b_p\mathrm{i} + c_p\mathrm{j} + d_p\mathrm{k}$, the multiplication can be expanded as:

$$\begin{aligned} z = pq &= \left(a_p + b_p\mathrm{i} + c_p\mathrm{j} + d_p\mathrm{k}\right)\left(a_q + b_q\mathrm{i} + c_q\mathrm{j} + d_q\mathrm{k}\right) \\ &= a_p a_q + a_p b_q\mathrm{i} + a_p c_q\mathrm{j} + a_p d_q\mathrm{k} \\ &+ b_p a_q\mathrm{i} + b_p b_q\mathrm{i}^2 + b_p c_q\mathrm{ij} + b_p d_q\mathrm{ik} \\ &+ c_p a_q\mathrm{j} + c_p b_q\mathrm{ji} + c_p c_q\mathrm{j}^2 + c_p d_q\mathrm{jk} \\ &+ d_p a_q k + d_p b_q\mathrm{ki} + d_p c_q\mathrm{kj} + d_p d_q\mathrm{k}^2 \end{aligned}$$

You can simplify the equation using the quaternion multiplication table.

$$\begin{aligned} z = pq &= a_p a_q + a_p b_q\mathrm{i} + a_p c_q\mathrm{j} + a_p d_q\mathrm{k} \\ &+ b_p a_q\mathrm{i} - b_p b_q + b_p c_q\mathrm{k} - b_p d_q\mathrm{j} \\ &+ c_p a_q\mathrm{j} - c_p b_q\mathrm{k} - c_p c_q + c_p d_q\mathrm{i} \\ &+ d_p a_q k + d_p b_q\mathrm{j} - d_p c_q\mathrm{i} - d_p d_q \end{aligned}$$

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

`mtimes, *` | `prod` | `quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

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
```

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


quatTransposed = quat.'

quatTransposed = 2x2 quaternion array
      0.53767 -   2.2588i +  0.31877j -  0.43359k       1.8339 +  0.86217i -   1.3077j
       3.5784 -   1.3499i +   0.7254j +  0.71474k       2.7694 +   3.0349i - 0.063055j
```

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

`ctranspose` | `quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

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

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

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

`minus, -` | `quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

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 $Q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, a quaternion zero is defined as $Q = 0 + 0\mathrm{i} + 0\mathrm{j} + 0\mathrm{k}$.

Data Types: `quaternion`

Extended Capabilities

C/C++ Code Generation

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

See Also

`ones` | `quaternion`

Topics

“Rotations, Orientation, and Quaternions”

Introduced in R2018a

constvel

Constant velocity state update

Syntax

```
updatedstate = constvel(state)
updatedstate = constvel(state,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`.

Examples

Update State for Constant-Velocity Motion

Update the state of two-dimensional constant-velocity motion for a time interval of one second.

```
state = [1;1;2;1];
state = constvel(state)
```

```
state = 4×1

     2
     1
     3
     1
```

Update State for Constant-Velocity Motion with Specified Time Step

Update the state of two-dimensional constant-velocity motion for a time interval of 1.5 seconds.

```
state = [1;1;2;1];
state = constvel(state,1.5)
```

```
state = 4×1

    2.5000
    1.0000
    3.5000
    1.0000
```

Input Arguments

`state` — Kalman filter state 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. 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`

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

Classes
trackingCKF | trackingEKF | trackingKF | trackingMSCEKF | trackingPF | trackingUKF

Introduced in R2018b

constveljac

Jacobian for constant-velocity motion

Syntax

```
jacobian = constveljac(state)
jacobian = constveljac(state,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`.

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 = 4×4

    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. 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`

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.

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`

Classes

`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 = 3×1

     1
     2
     0
```

The *z*-component of the measurement is zero.

Create Measurement from Constant Velocity Object in Spherical Frame

Define the state of an object in 2-D constant-velocity motion. The state is the position and velocity in each spatial dimension. The measurements are in spherical coordinates.

```
state = [1;10;2;20];
measurement = cvmeas(state,'spherical')
```

```
measurement = 4×1

   63.4349
         0
    2.2361
   22.3607
```

The elevation of the measurement is zero and the range rate is positive. These results indicate that the object is moving away from the sensor.

Create Measurement from Constant-Velocity Object in Translated Spherical Frame

Define the state of an object in 2-D constant-velocity motion. The state consists of position and velocity in each spatial dimension. The measurements are in spherical coordinates with respect to a frame located at *(20;40;0)* meters.

```
state = [1;10;2;20];
measurement = cvmeas(state,'spherical',[20;40;0])
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -22.3607
```

The elevation of the measurement is zero and the range rate is negative. These results indicate that the object is moving toward the sensor.

Create Measurement from Constant-Velocity Object Using Measurement Parameters

Define the state of an object in 2-D constant-velocity motion. The state consists of position and velocity in each spatial dimension. The measurements are in spherical coordinates with respect to a frame located at *(20;40;0)* meters.

```
state2d = [1;10;2;20];
frame = 'spherical';
sensorpos = [20;40;0];
sensorvel = [0;5;0];
laxes = eye(3);
measurement = cvmeas(state2d,frame,sensorpos,sensorvel,laxes)
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -17.8885
```

The elevation of the measurement is zero and the range rate is negative. These results indicate that the object is moving toward the sensor.

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

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel,
    'Orientation',laxes);
measurement = cvmeas(state2d,measparm)
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -17.8885
```

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

Measurement frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of the *x*, *y*, and *z* Cartesian coordinates of the tracked object. When specified as `'spherical'`, a measurement consists of the azimuth, elevation, range, and range rate of the tracked object.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the global coordinate system, 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 global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters/second.

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 global coordinate system.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure

Measurement parameters, specified as a structure. The fields of the structure are:

measurementParameters struct

Parameter	Definition	Default
`OriginPosition`	Sensor position with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters.	`[0;0;0]`
`OriginVelocity`	Sensor velocity with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in m/s.	`[0;0;0]`
`Orientation`	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 global coordinate system.	`eye(3)`
`HasVelocity`	Indicates whether measurements contain velocity or range rate components, specified as `true` or `false`.	`false` when `frame` argument is `'rectangular'` and `true` when `frame` argument is `'spherical'`
`HasElevation`	Indicates whether measurements contain elevation components, specified as `true` or `false`.	`true`

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.

<table>
<tr><th>frame</th><th>measurement</th></tr>
<tr><td>'spherical'</td><td>Specifies the azimuth angle, az, elevation angle, el, range, r, and range rate, rr, of the object with respect to the local ego vehicle coordinate system. Positive values for range rate indicate that an object is moving away from the sensor.

Spherical measurements
<table><tr><th></th><th></th><th colspan="2">HasElevation</th></tr><tr><td></td><td></td><td>false</td><td>true</td></tr><tr><td rowspan="2">HasVelocity</td><td>false</td><td>[az;r]</td><td>[az;el;r]</td></tr><tr><td>true</td><td>[az;r;rr]</td><td>[az;el;r;rr]</td></tr></table>
Angle units are in degrees, range units are in meters, and range rate units are in m/s.</td></tr>
</table>

<table>
<tr><th>frame</th><th>measurement</th></tr>
<tr><td>'rectangular</td><td>Specifies the Cartesian position and velocity coordinates of the tracked object with respect to the ego vehicle coordinate system.

Rectangular measurements
<table><tr><td rowspan="2">HasVelocity</td><td>false</td><td>[x;y;y]</td></tr><tr><td>true</td><td>[x;vx;y,vy;z;vz]</td></tr></table>
Position units are in meters and velocity units are in m/s.</td></tr>
</table>

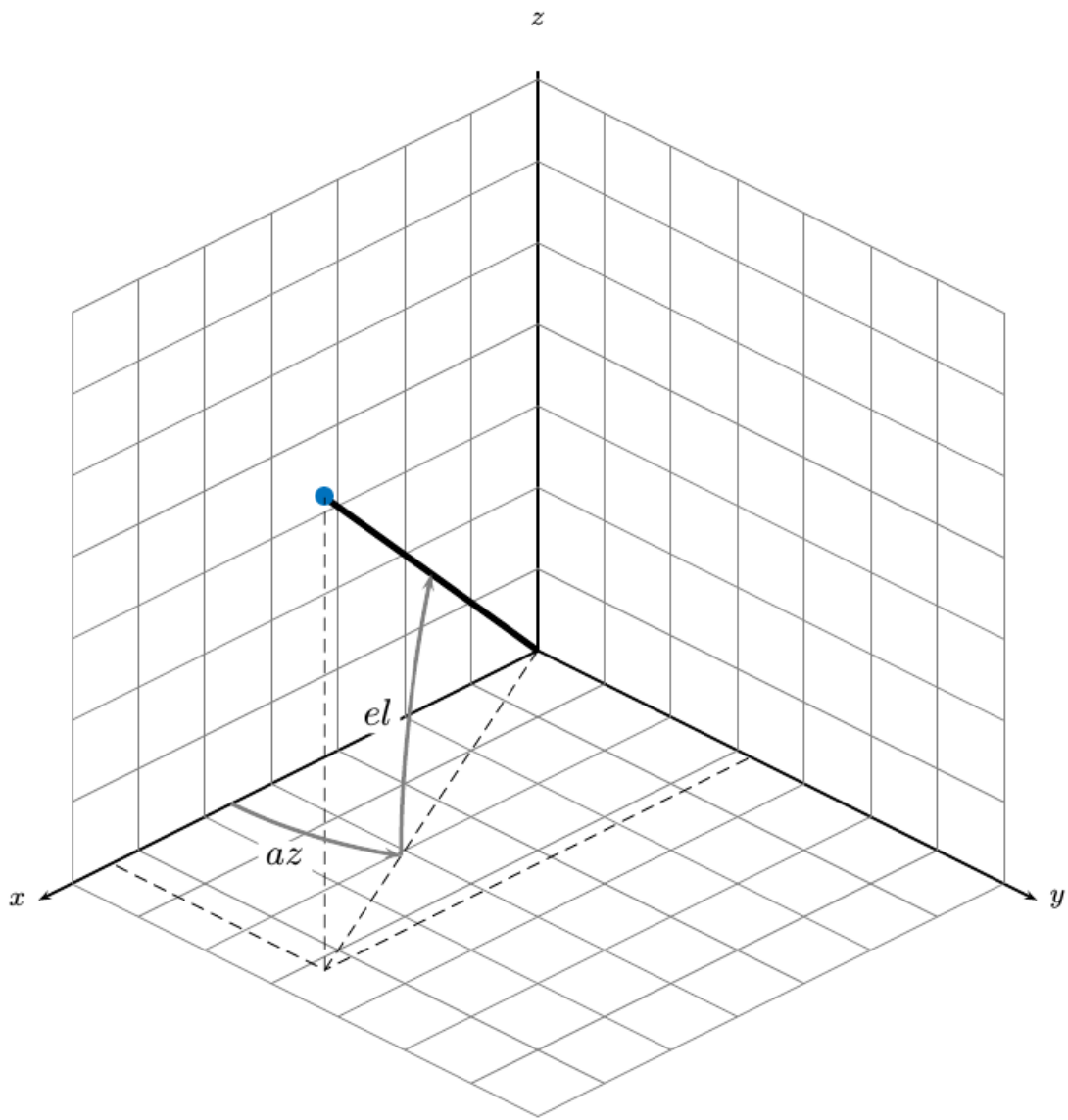
Data Types: `double`

Definitions

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.


z
el
az
x
y

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

Classes

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 = 3×4

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

Measurement Jacobian of Constant-Velocity Motion in Spherical Frame

Define the state of an object in 2-D constant-velocity motion. The state is the position and velocity in each dimension. Compute the measurement Jacobian with respect to spherical coordinates.

```
state = [1;10;2;20];
measurementjac = cvmeasjac(state,'spherical')
```

```
measurementjac = 4×4

  -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 = 4×4

   -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 = 4×4

    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 = 4×4
```

```
    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. 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 frame

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

Measurement frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of the *x*, *y*, and *z* Cartesian coordinates of the tracked object. When specified as `'spherical'`, a measurement consists of the azimuth, elevation, range, and range rate of the tracked object.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the global coordinate system, 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 global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters/second.

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 global coordinate system.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure

Measurement parameters, specified as a structure. The fields of the structure are:

measurementParameters struct

Parameter	Definition	Default
`OriginPosition`	Sensor position with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters.	`[0;0;0]`
`OriginVelocity`	Sensor velocity with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in m/s.	`[0;0;0]`
`Orientation`	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 global coordinate system.	`eye(3)`
`HasVelocity`	Indicates whether measurements contain velocity or range rate components, specified as `true` or `false`.	`false` when `frame` argument is `'rectangular'` and `true` when `frame` argument is `'spherical'`
`HasElevation`	Indicates whether measurements contain elevation components, specified as `true` or `false`.	`true`

Data Types: `struct`

Output Arguments

`measurementjac` — Measurement Jacobian

real-valued 3-by-*N* matrix | real-valued 4-by-*N* matrix

Measurement Jacobian, specified as a real-valued 3-by-*N* or 4-by-*N* matrix. *N* is the dimension of the state vector. The first dimension and meaning depend on value of the `frame` argument.

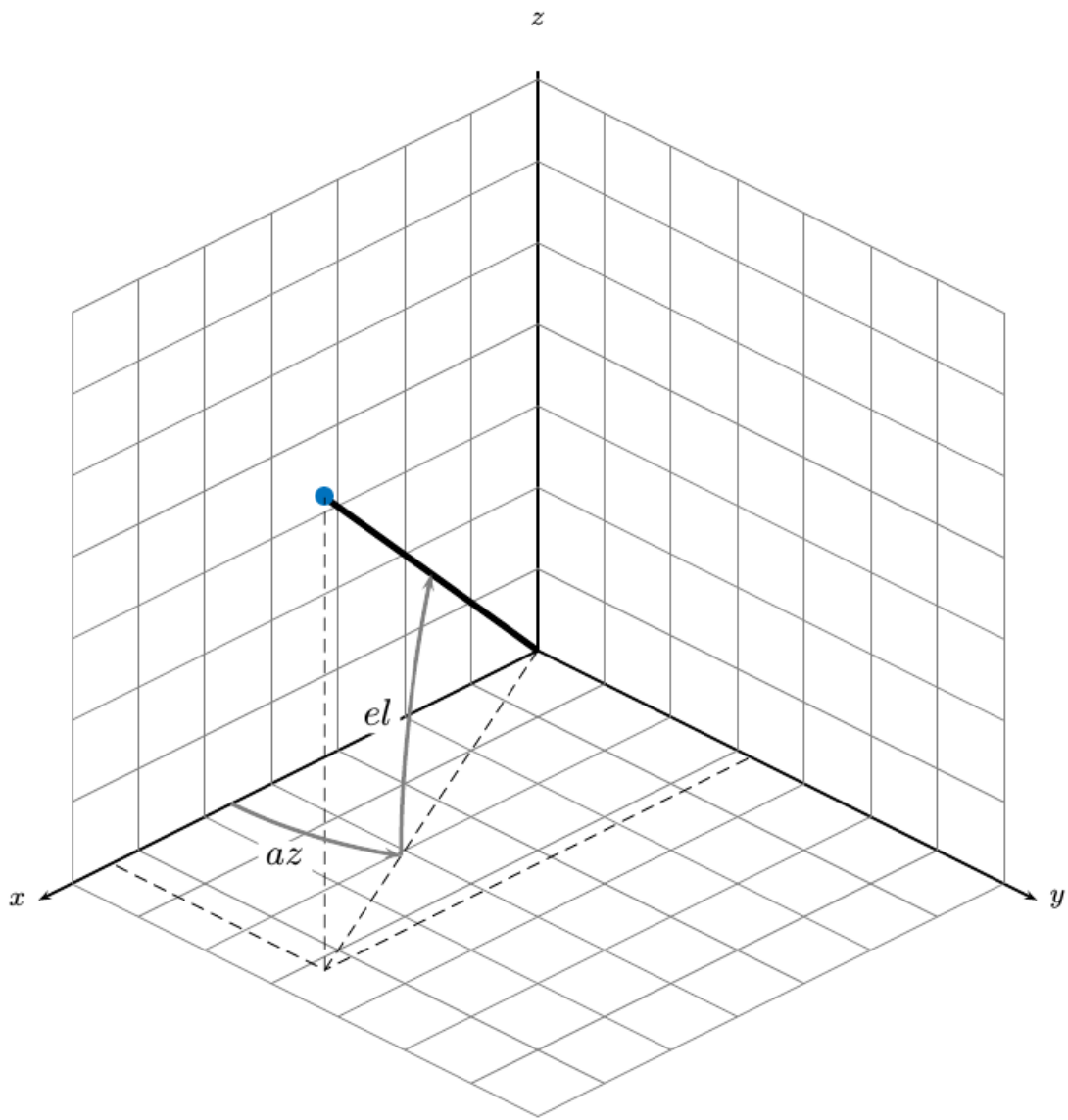
Frame	Measurement Jacobian
`'rectangular'`	Jacobian of the measurements `[x;y;z]` with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
`'spherical'`	Jacobian of the measurement vector `[az;el;r;rr]` with respect to the state vector. Measurement vector components specify the azimuth angle, elevation angle, range, and range rate of the object with respect to the local sensor coordinate system. Angle units are in degrees. Range units are in meters and range rate units are in meters/second.

Definitions

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.


z
el
az
x
y

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`

Classes

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

Introduced in R2018b

constacc

Constant-acceleration motion model

Syntax

```
updatedstate = constacc(state)
updatedstate = constacc(state,dt)
```

Description

`updatedstate = constacc(state)` returns the updated state, `state`, of a constant velocity Kalman filter motion model for a step time of one second.

`updatedstate = constacc(state,dt)` specifies the time step, `dt`.

Examples

Predict State for Constant-Acceleration Motion

Define an initial state for 2-D constant-acceleration motion.

```
state = [1;1;1;2;1;0];
```

Predict the state 1 second later.

```
state = constacc(state)
```

```
state = 6×1

    2.5000
    2.0000
    1.0000
    3.0000
    1.0000
```

```
     0
```

Predict State for Constant-Acceleration Motion With Specified Time Step

Define an initial state for 2-D constant-acceleration motion.

```
state = [1;1;1;2;1;0];
```

Predict the state 0.5 s later.

```
state = constacc(state,0.5)
```

```
state = 6×1

    1.6250
    1.5000
    1.0000
    2.5000
    1.0000
         0
```

Input Arguments

`state` — Kalman filter state 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^2.

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`

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`

Classes

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

Introduced in R2018b

constaccjac

Jacobian for constant-acceleration motion

Syntax

```
jacobian = constaccjac(state)
jacobian = constaccjac(state,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`.

Examples

Compute State Jacobian for Constant-Acceleration Motion

Compute the state Jacobian for two-dimensional constant-acceleration motion.

Define an initial state and compute the state Jacobian for a one second update time.

```
state = [1,1,1,2,1,0];
jacobian = constaccjac(state)
```

```
jacobian = 6×6

    1.0000    1.0000    0.5000         0         0         0
         0    1.0000    1.0000         0         0         0
         0         0    1.0000         0         0         0
         0         0         0    1.0000    1.0000    0.5000
         0         0         0         0    1.0000    1.0000
```

```
         0         0         0         0         0    1.0000
```

Compute State Jacobian for Constant-Acceleration Motion with Specified Time Step

Compute the state Jacobian for two-dimensional constant-acceleration motion. Set the step time to 0.5 seconds.

```
state = [1,1,1,2,1,0].';
jacobian = constaccjac(state,0.5)
```

```
jacobian = 6×6

    1.0000    0.5000    0.1250         0         0         0
         0    1.0000    0.5000         0         0         0
         0         0    1.0000         0         0         0
         0         0         0    1.0000    0.5000    0.1250
         0         0         0         0    1.0000    0.5000
         0         0         0         0         0    1.0000
```

Input Arguments

`state` — Kalman filter state vector

real-valued *3N*-element vector

Kalman filter state vector for constant-acceleration motion, specified as a real-valued *3N*-element vector. *N* is the number of spatial degrees of freedom of motion. For each spatial degree of motion, the state vector takes the form shown in this table.

Spatial Dimensions	State Vector Structure
1-D	`[x;vx;ax]`
2-D	`[x;vx;ax;y;vy;ay]`
3-D	`[x;vx;ax;y;vy;ay;z;vz;az]`

For example, `x` represents the *x*-coordinate, `vx` represents the velocity in the *x*-direction, and `ax` represents the acceleration in the *x*-direction. If the motion model is in one-

dimensional space, the *y*- and *z*-axes are assumed to be zero. If the motion model is in two-dimensional space, values along the *z*-axis are assumed to be zero. Position coordinates are in meters. Velocity coordinates are in meters/second. Acceleration coordinates are in meters/second^2.

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`

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.

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`

Classes

`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 = 6×1

    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 = 6×1

    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];
```

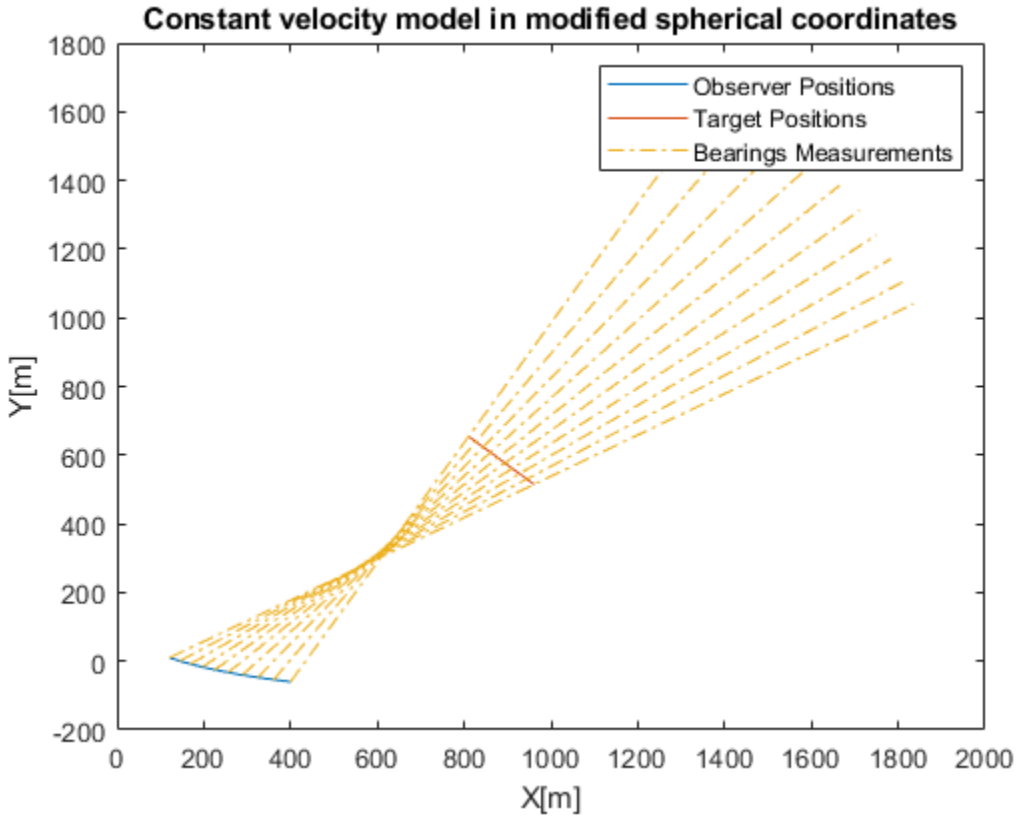
Pre-allocate memory. `rPlot` is the range for plotting bearing measurements.

```
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, `cvmeasmsc`.

```
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 = cvmeasmsc(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 cvmeasmsc function.
    relativePosition = cvmeasmsc(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]'); ylab
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* × 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 Systems”.

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
`trackingMSCEKF`

Classes
`trackingEKF`

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 = 4×4

    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 = 4×2
10^-3 ×

   -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 = 4×4

    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 = 4×2
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 = 4×4

    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 = 4×2
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* × 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

Classes
`trackingEKF`

Functions
`constvelmsc`

Introduced in R2018b

cvmeasmsc

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

Syntax

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

Description

`measurement = cvmeasmsc(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 `cvmeasmsc` function can be used as the `MeasurementFcn`. To use this `MeasurementFcn` with `trackerGNN` and `trackerTOMHT`, you can use the `trackingMSCEKF` filter.

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

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

Examples

Obtain Measurements in MSC Frame

Using the `cvmeasmsc` 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];
cvmeasmsc(mscState)
```

```
ans = 2×1

   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 = 3×1

  838.3866
  458.0127
  295.5202
```

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

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

```
ans = 3×1

  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* × 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

cvmeasmscjac

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

Syntax

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

Description

`jacobian = cvmeasmscjac(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 `cvmeasmscjac` function can be used as the `MeasurementJacobianFcn`. To use this `MeasurementFcn` with `trackerGNN` and `trackerTOMHT`, you can use the `trackingMSCEKF` filter.

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

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

Examples

Obtain Jacobian of State Measurements in MSC Frame

Using the `cvmeasmscjac` 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];
cvmeasmscjac(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.

```
cvmeasmscjac(mscState,'rectangular')
```

```
ans = 3×6
10^5 ×

   -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`.

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

```
ans = 3×6
10^5 ×

   -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* × 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'` or an enum 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

`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 = 3×1

     1
     2
     0
```

The measurement is returned in three-dimensions with the *z*-component set to zero.

Create Measurement from Accelerating Object in Spherical Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. The measurements are in spherical coordinates.

```
state = [1,10,3,2,20,5].';
measurement = cameas(state,'spherical')
```

```
measurement = 4×1

   63.4349
         0
    2.2361
   22.3607
```

The elevation of the measurement is zero and the range rate is positive. These results indicate that the object is moving away from the sensor.

Create Measurement from Accelerating Object in Translated Spherical Frame

Define the state of an object moving in 2-D constant-acceleration motion. The state consists of position, velocity, and acceleration in each dimension. The measurements are in spherical coordinates with respect to a frame located at *(20;40;0)* meters from the origin.

```
state = [1,10,3,2,20,5].';
measurement = cameas(state,'spherical',[20;40;0])
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -22.3607
```

The elevation of the measurement is zero and the range rate is negative indicating that the object is moving toward the sensor.

Create Measurement from Constant-Accelerating Object Using Measurement Parameters

Define the state of an object moving in 2-D constant-acceleration motion. The state consists of position, velocity, and acceleration in each dimension. The measurements are in spherical coordinates with respect to a frame located at *(20;40;0)* meters from the origin.

```
state2d = [1,10,3,2,20,5].';
```

The elevation of the measurement is zero and the range rate is negative indicating that the object is moving toward the sensor.

```
frame = 'spherical';
sensorpos = [20;40;0];
sensorvel = [0;5;0];
laxes = eye(3);
measurement = cameas(state2d,'spherical',sensorpos,sensorvel,laxes)
```

```
measurement = 4×1
```

```
 -116.5651
         0
   42.4853
  -17.8885
```

The elevation of the measurement is zero and the range rate is negative. These results indicate that the object is moving toward the sensor.

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

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel,
    'Orientation',laxes);
measurement = cameas(state2d,measparm)
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -17.8885
```

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^2.

Example: `[5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]`

Data Types: `double`

`frame` — Measurement frame
`'rectangular'` (default) | `'spherical'`

Measurement frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of the *x*, *y*, and *z* Cartesian coordinates of the tracked object. When specified as `'spherical'`, a measurement consists of the azimuth, elevation, range, and range rate of the tracked object.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the global coordinate system, 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 global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters/second.

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 global coordinate system.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure

Measurement parameters, specified as a structure. The fields of the structure are:

measurementParameters struct

Parameter	Definition	Default
`OriginPosition`	Sensor position with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters.	`[0;0;0]`
`OriginVelocity`	Sensor velocity with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in m/s.	`[0;0;0]`
`Orientation`	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 global coordinate system.	`eye(3)`
`HasVelocity`	Indicates whether measurements contain velocity or range rate components, specified as `true` or `false`.	`false` when `frame` argument is `'rectangular'` and `true` when `frame` argument is `'spherical'`
`HasElevation`	Indicates whether measurements contain elevation components, specified as `true` or `false`.	`true`

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.

<table>
<tr><th>frame</th><th>measurement</th></tr>
<tr><td>'spherical'</td><td>Specifies the azimuth angle, az, elevation angle, el, range, r, and range rate, rr, of the object with respect to the local ego vehicle coordinate system. Positive values for range rate indicate that an object is moving away from the sensor.

Spherical measurements
<table><tr><th></th><th></th><th colspan="2">HasElevation</th></tr><tr><td></td><td></td><td>false</td><td>true</td></tr><tr><td rowspan="2">HasVelocity</td><td>false</td><td>[az;r]</td><td>[az;el;r]</td></tr><tr><td>true</td><td>[az;r;rr]</td><td>[az;el;r;rr]</td></tr></table>
Angle units are in degrees, range units are in meters, and range rate units are in m/s.</td></tr>
</table>

<table>
<tr><th>frame</th><th>measurement</th></tr>
<tr><td>'rectangular</td><td>Specifies the Cartesian position and velocity coordinates of the tracked object with respect to the ego vehicle coordinate system.

Rectangular measurements
<table><tr><td rowspan="2">HasVelocity</td><td>false</td><td>[x;y;y]</td></tr><tr><td>true</td><td>[x;vx;y,vy;z;vz]</td></tr></table>
Position units are in meters and velocity units are in m/s.</td></tr>
</table>

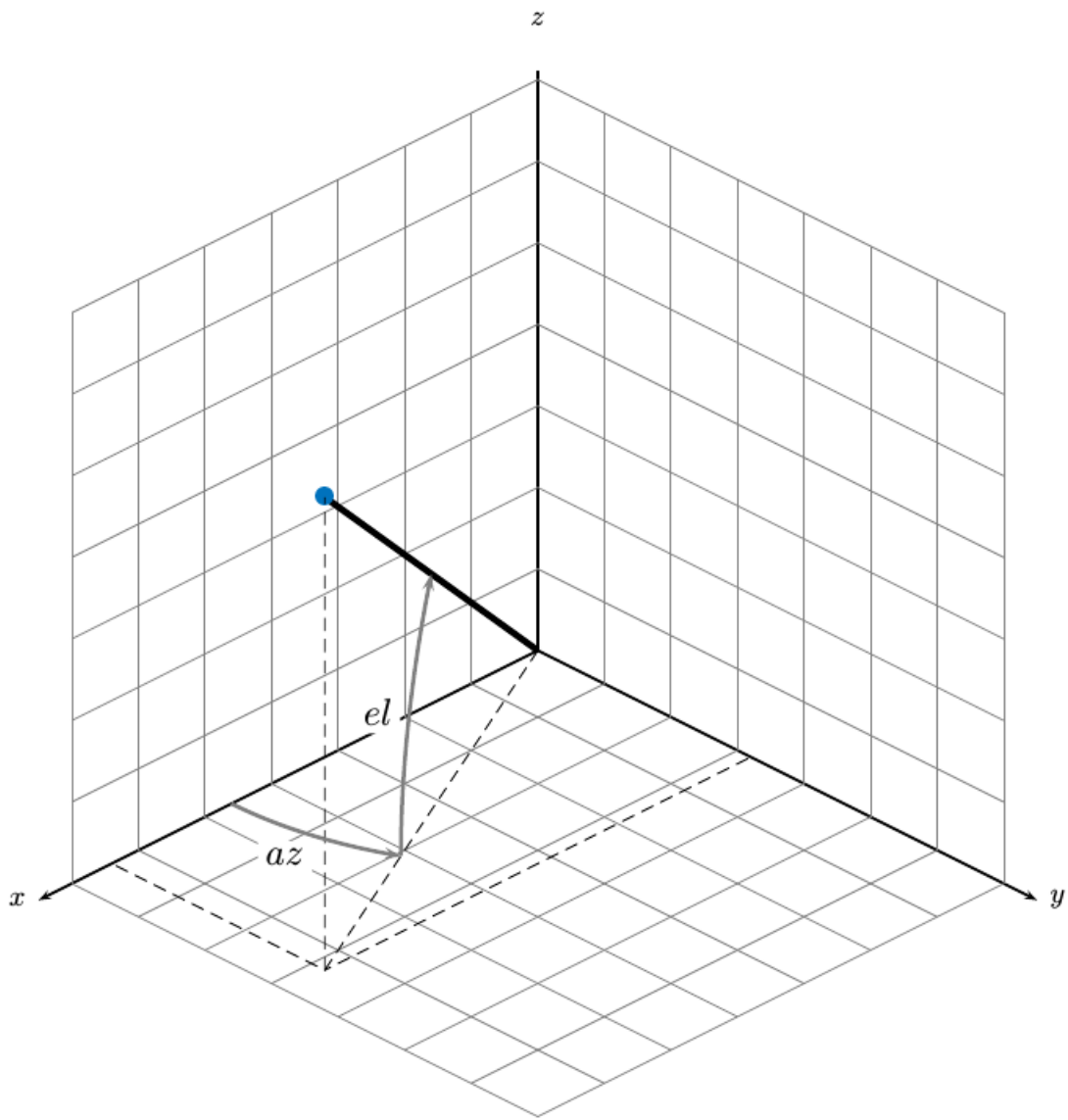
Data Types: `double`

Definitions

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.


z
el
az
x
y

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`

Classes

`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 = 3×6

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

Measurement Jacobian of Accelerating Object in Spherical Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. Compute the measurement Jacobian in spherical coordinates.

```
state = [1;10;3;2;20;5];
measurementjac = cameasjac(state,'spherical')
```

```
measurementjac = 4×6

  -22.9183         0         0   11.4592         0         0
         0         0         0         0         0         0
    0.4472         0         0    0.8944         0         0
    0.0000    0.4472         0    0.0000    0.8944         0
```

Measurement Jacobian of Accelerating Object in Translated Spherical Frame

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. Compute the measurement Jacobian in spherical coordinates with respect to an origin at *(5;-20;0)* meters.

```
state = [1,10,3,2,20,5].';
sensorpos = [5,-20,0].';
measurementjac = cameasjac(state,'spherical',sensorpos)
```

```
measurementjac = 4×6

   -2.5210         0         0   -0.4584         0         0
         0         0         0         0         0         0
   -0.1789         0         0    0.9839         0         0
    0.5903   -0.1789         0    0.1073    0.9839         0
```

Create Measurement Jacobian of Accelerating Object Using Measurement Parameters

Define the state of an object in 2-D constant-acceleration motion. The state is the position, velocity, and acceleration in both dimensions. Compute the measurement Jacobian in spherical coordinates with respect to an origin at *(5;-20;0)* meters.

```
state2d = [1,10,3,2,20,5].';
sensorpos = [5,-20,0].';
frame = 'spherical';
sensorvel = [0;8;0];
laxes = eye(3);
measurementjac = cameasjac(state2d,frame,sensorpos,sensorvel,laxes)
```

```
measurementjac = 4×6

   -2.5210         0         0   -0.4584         0         0
         0         0         0         0         0         0
   -0.1789         0         0    0.9839         0         0
    0.5274   -0.1789         0    0.0959    0.9839         0
```

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

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos,'OriginVelocity',sensorvel,
    'Orientation',laxes);
measurementjac = cameasjac(state2d,measparm)
```

```
measurementjac = 4×6
```

```
   -2.5210         0         0   -0.4584         0         0
         0         0         0         0         0         0
   -0.1789         0         0    0.9839         0         0
    0.5274   -0.1789         0    0.0959    0.9839         0
```

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^2.

Example: `[5;0.1;0.01;0;-0.2;-0.01;-3;0.05;0]`

Data Types: `double`

`frame` — Measurement frame

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

Measurement frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of the *x*, *y*, and *z* Cartesian coordinates of the tracked object. When specified as `'spherical'`, a measurement consists of the azimuth, elevation, range, and range rate of the tracked object.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the global coordinate system, 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 global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters/second.

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 global coordinate system.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure

Measurement parameters, specified as a structure. The fields of the structure are:

measurementParameters struct

Parameter	Definition	Default
`OriginPosition`	Sensor position with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters.	`[0;0;0]`
`OriginVelocity`	Sensor velocity with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in m/s.	`[0;0;0]`
`Orientation`	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 global coordinate system.	`eye(3)`
`HasVelocity`	Indicates whether measurements contain velocity or range rate components, specified as `true` or `false`.	`false` when `frame` argument is `'rectangular'` and `true` when `frame` argument is `'spherical'`
`HasElevation`	Indicates whether measurements contain elevation components, specified as `true` or `false`.	`true`

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.

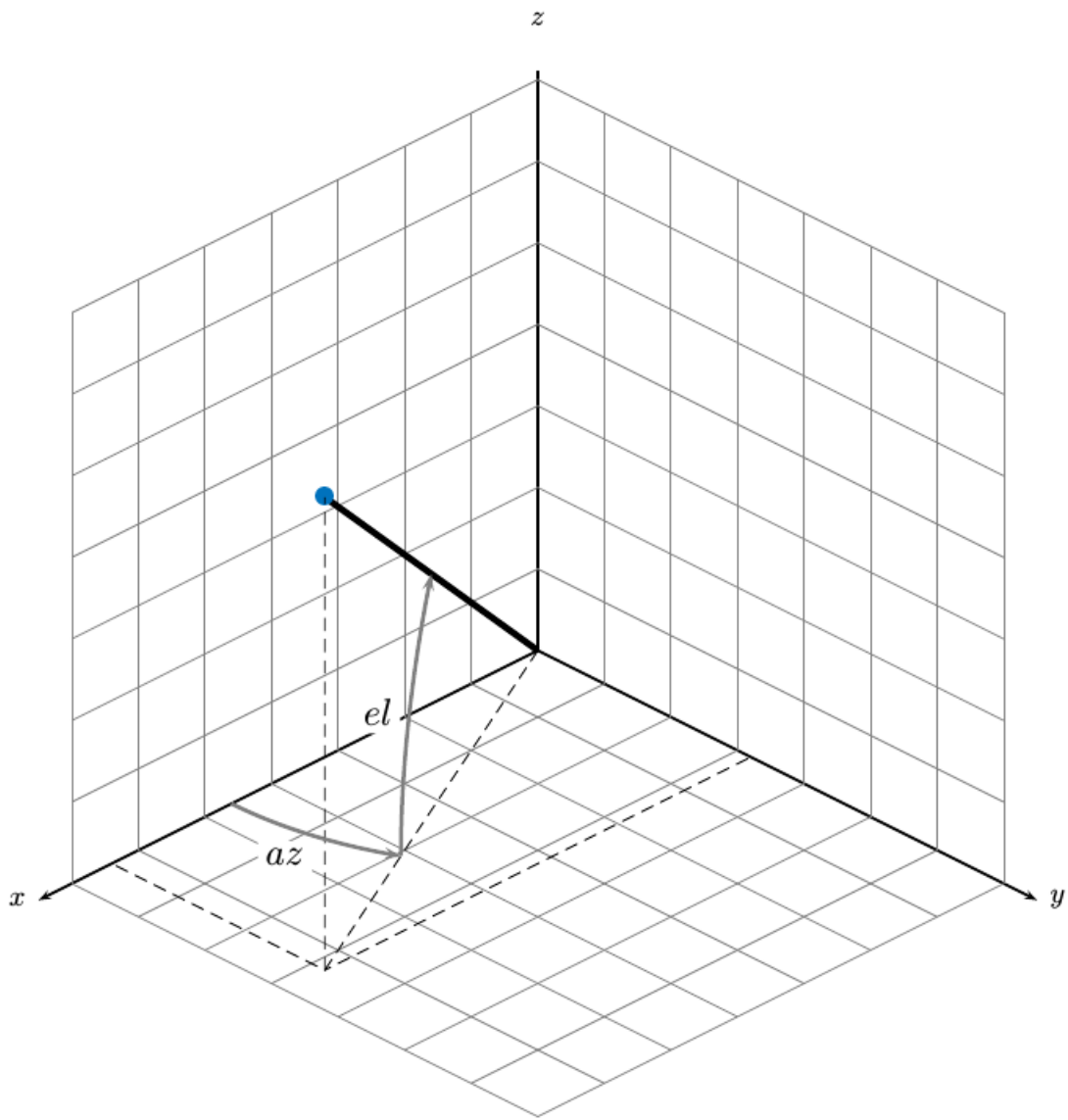
Frame	Measurement Jacobian
`'rectangular'`	Jacobian of the measurements `[x;y;z]` with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
`'spherical'`	Jacobian of the measurement vector `[az;el;r;rr]` with respect to the state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

Definitions

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.


z
el
az
x
y

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`

Classes

`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,dt,w)
```

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,dt,w)` also specifies noise, `w`.

Examples

Update State for Constant Turn-Rate Motion

Define an initial state for 2-D constant turn-rate motion. The turn rate is 12 degrees per second. Update the state to one second later.

```
state = [500,0,0,100,12].';
state = constturn(state)
```

```
state = 5×1

  489.5662
```

```
  -20.7912
   99.2705
   97.8148
   12.0000
```

Update State for Constant Turn-Rate Motion with Specified Time Step

Define an initial state for 2-D constant turn-rate motion. The turn rate is 12 degrees per second. Update the state to 0.1 seconds later.

```
state = [500,0,0,100,12].';
state = constturn(state,0.1)
```

```
state = 5×1

  499.8953
   -2.0942
    9.9993
   99.9781
   12.0000
```

Input Arguments

`state` — State vector

real-valued 5-element vector | real-valued 7-element vector | 5-by-*N* real-valued matrix | 7-by-*N* real-valued matrix

State vector for a constant turn-rate motion model in two or three spatial dimensions, specified as a real-valued vector or matrix.

- When specified as a 5-element vector, the state vector describes 2-D motion in the *x*-*y* plane. You can specify the state vector as a row or column vector. The components of the state vector are `[x;vx;y;vy;omega]` where `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. `y` represents the *y*-coordinate and `vy` represents the velocity in the *y*-direction. `omega` represents the turn rate.

When specified as a 5-by-*N* matrix, each column represents a different state vector *N* represents the number of states.

- When specified as a 7-element vector, the state vector describes 3-D motion. You can specify the state vector as a row or column vector. The components of the state vector are `[x;vx;y;vy;omega;z;vz]` where `x` represents the *x*-coordinate and `vx` represents the velocity in the *x*-direction. `y` represents the *y*-coordinate and `vy` represents the velocity in the *y*-direction. `omega` represents the turn rate. `z` represents the *z*-coordinate and `vz` represents the velocity in the *z*-direction.

  When specified as a 7-by-*N* matrix, each column represents a different state vector. *N* represents the number of states.

Position coordinates are in meters. Velocity coordinates are in meters/second. Turn rate is in degrees/second.

Example: `[5;0.1;4;-0.2;0.01]`

Data Types: `double`

`dt` — Time step interval of filter

`1.0` (default) | positive scalar

Time step interval of filter, specified as a positive scalar. Time units are in seconds.

Example: `0.5`

Data Types: `single` | `double`

`w` — State noise

scalar | real-valued (*D*+1)-by-*N* matrix

State noise, specified as a scalar or real-valued (*D*+1)-length -by-*N* matrix. *D* is the number of motion dimensions and *N* is the number of state vectors. The components are each columns are `[ax;ay;alpha]` for 2-D motion or `[ax;ay;alpha;az]` for 3-D motion. `ax`, `ay`, and `az` are the linear acceleration noise values in the *x*-, *y*-, and *z*-axes, respectively, and `alpha` is the angular acceleration noise value. If specified as a scalar, the value expands to a (*D*+1)-by-*N* matrix.

Data Types: `single` | `double`

Output Arguments

`updatedstate` — Updated state vector
real-valued column or row vector | real-valued matrix

Updated state vector, returned as a real-valued vector or real-valued matrix with same number of elements and dimensions as the input state vector.

Extended Capabilities

C/C++ Code Generation

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

See Also

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

Classes
`trackingEKF` | `trackingUKF`

Introduced in R2018b

constturnjac

Jacobian for constant turn-rate motion

Syntax

```
jacobian = constturnjac(state)
jacobian = constturnjac(state,dt)
[jacobian,noisejacobian] = constturnjac(state,dt,w)
```

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,dt,w)` also specifies noise, `w`, and returns the Jacobian, `noisejacobian`, of the state with respect to the noise.

Examples

Compute State Jacobian for Constant Turn-Rate Motion

Compute the Jacobian for a constant turn-rate motion state. Assume the turn rate is 12 degrees/second. The time step is one second.

```
state = [500,0,0,100,12];
jacobian = constturnjac(state)
```

```
jacobian = 5×5
```

```
    1.0000    0.9927         0   -0.1043   -0.8631
         0    0.9781         0   -0.2079   -1.7072
         0    0.1043    1.0000    0.9927   -0.1213
         0    0.2079         0    0.9781   -0.3629
         0         0         0         0    1.0000
```

Compute State Jacobian for Constant Turn-Rate Motion with Specified Time Step

Compute the Jacobian for a constant turn-rate motion state. Assume the turn rate is 12 degrees/second. The time step is 0.1 second.

```
state = [500,0,0,100,12];
jacobian = constturnjac(state,0.1)
```

```
jacobian = 5×5

    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`

Classes
`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 = 3×1

     1
     2
     0
```

The *z*-component of the measurement is zero.

Create Measurement from Constant Turn-Rate Motion in Spherical Frame

Define the state of an object in 2-D constant turn-rate motion. The state is the position and velocity in each dimension, and the turn rate. The measurements are in spherical coordinates.

```
state = [1;10;2;20;5];
measurement = ctmeas(state,'spherical')
```

```
measurement = 4×1

   63.4349
         0
    2.2361
   22.3607
```

The elevation of the measurement is zero and the range rate is positive indicating that the object is moving away from the sensor.

Create Measurement from Constant Turn-Rate Motion in Translated Spherical Frame

Define the state of an object moving in 2-D constant turn-rate motion. The state consists of position and velocity, and the turn rate. The measurements are in spherical coordinates with respect to a frame located at `[20;40;0]`.

```
state = [1;10;2;20;5];
measurement = ctmeas(state,'spherical',[20;40;0])
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -22.3607
```

The elevation of the measurement is zero and the range rate is negative indicating that the object is moving toward the sensor.

Create Measurement from Constant Turn-Rate Motion using Measurement Parameters

Define the state of an object moving in 2-D constant turn-rate motion. The state consists of position and velocity, and the turn rate. The measurements are in spherical coordinates with respect to a frame located at `[20;40;0]`.

```
state2d = [1;10;2;20;5];
frame = 'spherical';
sensorpos = [20;40;0];
sensorvel = [0;5;0];
laxes = eye(3);
measurement = ctmeas(state2d,frame,sensorpos,sensorvel,laxes)
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -17.8885
```

The elevation of the measurement is zero and the range rate is negative indicating that the object is moving toward the sensor.

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

```
measparm = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes);
measurement = ctmeas(state2d,measparm)
```

```
measurement = 4×1

 -116.5651
         0
   42.4853
  -17.8885
```

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

Measurement frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of the *x*, *y*, and *z* Cartesian coordinates of the tracked object. When specified as `'spherical'`, a measurement consists of the azimuth, elevation, range, and range rate of the tracked object.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the global coordinate system, 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 global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters/second.

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 global coordinate system.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure

Measurement parameters, specified as a structure. The fields of the structure are:

measurementParameters struct

Parameter	Definition	Default
`OriginPosition`	Sensor position with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters.	`[0;0;0]`
`OriginVelocity`	Sensor velocity with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in m/s.	`[0;0;0]`
`Orientation`	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 global coordinate system.	`eye(3)`
`HasVelocity`	Indicates whether measurements contain velocity or range rate components, specified as `true` or `false`.	`false` when `frame` argument is `'rectangular'` and `true` when `frame` argument is `'spherical'`
`HasElevation`	Indicates whether measurements contain elevation components, specified as `true` or `false`.	`true`

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.

<table>
<tr><th>frame</th><th>measurement</th></tr>
<tr><td>'spherical'</td><td>Specifies the azimuth angle, az, elevation angle, el, range, r, and range rate, rr, of the object with respect to the local ego vehicle coordinate system. Positive values for range rate indicate that an object is moving away from the sensor.

Spherical measurements
<table><tr><th></th><th></th><th colspan="2">HasElevation</th></tr><tr><td></td><td></td><td>false</td><td>true</td></tr><tr><td rowspan="2">HasVelocity</td><td>false</td><td>[az;r]</td><td>[az;el;r]</td></tr><tr><td>true</td><td>[az;r;rr]</td><td>[az;el;r;rr]</td></tr></table>
Angle units are in degrees, range units are in meters, and range rate units are in m/s.</td></tr>
</table>

<table>
<tr><th>frame</th><th>measurement</th></tr>
<tr><td>'rectangular</td><td>Specifies the Cartesian position and velocity coordinates of the tracked object with respect to the ego vehicle coordinate system.

Rectangular measurements
<table><tr><td>HasVelocity</td><td>false</td><td>[x;y;y]</td></tr><tr><td></td><td>true</td><td>[x;vx;y,vy;z;vz]</td></tr></table>
Position units are in meters and velocity units are in m/s.</td></tr>
</table>

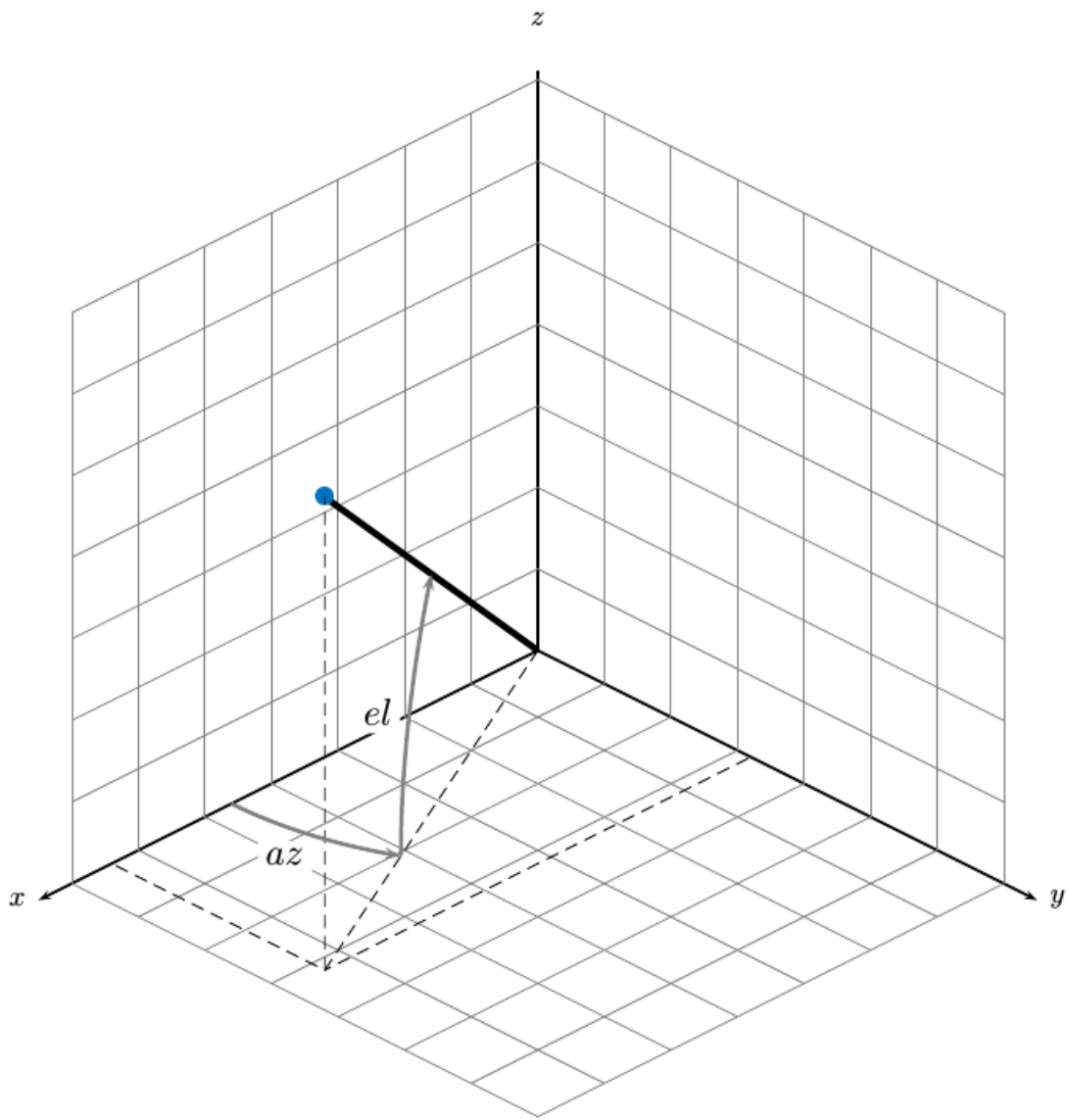
Data Types: `double`

Definitions

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.


z
el
az
x
y

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`

Classes

`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 = 3×5

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

Measurement Jacobian of Constant Turn-Rate Motion in Spherical Frame

Define the state of an object in 2-D constant turn-rate motion. The state is the position and velocity in each dimension, and the turn rate. Compute the measurement Jacobian with respect to spherical coordinates.

```
state = [1;10;2;20;5];
measurementjac = ctmeasjac(state,'spherical')
```

```
measurementjac = 4×5

  -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 = 4×5

   -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 = 4×5

   -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 = 4×5
```

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

Measurement frame, specified as `'rectangular'` or `'spherical'`. When the frame is `'rectangular'`, a measurement consists of the *x*, *y*, and *z* Cartesian coordinates of the tracked object. When specified as `'spherical'`, a measurement consists of the azimuth, elevation, range, and range rate of the tracked object.

Data Types: `char`

`sensorpos` — Sensor position
`[0;0;0]` (default) | real-valued 3-by-1 column vector

Sensor position with respect to the global coordinate system, 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 global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters/second.

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 global coordinate system.

Data Types: `double`

`measurementParameters` — Measurement parameters
structure

Measurement parameters, specified as a structure. The fields of the structure are:

measurementParameters struct

Parameter	Definition	Default
`OriginPosition`	Sensor position with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in meters.	`[0;0;0]`
`OriginVelocity`	Sensor velocity with respect to the global coordinate system, specified as a real-valued 3-by-1 column vector. Units are in m/s.	`[0;0;0]`
`Orientation`	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 global coordinate system.	`eye(3)`
`HasVelocity`	Indicates whether measurements contain velocity or range rate components, specified as `true` or `false`.	`false` when `frame` argument is `'rectangular'` and `true` when `frame` argument is `'spherical'`
`HasElevation`	Indicates whether measurements contain elevation components, specified as `true` or `false`.	`true`

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.

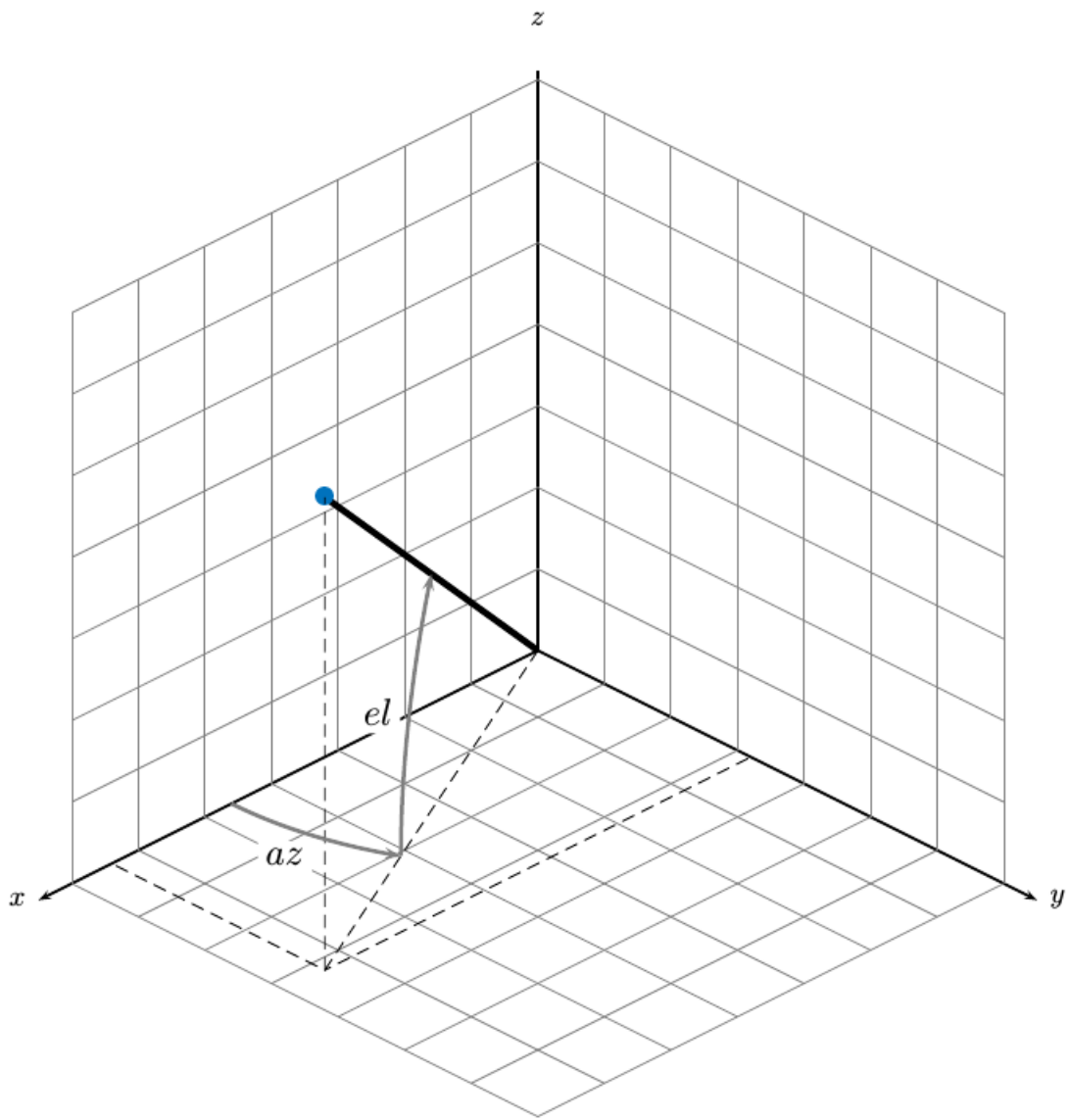
Frame	Measurement Jacobian
`'rectangular'`	Jacobian of the measurements `[x;y;z]` with respect to the state vector. The measurement vector is with respect to the local coordinate system. Coordinates are in meters.
`'spherical'`	Jacobian of the measurement vector `[az;el;r;rr]` with respect to the state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

Definitions

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.


z
el
az
x
y

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

Classes

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 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 = struct with fields:
            TrackID: 1
          BranchID: 1
        UpdateTime: 0
               Age: 1
```

```
              State: [6x1 double]
     StateCovariance: [6x6 double]
          TrackLogic: 'Score'
     TrackLogicState: [13.7102 13.7102]
         IsConfirmed: 1
           IsCoasted: 0
         ObjectClassID: 3
    ObjectAttributes: {}
```

Obtain the position vector and position covariance for that track

```
positionSelector = [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 1 0];
[position,positionCovariance] = getTrackPositions(tracks,positionSelector)
```

```
position = 1×3

   10.0000    3.0000   -7.0000


positionCovariance = 3×3

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

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 = struct with fields:
            TrackID: 1
           BranchID: 1
         UpdateTime: 0
                Age: 1
```

```
            State: [9x1 double]
       StateCovariance: [9x9 double]
            TrackLogic: 'Score'
       TrackLogicState: [13.7102 13.7102]
           IsConfirmed: 1
             IsCoasted: 0
         ObjectClassID: 3
      ObjectAttributes: {}
```

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 = 1×3

   10.0000  -20.0000    4.0000
```

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.

Definitions

Position Selector for 2-Dimensional Motion

Show the position selection matrix for two-dimensional motion when the state consists of the position and velocity.

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

Position Selector for 3-Dimensional Motion

Show the position selection matrix for three-dimensional motion when the state consists of the position and velocity.

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

Position Selector for 3-Dimensional Motion with Acceleration

Show the position selection matrix for three-dimensional motion when the state consists of the position, velocity, and acceleration.

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

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions

`getTrackVelocities` | `initcaekf` | `initcakf` | `initcaukf` | `initctekf` | `initctukf` | `initcvkf` | `initcvukf`

Classes

`objectDetection`

System Objects

`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 0 1 0 0 0 0; 0 0 0 0 0 0 0 1 0];
velocity = getTrackVelocities(tracks,velocitySelector)
```

```
velocity = 1×3

    1.0093   -0.6728         0
```

Velocity and Covariance of 3-D Constant-Acceleration Object

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

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

```
velocity = 1×3

    1.0093   -0.6728    1.0093


velocityCovariance = 3×3

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

Input Arguments

`tracks` — 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.

Definitions

Velocity Selector for 2-Dimensional Motion

Show the velocity selection matrix for two-dimensional motion when the state consists of the position and velocity.

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

Velocity Selector for 3-Dimensional Motion

Show the velocity selection matrix for three-dimensional motion when the state consists of the position and velocity.

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

Velocity Selector for 3-Dimensional Motion with Acceleration

Show the velocity selection matrix for three-dimensional motion when the state consists of the position, velocity, and acceleration.

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

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`

Classes
`objectDetection`

System Objects
`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
ABF.MeasurementNoise
```

```
ans =
```

```
     1
     0
     0
     3
     0
     0


ans =

    1.0000    0.2000
    0.2000    2.0000
```

Input Arguments

`detection` — Detection report
`objectDetection` class object

Detection report, specified as an "Object Detections" 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 `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`

Classes

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

System Objects

`trackerGNN` | `trackerTOMHT`

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
ABF.MeasurementNoise
```

```
ans =
```

```
     1
     0
     3
     0


ans =

    1.0000    0.2000
    0.2000    2.0000
```

Input Arguments

`detection` — Detection report
`objectDetection` class object

Detection report, specified as an “Object Detections” 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 `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` | `initcaabf` | `initcackf` | `initcaekf` | `initcakf` | `initcaukf` | `initctckf` | `initctekf` | `initctukf` | `initcvckf` | `initcvekf` | `initcvkf` | `initcvukf`

Classes

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

System Objects

`trackerGNN` | `trackerTOMHT`

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` class 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 = 9×1

     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 = 3×3

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

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, el = 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` class object

Detection report, specified as an “Object Detections” 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`

Classes

`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` class 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: [9×1 double]
          StateCovariance: [9×9 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 = 9×1

    0.9857
   -0.2361
   -0.0030
    3.0097
    0.4079
    0.0214
    0.0460
   -0.2209
   -0.5646
```

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

```
pf.MeasurementNoise
```

```
ans = 3×3

    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, el = 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: [9×1 double]
            StateCovariance: [9×9 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 = 4×1

   29.9188
    5.0976
   99.8303
    4.0255
```

Input Arguments

`detection` — Detection report
`objectDetection` class object

Detection report, specified as an “Object Detections” 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

Classes
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` class 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 = 6×1

     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 = 3×3

    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, el = 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` class object

Detection report, specified as an “Object Detections” 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`

Classes

`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` class 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: [6x1000 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 = 6×1

    1.0208
    0.2489
    2.9758
    0.1525
    0.0186
   -0.1791
```

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

```
pf.MeasurementNoise
```

```
ans = 3×3

    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, el = 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: [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: [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 = 4×1

   29.9188
    5.0976
   99.8303
    4.0255
```

Input Arguments

`detection` — Detection report

`objectDetection` class object

Detection report, specified as an “Object Detections” 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`

Classes
`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` class 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 = 7×1

     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 = 3×3

    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, el = 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` class object

Detection report, specified as an “Object Detections” 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`

Classes

`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` class 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: [4x4 double]
 HasAdditiveProcessNoise: 0

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

                Particles: [7x1000 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 = 7×1

    1.0043
    0.5556
    3.0166
   -0.1605
   -0.0213
   -0.0163
    0.0866
```

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

```
pf.MeasurementNoise
```

```
ans = 3×3

    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, el = 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: [7×1 double]
            StateCovariance: [7×7 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` class object

Detection report, specified as an “Object Detections” 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`

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

System Objects
`trackerGNN` | `trackerTOMHT`

Introduced in R2018b

initcaggiwphd

Create constant acceleration ggiwphd filter from detection report

Syntax

```
phd = initggiwphd(detections)
```

Description

`phd = initggiwphd(detections)` initializes a constant acceleration `ggiwphd` filter based on information provided in object detections, `detections`.

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
measurementMean = mean(measurements,2)
```

```
states =

    1.2856
         0
         0
    1.9950
         0
         0
    2.9779
         0
         0


measurementMean =

    1.2856
    1.9950
    2.9779
```

Check the extent and expected number of detections.

```
extent = phd.ScaleMatrices/(phd.DegreesOfFreedom - 4)
expDetections = phd.Shapes/phd.Rates
```

```
extent =

    1.4603    0.0885   -0.2403
    0.0885    3.0050   -0.0225
   -0.2403   -0.0225    4.8365


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 from detection report

Syntax

```
phd = initggiwphd(detections)
```

Description

`phd = initggiwphd(detections)` initializes a constant turn-rate `ggiwphd` filter based on information provided in object detections, `detections`.

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
measurementMean = mean(measurements,2)
```

```
states =

    1.2856
         0
    1.9950
         0
         0
    2.9779
         0


measurementMean =

    1.2856
    1.9950
    2.9779
```

Check the extent and expected number of detections.

```
extent = phd.ScaleMatrices/(phd.DegreesOfFreedom - 4)
expDetections = phd.Shapes/phd.Rates
```

```
extent =

    1.4603    0.0885   -0.2403
    0.0885    3.0050   -0.0225
   -0.2403   -0.0225    4.8365


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

initcvggiwphd

Create constant velocity `ggiwphd` filter from detection report

Syntax

```
phd = initggiwphd(detections)
```

Description

`phd = initggiwphd(detections)` initializes a constant velocity `ggiwphd` filter based on information provided in object detections, `detections`.

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 = initcvggiwphd(detections);
```

Check the values of state in the filter has the same position estimates as the mean of measurements.

```
states = phd.States
measurementMean = mean(measurements,2)
```

```
states =

    1.2856
         0
    1.9950
         0
    2.9779
         0


measurementMean =

    1.2856
    1.9950
    2.9779
```

Check the extent and expected number of detections.

```
extent = phd.ScaleMatrices/(phd.DegreesOfFreedom - 4)
expDetections = phd.Shapes/phd.Rates
```

```
extent =

    1.4603    0.0885   -0.2403
    0.0885    3.0050   -0.0225
   -0.2403   -0.0225    4.8365


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 `initcvggiwphd` 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

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;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 = 7×1

     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:

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

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;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:

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

Example: [0;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 dimnension 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

```
mscekf = initcvmscekf(detection)
mscekf = initcvmscekf(detection,rangeEstimation)
```

Description

`mscekf = initcvmscekf(detection)` initializes a `trackingMSCEKF` class (extended Kalman filter for tracking in modified spherical coordinates) based on information provided in an `objectDetection` class object, `detection`. The function assumes a target range of $3e^4$ units and a range-covariance of $1e^{10}$ units^2.

The `trackingMSCEKF` object can be used with trackers for tracking targets with angle-only measurements from a single observer.

`mscekf = 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: [6×1 double]
                StateCovariance: [6×6 double]

             StateTransitionFcn: @constvelmsc
     StateTransitionJacobianFcn: @constvelmscjac
                  ProcessNoise: [3×3 double]
        HasAdditiveProcessNoise: 0
                  ObserverInput: [3×1 double]

                MeasurementFcn: @cvmeasmsc
         MeasurementJacobianFcn: @cvmeasmscjac
               MeasurementNoise: [2×2 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: [3×3 double]


measParamPlatToScenario = struct('Frame','Rectangular','HasRange',false,...
'Orientation',rotmat(quaternion([30 0 0],'rotvecd'),'frame'))

measParamPlatToScenario = struct with fields:
          Frame: 'Rectangular'
       HasRange: 0
    Orientation: [3×3 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 = 2×1

   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.

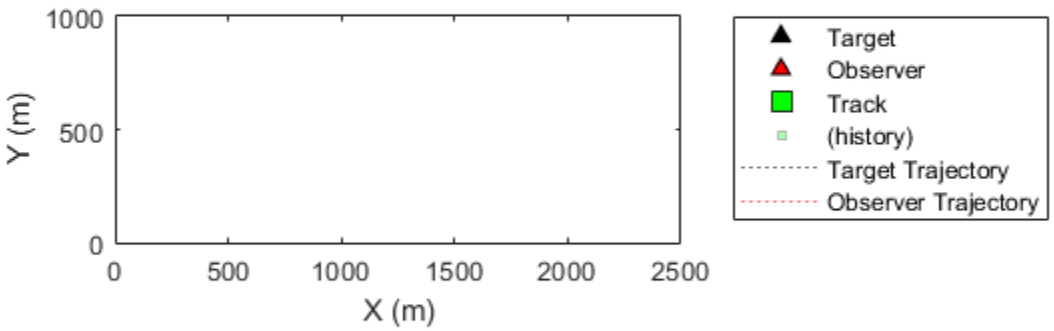
```
syntheticParams = struct('Frame','Spherical','HasRange',false,...
    'OriginPosition',observerState(1:3:end));
meas = cvmeas(tgtState,syntheticParams);
```

Create an angle-only objectDetection to simulate synthetic detection.

```
detection = objectDetection(0,meas,'MeasurementParameters',...
    struct('Frame','Spherical','HasRange',false),'MeasurementNoise',0.033*eye(2));
```

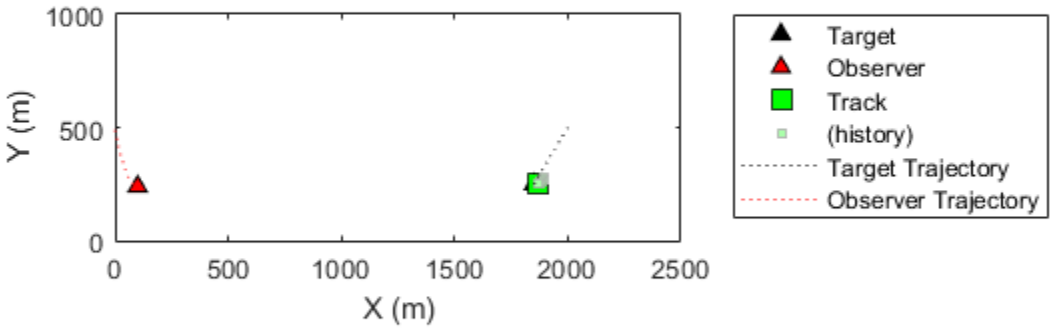
Create trackPlotter and platformPlotter to visualize the scenario.

```
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
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',observerS
    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 = cvmeasmsc(allTracks(i).State,'rectangular') + observerState(1:3:end);
    trkPlotter.plotTrack(cartState');
end
```



Input Arguments

`detection` — Detection report
`objectDetection` class object

Detection report, specified as an “Object Detections” 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` class 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 `trackGNN` 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`

Classes
`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,'HasVelo
```

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',measPa
```

```
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.

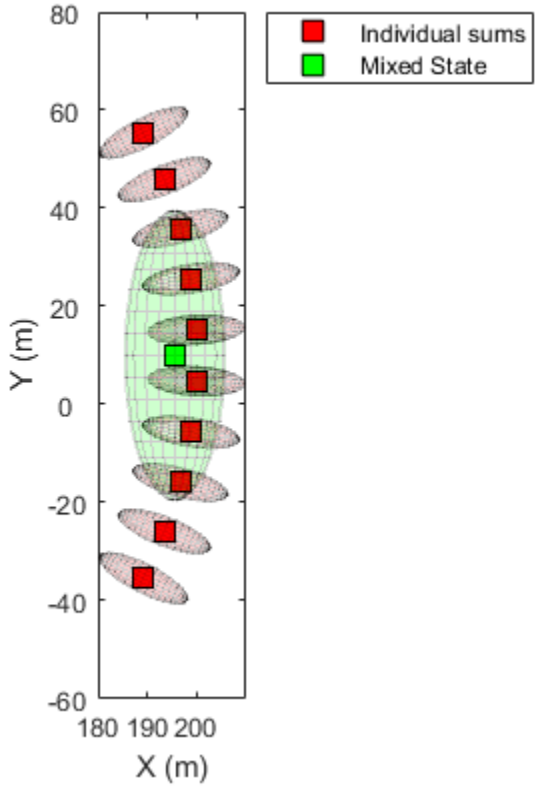
```
funcHandle = @(detection)initapekf(detection,numFilters,angleLimits)
```

```
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,'HasVeloc
```

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.

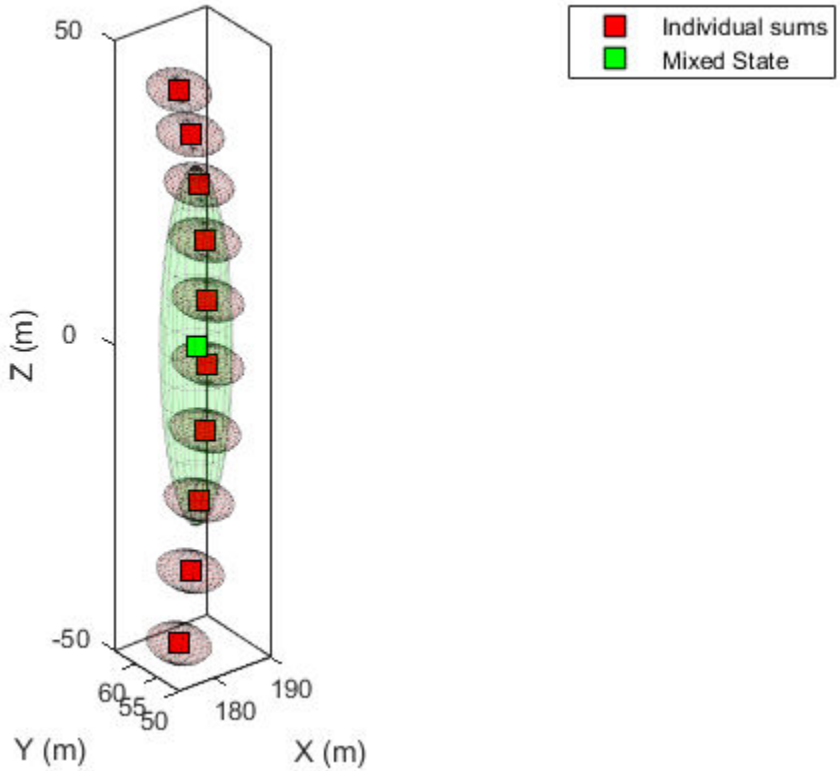
```
det = objectDetection(0,[30;100],'MeasurementParameters',measParam,'MeasurementNoise',1
```

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` class object

Detection report, specified as an “Object Detections” 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.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions
`initcvekf`

Classes
`objectDetection` | `trackingEKF`

Objects
`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,'MeasurementNo
```

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)
```

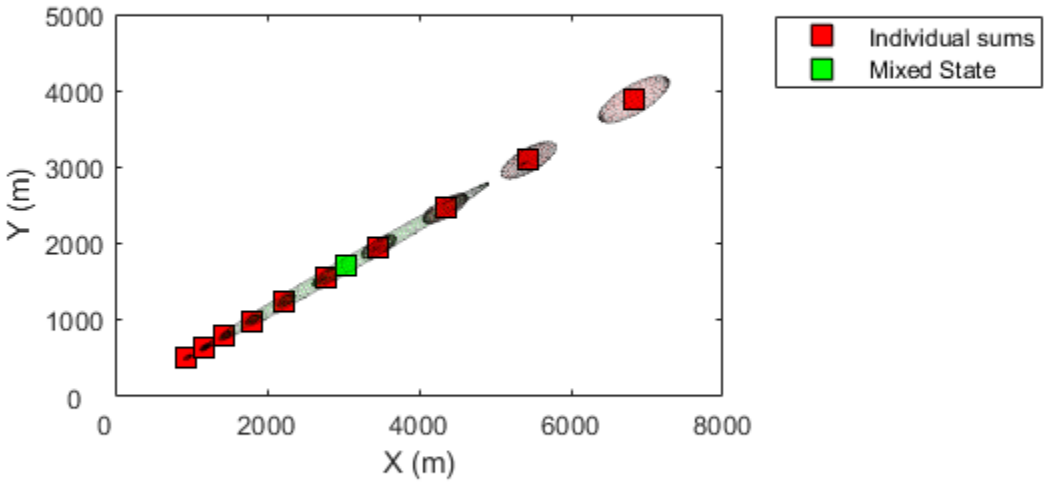
```
tracker = trackerGNN('FilterInitializationFcn',funcHandle)
```

```
tracker =
  trackerGNN with properties:

       FilterInitializationFcn: [function_handle]
                    Assignment: 'Munkres'
           AssignmentThreshold: [30 Inf]
                  MaxNumTracks: 100
                MaxNumSensors: 20

                    TrackLogic: 'History'
          ConfirmationThreshold: [2 3]
              DeletionThreshold: [5 5]

             HasCostMatrixInput: false
    HasDetectableTrackIDsInput: false

                     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` class object

Detection report, specified as an “Object Detections” 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.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions
`initapekf` | `initcvekf` | `initcvmscekf`

Classes
`objectDetection` | `trackingEKF`

Objects
`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` class 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 = 6×1
```

```
     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 = 3×3

    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: `az = 40`, `el = 6`, `r = 100`, `rr = 5`. Measurement noise is `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` class object

Detection report, specified as an “Object Detections” 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

Classes
`objectDetection`

Objects
`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 = 9×1

  -200
     0
     0
   -30
     0
     0
     0
     0
     0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 9×9

    2.1000         0         0         0         0         0         0         0
         0  100.0000         0         0         0         0         0         0
         0         0  100.0000         0         0         0         0         0
         0         0         0    2.1000         0         0         0         0
         0         0         0         0  100.0000         0         0         0
         0         0         0         0         0  100.0000         0         0
         0         0         0         0         0         0    2.1000         0
         0         0         0         0         0         0         0  100.0000
         0         0         0         0         0         0         0         0  100.0
```

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^3.
- 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`

Classes

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

System Objects

`trackerGNN` | `trackerTOMHT`

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 = 6×1
```

```
    10
     0
     0
    -5
     0
     0
```

Show the state transition model.

```
filter.StateTransitionModel
```

```
ans = 6×6

    1.0000    1.0000    0.5000         0         0         0
         0    1.0000    1.0000         0         0         0
         0         0    1.0000         0         0         0
         0         0         0    1.0000    1.0000    0.5000
         0         0         0         0    1.0000    1.0000
         0         0         0         0         0    1.0000
```

Input Arguments

`detection` — Detection report
`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

`filter` — Linear Kalman filter
`trackingKF` object

Linear Kalman filter, returned as a `trackingKF` object.

Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration rate standard deviation of 1 m/s^3.
- 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`

Classes

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

System Objects

`trackerGNN` | `trackerTOMHT`

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 = 9×1

  -200
     0
     0
   -30
     0
     0
     5
     0
     0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 9×9

     2     0     0     0     0     0     0     0     0
     0   100     0     0     0     0     0     0     0
     0     0   100     0     0     0     0     0     0
     0     0     0     2     0     0     0     0     0
     0     0     0     0   100     0     0     0     0
     0     0     0     0     0   100     0     0     0
     0     0     0     0     0     0     2     0     0
     0     0     0     0     0     0     0   100     0
```

```
     0     0     0     0     0     0     0     0   100
```

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^3.
- 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`

Classes

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

System Objects

`trackerGNN` | `trackerTOMHT`

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 = 7×1

  -250
     0
   -40
     0
     0
     0
     0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 7×7

     2     0     0     0     0     0     0
     0   100     0     0     0     0     0
     0     0     2     0     0     0     0
     0     0     0   100     0     0     0
     0     0     0     0   100     0     0
     0     0     0     0     0     2     0
     0     0     0     0     0     0   100
```

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^2, and a turn-rate acceleration standard deviation of $1°/s^2$.
- 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`

Classes

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

System Objects

`trackerGNN` | `trackerTOMHT`

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 = 7×1

  -250
     0
   -40
     0
     0
     0
     0
```

Show the state covariance matrix.

```
filter.StateCovariance
```

```
ans = 7×7

     2     0     0     0     0     0     0
     0   100     0     0     0     0     0
     0     0     2     0     0     0     0
     0     0     0   100     0     0     0
     0     0     0     0   100     0     0
     0     0     0     0     0     2     0
     0     0     0     0     0     0   100
```

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^2, and a turn-rate acceleration standard deviation of $1°/s^2$.
- 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`

Classes

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

System Objects

`trackerGNN` | `trackerTOMHT`

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 = 6×6

    1.5000         0         0         0         0         0
         0  100.0000         0         0         0         0
         0         0    1.5000         0         0         0
         0         0         0  100.0000         0         0
         0         0         0         0    1.5000         0
         0         0         0         0         0  100.0000
```

Create 3-D Constant Velocity EKF from Spherical Measurement

Initialize a 3-D constant-velocity extended Kalman filter from an initial detection report made from a 3-D measurement in spherical coordinates. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to 45 degrees, the elevation to -10 degrees, the range to 1000 meters, and the range rate to -4.0 m/s.

```
frame = 'spherical';
sensorpos = [25,-40,0].';
sensorvel = [0;5;0];
laxes = eye(3);
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',true, ...
    'HasElevation',true);
meas = [45;-10;1000;-4];
measnoise = diag([3.0,2.5,2,1.0].^2);
detection = objectDetection(0,meas,'MeasurementNoise', ...
    measnoise,'MeasurementParameters',measparms)
```

```
detection =
  objectDetection with properties:

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

```
filter = initcvekf(detection);
```

Filter state vector.

```
disp(filter.State)
```

```
  721.3642
   -2.7855
  656.3642
    2.2145
 -173.6482
    0.6946
```

Input Arguments

`detection` — Detection report
`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

`filter` — Extended Kalman filter
`trackingEKF` object

Extended Kalman filter, returned as a `trackingEKF` object.

Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration standard deviation of 1 m/s^2.
- 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`

Classes
`objectDetection` | `trackingEKF` | `trackingKF` | `trackingUKF`

System Objects
`trackerGNN` | `trackerTOMHT`

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 = 4×1

    10
     0
    20
     0
```

Show the state transition model.

```
filter.StateTransitionModel
```

```
ans = 4×4

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

Initialize 3-D Constant-Velocity Linear Kalman Filter

Create and initialize a 3-D linear Kalman filter object from an initial detection report.

Create the detection report from an initial 3-D measurement, (10,20,−5), of the object position.

```
detection = objectDetection(0,[10;20;-5],'MeasurementNoise',eye(3), ...
    'SensorIndex', 1,'ObjectClassID',1,'ObjectAttributes',{'Green Car', 5});
```

Create the new filter from the detection report and display its properties.

```
filter = initcvkf(detection)
```

```
filter =
  trackingKF with properties:

               State: [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 = 6×1

    10
     0
    20
     0
    -5
     0
```

Show the state transition model.

```
filter.StateTransitionModel
```

```
ans = 6×6

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

Input Arguments

`detection` — Detection report
`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

`filter` — Linear Kalman filter
`trackingKF` object

Linear Kalman filter, returned as a `trackingKF` object.

Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration standard deviation of 1 m/s^2.
- 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`

Classes
`trackingEKF` | `trackingKF` | `trackingUKF`

System Objects
`trackerGNN` | `trackerTOMHT`

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 = 6×1

    10
     0
   200
     0
    -5
     0
```

Show the state covariance.

```
filter.StateCovariance
```

```
ans = 6×6

    1.5000         0         0         0         0         0
         0  100.0000         0         0         0         0
         0         0    1.5000         0         0         0
         0         0         0  100.0000         0         0
         0         0         0         0    1.5000         0
         0         0         0         0         0  100.0000
```

Create Constant Velocity UKF from Spherical Measurement

Initialize a constant-velocity unscented Kalman filter from an initial detection report made from an initial measurement in spherical coordinates. Because the object lies in the *x*-*y* plane, no elevation measurement is made. If you want to use spherical coordinates, then you must supply a measurement parameter structure as part of the detection report with the `Frame` field set to `'spherical'`. Set the azimuth angle of the target to 45 degrees, the range to 1000 meters, and the range rate to -4.0 m/s.

```
frame = 'spherical';
sensorpos = [25,-40,0].';
sensorvel = [0;5;0];
laxes = eye(3);
```

Create the measurement parameters structure. Set `'HasElevation'` to `false`. Then, the measurement consists of azimuth, range, and range rate.

```
measparms = struct('Frame',frame,'OriginPosition',sensorpos, ...
    'OriginVelocity',sensorvel,'Orientation',laxes,'HasVelocity',true, ...
    'HasElevation',false);
meas = [45;1000;-4];
measnoise = diag([3.0,2,1.0].^2);
detection = objectDetection(0,meas,'MeasurementNoise', ...
    measnoise,'MeasurementParameters',measparms)
```

```
detection =
  objectDetection with properties:

                    Time: 0
             Measurement: [3x1 double]
         MeasurementNoise: [3x3 double]
              SensorIndex: 1
            ObjectClassID: 0
    MeasurementParameters: [1x1 struct]
          ObjectAttributes: {}
```

```
filter = initcvukf(detection);
```

Display filter state vector.

```
disp(filter.State)
```

```
  732.1068
   -2.8284
```

```
667.1068
  2.1716
       0
       0
```

Input Arguments

`detection` — Detection report
`objectDetection` object

Detection report, specified as an `objectDetection` object.

Example: `detection = objectDetection(0,[1;4.5;3],'MeasurementNoise', [1.0 0 0; 0 2.0 0; 0 0 1.5])`

Output Arguments

`filter` — Unscented Kalman filter
`trackingUKF` object

Unscented Kalman filter, returned as a `trackingUKF` object.

Algorithms

- The function computes the process noise matrix assuming a one-second time step and an acceleration standard deviation of 1 m/s^2.
- 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

Classes
objectDetection | trackingEKF | trackingKF | trackingUKF

System Objects
trackerGNN | trackerTOMHT

Introduced in R2018b

clone

Copy filter for object tracking

Syntax

```
copiedObj = clone(filterObj)
```

Description

`copiedObj = clone(filterObj)` creates a copy of the given filter object with the same property values.

Input Arguments

`filterObj` — Filter for object tracking
object

Filter for object tracking, specified as an object.

Output Arguments

`copiedObj` — Copy of filter
object

Copy of filter, returned as an object.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions
`constvel` | `correct` | `cvmeas` | `distance` | `likelihood` | `predict` | `residual`

Objects
`trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

Introduced in R2017a

correct

Correct state and state estimation error covariance

Syntax

```
[xCorr,pCorr] = correct(filterObj,zMeas)
[xCorr,pCorr] = correct(filterObj,zMeas,vargin)

xCorr = correct(filterObj,zMeas)
[xCorr,pCorr,zCorr] = correct(filterObj,zMeas)
```

Description

`[xCorr,pCorr] = correct(filterObj,zMeas)` corrects the state and measurement of the tracking filter object based on the current measurement, `zMeas`. The internal state and covariance of the filter are overwritten by the corrected values.

`[xCorr,pCorr] = correct(filterObj,zMeas,vargin)` specifies additional parameters used by the state transition function defined in the `MeasurementFcn` property of the tracking object.

`xCorr = correct(filterObj,zMeas)` corrects the state of the tracking filter object based on the current measurement, `zMeas`. This syntax is only supported for the `trackingABF` object.

`[xCorr,pCorr,zCorr] = correct(filterObj,zMeas)` also returns the corrected measurement. This syntax is only supported for the `trackingABF` object.

Input Arguments

`filterObj` — Filter for object tracking

object

Filter for object tracking, specified as an object. Calling the `predict` overwrites the internal states of the object.

`zMeas` — Measurement of filter
vector | matrix

Measurement of the tracked object, specified as a vector or matrix.

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 `filterObj` 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 `filterObj` is overwritten with this value.

`zCorr` — Corrected measurement of filter
vector | matrix

Corrected measurement of the filter, specified as a vector or matrix. This output is only supported for the `trackingABF` object.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions
`clone` | `constvel` | `cvmeas` | `distance` | `likelihood` | `predict` | `residual`

Objects
`trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

Introduced in R2017a

distance

Distances between measurements and predicted measurements

Syntax

```
dist = distance(filterObj,zMatrix)
dist = distance(filterObj,zMatrix,measurementParams)
```

Description

`dist = distance(filterObj,zMatrix)` computes the distance between one or more predicted measurements given in `zMatrix` and the current measurement in the `filterObj`.

`dist = distance(filterObj,zMatrix,measurementParams)` specifies additional parameters that are used by the `MeasurementFcn` of the `filterObj`.

Input Arguments

`filterObj` — Filter for object tracking
object

Filter for object tracking, specified as an object.

`zMatrix` — Measurements of tracked objects
matrix

Measurements of tracked objects, specified as a matrix. Each row of the matrix contains a measurement vector.

`measurementParams` — Parameters for measurement function
cell array

Parameters for measurement function, specified as a cell array. The parameters are passed to the measurement function defined in the `MeasurementFcn` property of the `filterObj`

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 `filterObj` and a row of the `zMatrix`.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions
`clone` | `constvel` | `correct` | `cvmeas` | `likelihood` | `predict` | `residual`

Objects
`trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

Introduced in R2017a

initialize

Initialize state and covariance of filter

Syntax

```
initialize(filterObj,state,stateCovariance)
initialize(filterObj,state,stateCovariance,Name,Value)
```

Description

`initialize(filterObj,state,stateCovariance)` initializes the filter by setting the `State` and `StateCovariance` properties of the `filterObj` with the given values.

`initialize(filterObj,state,stateCovariance,Name,Value)` also sets additional property names and their values as `Name,Value` pairs in the `filterObj`.

Input Arguments

`filterObj` — Filter for object tracking

object

Filter for object tracking, specified as an object.

`state` — Filter state

real-valued *M*-element vector

Filter state, specified as a real-valued *M*-element vector.

Example: `[200;0.2]`

Data Types: `double`

`stateCovariance` — 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 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]`

See Also

Functions
`clone` | `constvel` | `correct` | `cvmeas` | `distance` | `predict` | `residual`

Objects
`trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

Introduced in R2018b

predict

Predict state and state estimation error covariance

Syntax

```
[xPred,pPred] = predict(filterObj)
[xPred,pPred] = predict(filterObj,tStep)
[xPred,pPred] = predict(filterObj,vargin)

xPred = predict(filterObj)
[xPred,pPred,zPred] = predict(filterObj,tStep)
```

Description

`[xPred,pPred] = predict(filterObj)` returns the predicted state and state estimation error covariance at the next time step.

`[xPred,pPred] = predict(filterObj,tStep)` specifies the time step as a positive scalar in seconds.

`[xPred,pPred] = predict(filterObj,vargin)` specifies additional parameters used by the state transition function defined in the `StateTransitionFcn` property of the tracking object.

`xPred = predict(filterObj)` returns the predicted state at the next time step. The internal state of the input filter object is overwritten by the prediction results. This syntax is only supported for the `trackingABF` object.

`[xPred,pPred,zPred] = predict(filterObj,tStep)` also returns the predicted measurement, `zPred`. This syntax is only supported for the `trackingABF` object.

Input Arguments

`filterObj` — Filter for object tracking
object

Filter for object tracking, specified as an object. Calling the `predict` overwrites the internal states of the object.

`tStep` — Time step
positive scalar

Time step for next prediction, specified as a positive scalar in seconds.

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 `filterObj` 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 `filterObj` is overwritten with this value.

`zPred` — Predicted measurement
vector | matrix

Predicted measurement, specified as a vector or matrix. This output is only supported for the `trackingABF` object.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions
`clone` | `constvel` | `correct` | `cvmeas` | `distance` | `likelihood` | `residual`

Objects
`trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

Introduced in R2017a

residual

Measurement residual and residual noise

Syntax

```
[zRes,resCovariance] = residual(filterObj,zMeas)
[zRes,resCovariance] = residual(filterObj,zMeas,measurementParams)
```

Description

`[zRes,resCovariance] = residual(filterObj,zMeas)` computes the residual and residual covariance of the current given measurement, `zMeas`, with the predicted measurement in the filter, `filterObj`.

`[zRes,resCovariance] = residual(filterObj,zMeas,measurementParams)` specifies additional parameters that are used by the `MeasurementFcn` of the `filterObj` filter.

Input Arguments

`filterObj` — Filter for object tracking
object

Filter for object tracking, specified as an object. Calling the `predict` overwrites the internal states of the object.

`zMeas` — Measurement of filter
vector | matrix

Measurement of the tracked object, specified as a vector or matrix.

`measurementParams` — Parameters for measurement function
cell array

Parameters for measurement function, specified as a cell array. The parameters are passed to the measurement function defined in the `MeasurementFcn` property of the `filterObj`

Output Arguments

`zRes` — Residual between current and predicted measurement
matrix

Residual between current and predicted measurement, returned as a matrix.

`resCovariance` — Residual covariance
matrix

Residual covariance, returned as a matrix.

Extended Capabilities

C/C++ Code Generation

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

See Also

Functions
`clone` | `constvel` | `correct` | `cvmeas` | `distance` | `likelihood` | `predict`

Objects
`trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

Introduced in R2017a

likelihood

Likelihood of measurement

Syntax

```
measLikelihood = likelihood(filterObj,zMeas)
[measLikelihood = likelihood(filterObj,zMeas,measurementParams)
```

Description

`measLikelihood = likelihood(filterObj,zMeas)` returns the likelihood of a measurement given by the specified filter, `filterObj`.

`[measLikelihood = likelihood(filterObj,zMeas,measurementParams)` specifies additional parameters that are used by the `MeasurementFcn` of the `filterObj`.

Input Arguments

`filterObj` — Filter for object tracking
object

Filter for object tracking, specified as an object. Calling the `predict`overwrites the internal states of the object.

`zMeas` — Measurement of the filter
vector | matrix

Measurement of the tracked object, specified a vector, or matrix.

`measurementParams` — Parameters for measurement function
cell array

Parameters for measurement function, specified as a cell array. The parameters are passed to the measurement function defined in the `MeasurementFcn` of the `filterObj`

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

Functions
`clone` | `constvel` | `correct` | `cvmeas` | `distance` | `predict` | `residual`

Objects
`trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

Introduced in R2018b

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)
```

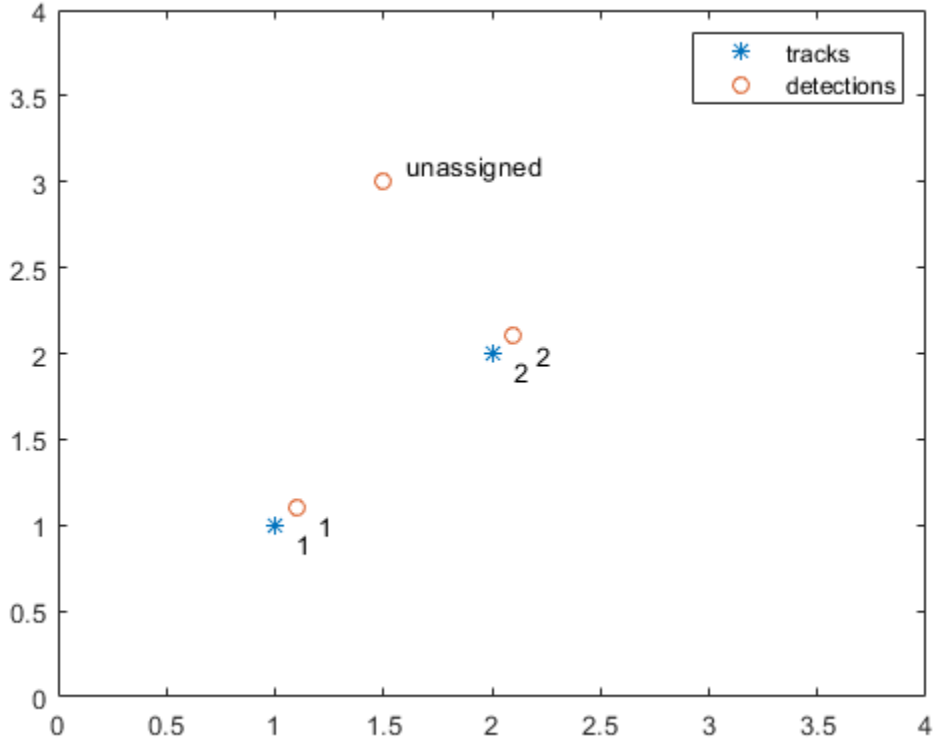
Display the unassigned detections.

```
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` | `assignsd` | `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)
```

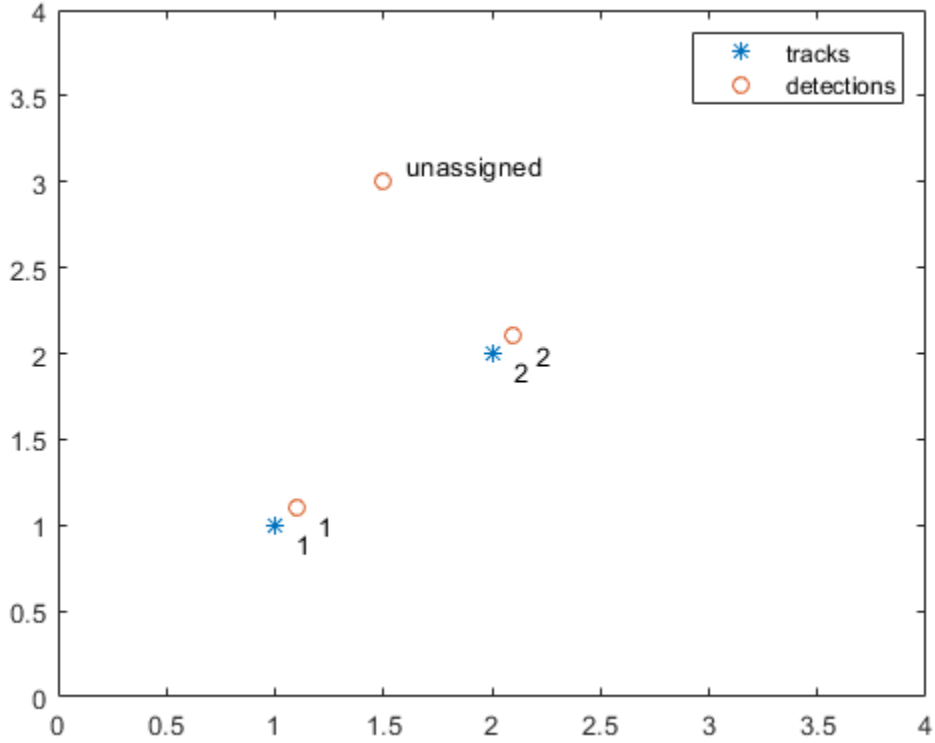
Display the unassigned detections.

```
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 Assignkbest

Create a cost matrix containing prohibited assignments. Then, use the `assignkbest` 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] = ...
    assignkbest(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 =

   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^{th} 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^{th} 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^{th} 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] 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` | `assignsd` | `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 =

  -34.7000
  -31.7000
  -29.1000
  -28.6000
  -28.0000


solutionGap =

         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, ...)` 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 $(Q\text{-}1)^{th}$ 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` | `assignsd`

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)
```

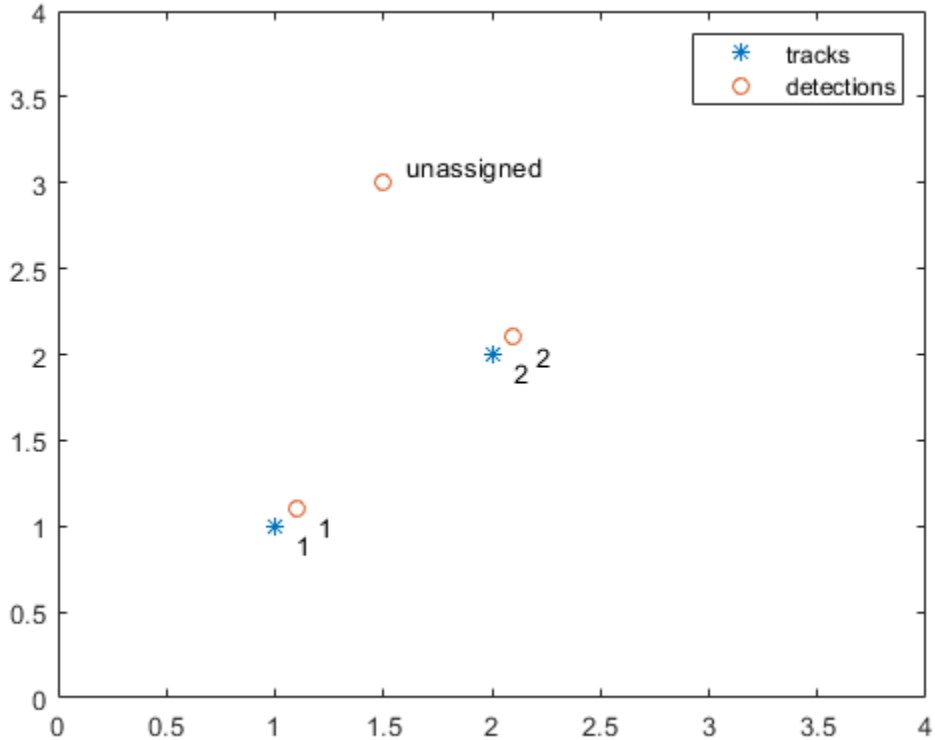
Display the unassigned detections.

```
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` | `assignsd` | `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 `fullmatix` 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] = assignsd(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, ...)` 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 $(Q\text{-}1)^{\text{th}}$ 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.

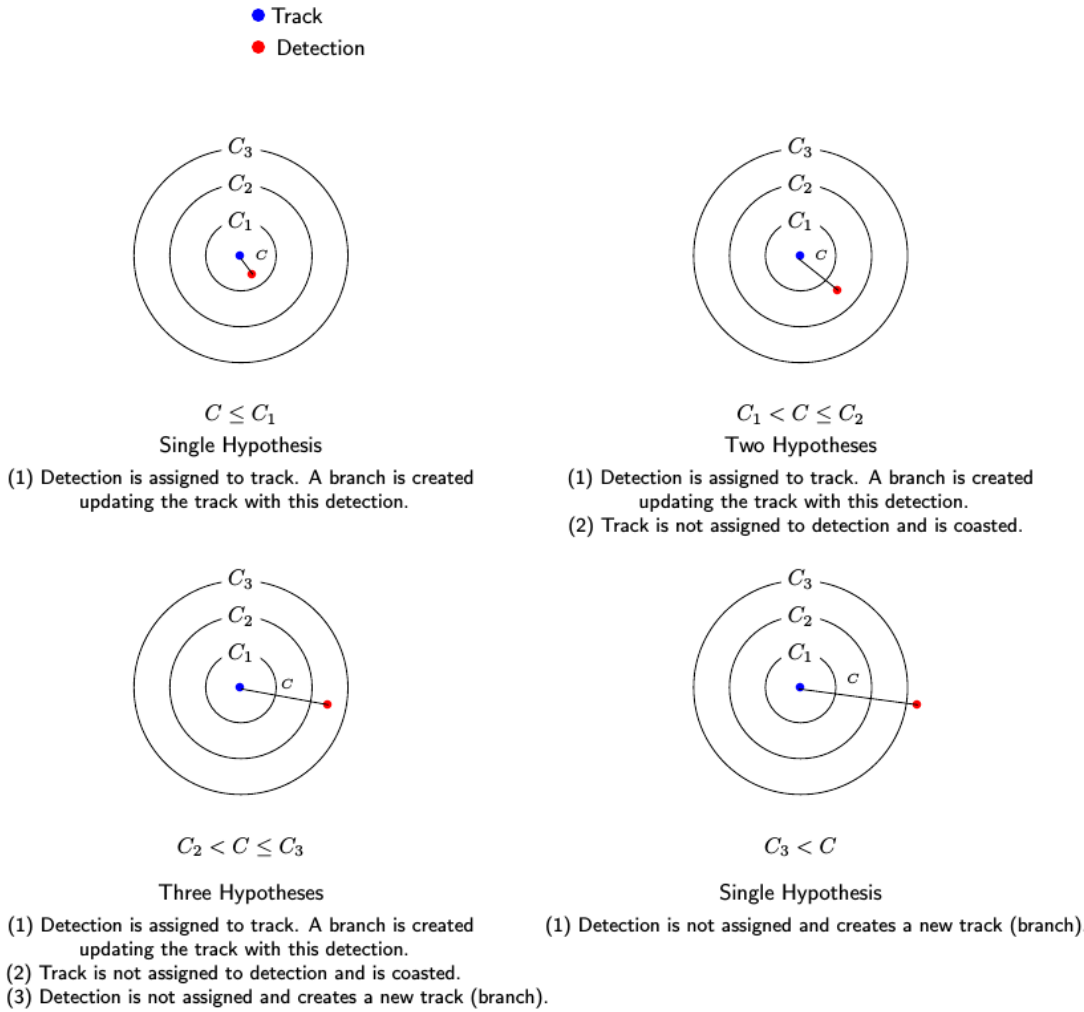
Data Types: `uint32`

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 <= C_2 <= C_3.

If the cost of an assignment is `C = 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.



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 $C_2 = 0$.

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.

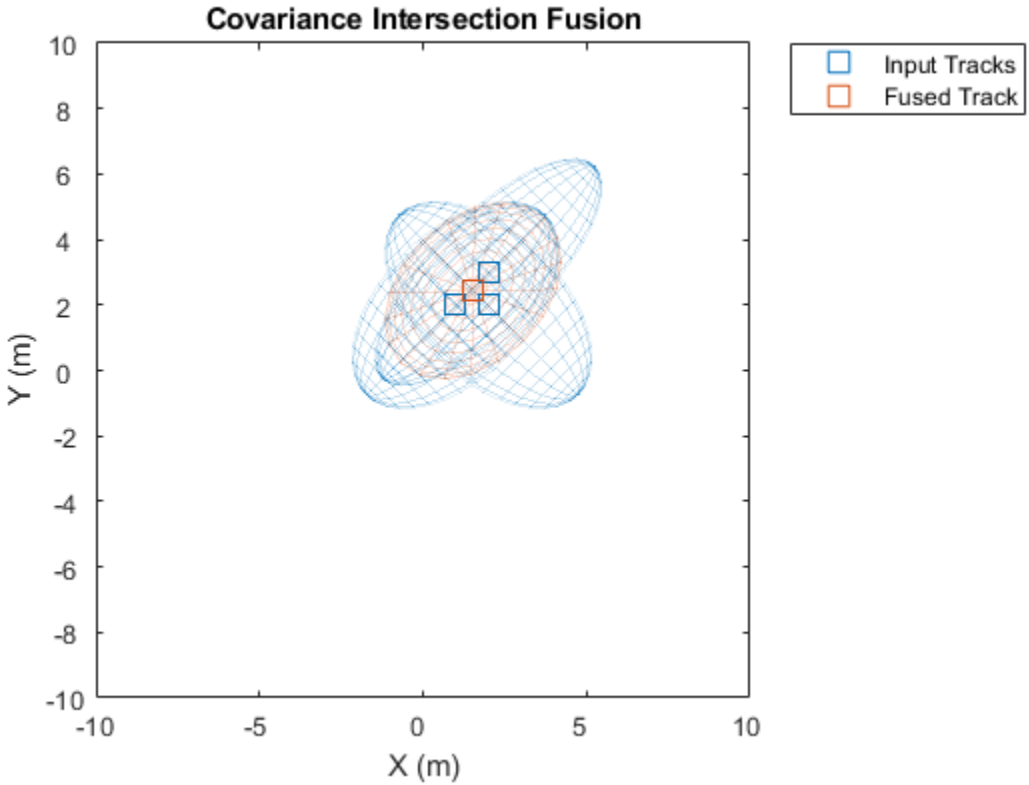
```
p(:,:,1) = [10 5 0; 5 10 0; 0 0 1];
p(:,:,2) = [10 -5 0; -5 10 0; 0 0 1];
p(:,:,3) = [12 9 0; 9 12 0; 0 0 1];
```

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.

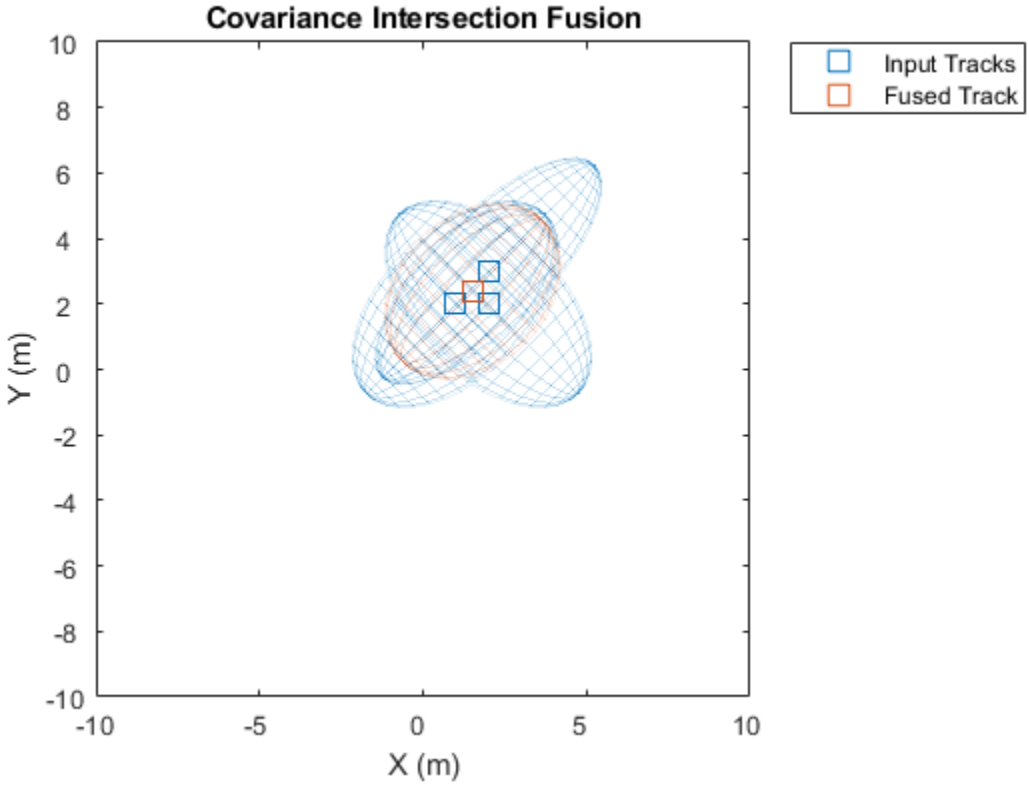
```
p(:,:,1) = [10 5 0; 5 10 0; 0 0 1];
p(:,:,2) = [10 -5 0; -5 10 0; 0 0 1];
p(:,:,3) = [12 9 0; 9 12 0; 0 0 1];
```

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.

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 1-486.

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.

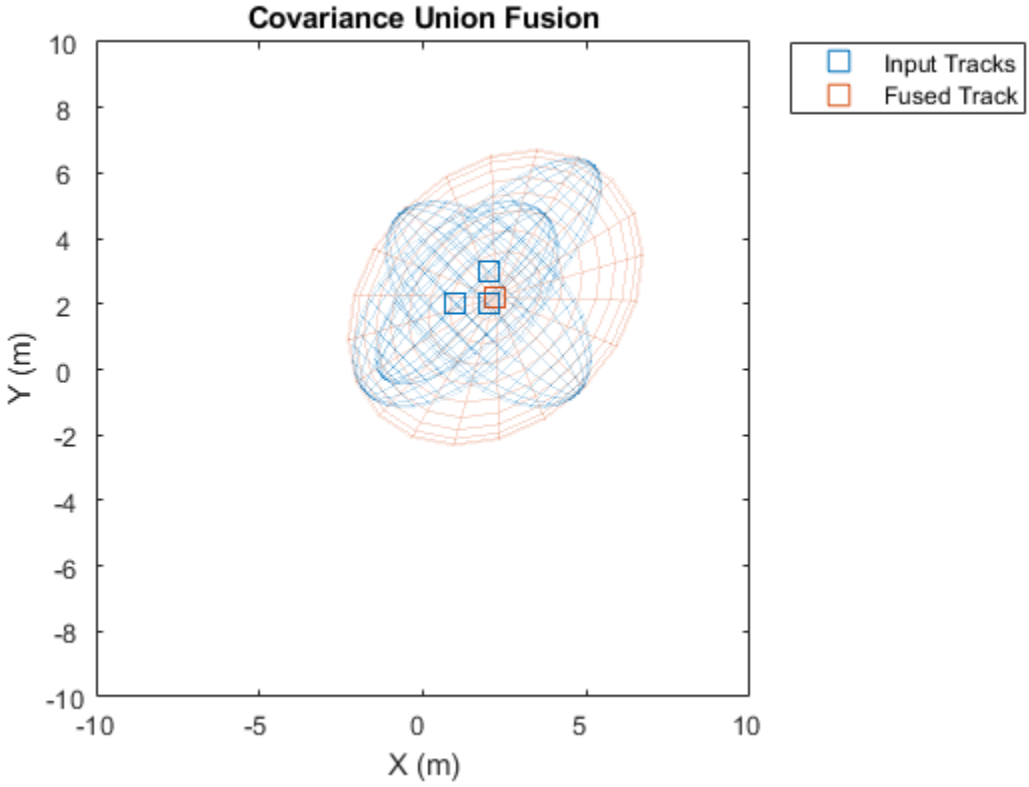
```
p(:,:,1) = [10 5 0; 5 10 0;0 0 1];
p(:,:,2) = [10 -5 0; -5 10 0;0 0 1];
p(:,:,3) = [12 9 0; 9 12 0;0 0 1];
```

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.

Definitions

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.

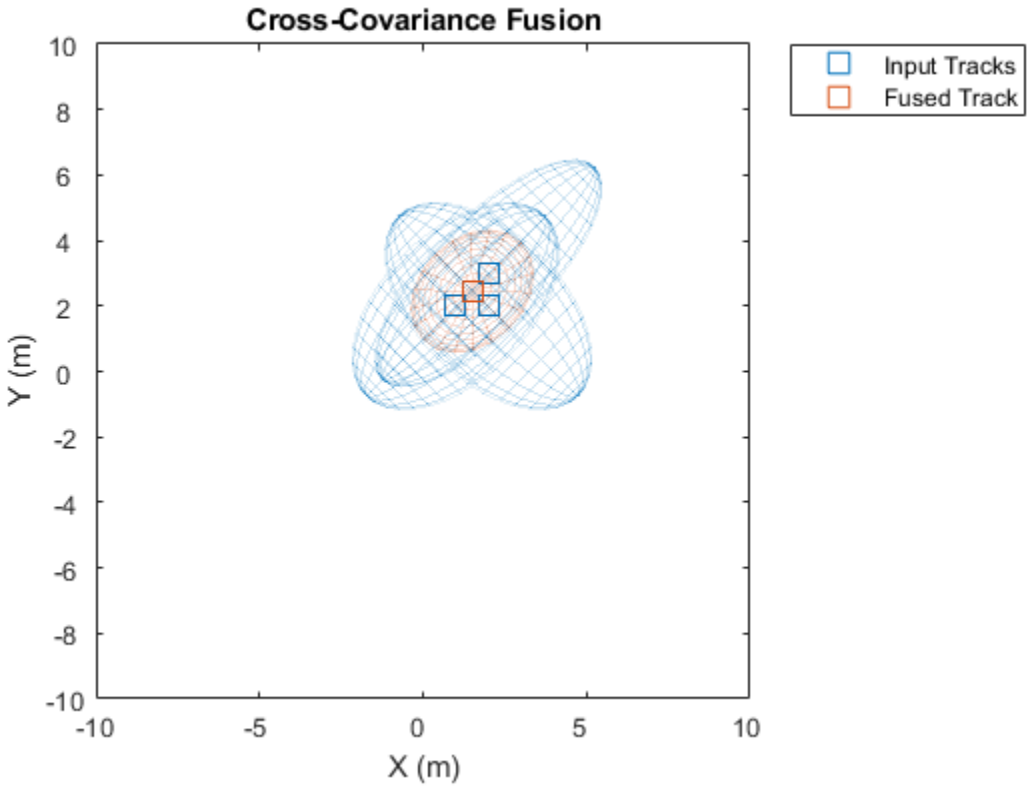
```
p(:,:,1) = [10 5 0; 5 10 0;0 0 1];
p(:,:,2) = [10 -5 0; -5 10 0;0 0 1];
p(:,:,3) = [12 9 0; 9 12 0;0 0 1];
```

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.

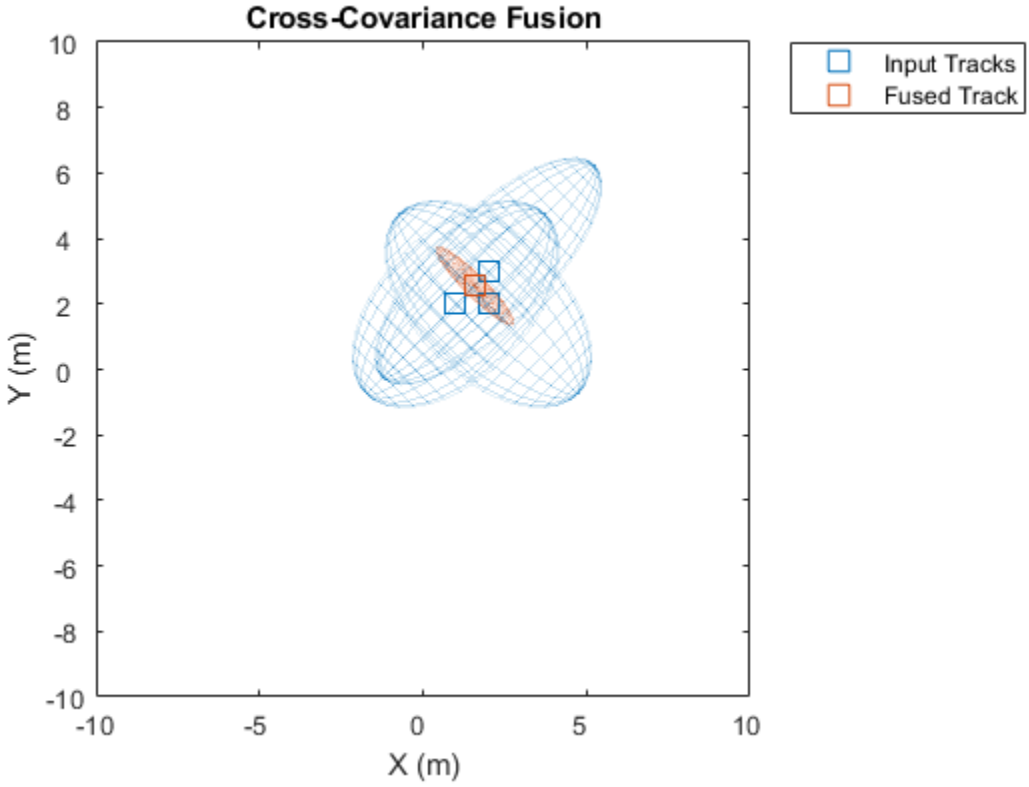
```
p(:,:,1) = [10 5 0; 5 10 0;0 0 1];
p(:,:,2) = [10 -5 0; -5 10 0;0 0 1];
p(:,:,3) = [12 9 0; 9 12 0;0 0 1];
```

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.

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]);
```

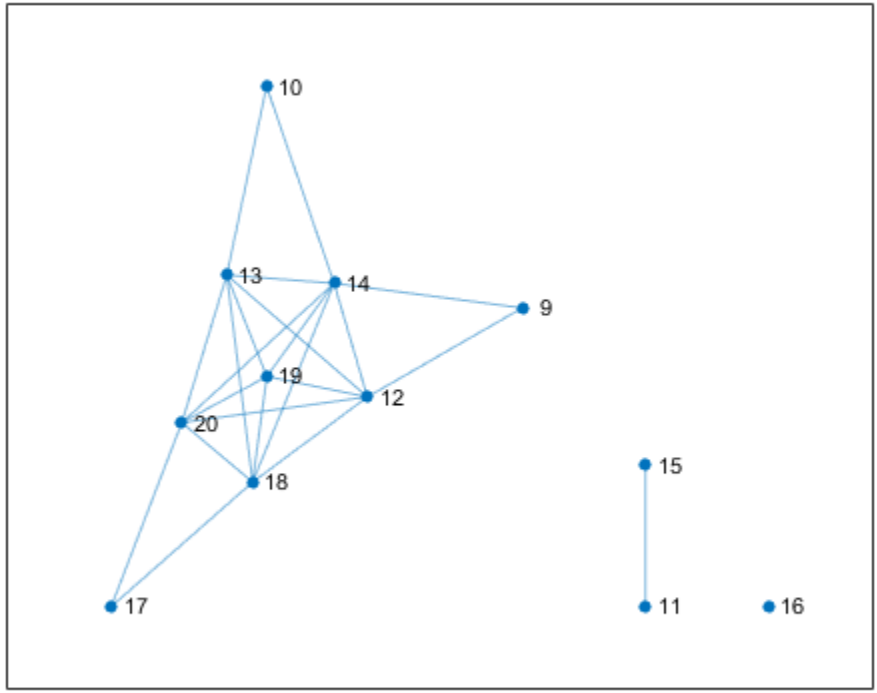
Get the list of clusters and the list of incompatible branches. The `clusters` matrix has three columns, therefore there are three clusters.

```
[clusters,incompBranches] = clusterTrackBranches(branchHistory);
size(clusters)
```

```
ans = 1×2

    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
```

```
hyps = 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 = 1×6

  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)
```

```
infoTable=20×6 table
    BranchID    PriorProbability    GlobalProbability    PrunedByProbability    PrunedB
    ________    ________________    _________________    ___________________    _______

       14           0.98901             0.098901              false               fal
       23                 1                   0.1             false               fal
       24                 1                   0.1             false               fal
       25           0.99889             0.099889              false               fal
       26           0.99889             0.099889              false               fal
       28                 1                     0             true                tru
       33                 1                     0             true                fal
       34                 1                   0.2             false               fal
       35                 1                   0.2             false               fal
       36           0.99989              0.19998              false               fal
       37           0.99989              0.19998              false               fal
       38                 1                     0             true                fal
       39                 1                   0.1             false               fal
       40                 1                   0.1             false               fal
       41                 1                   0.1             false               fal
       42                 1                   0.1             false               fal
      ⋮
```

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 `pruneTrackBrances` 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 `pruneTrackBrances` prunes branches that are incompatible with the current most likely branch in the most recent *N* scans. By default, `pruneTrackBrances` 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 1-513.

`[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
```

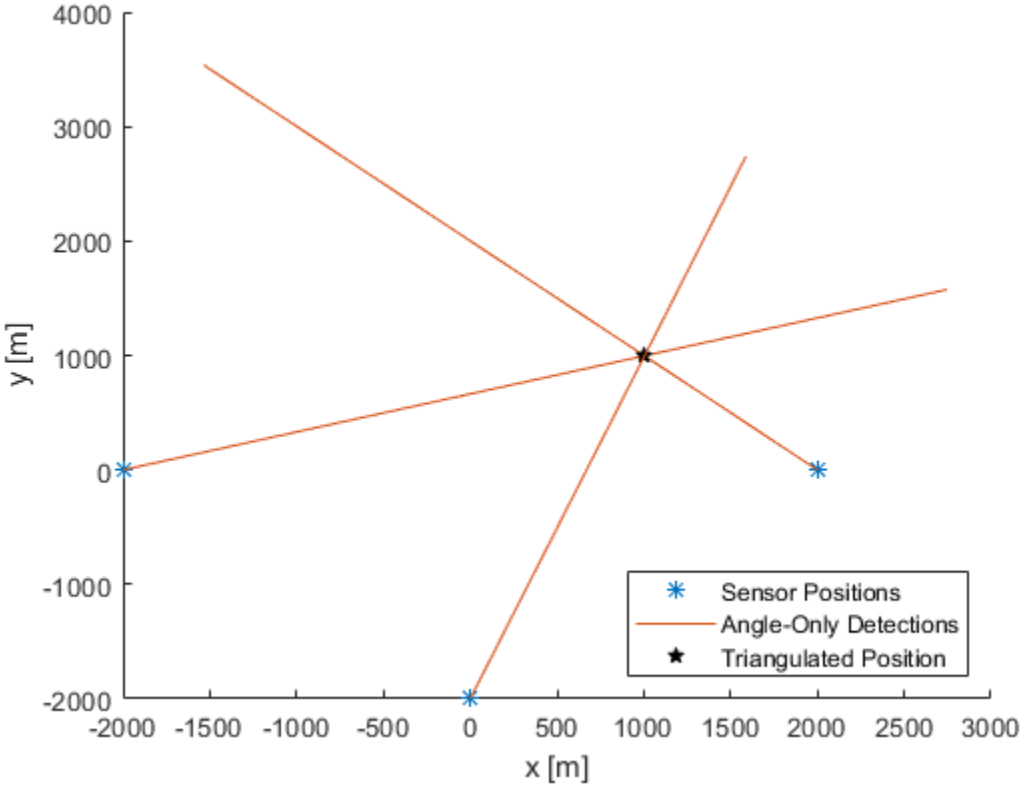
Plot the angle-only detections and the sensor positions. Specify a range of 5 km for plotting the direction vector. To specify the position of the origin, use the second measurement parameter because the sensor is located at the center of the platform. Convert the azimuth and elevation readings to Cartesian 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

Property	Definition
`Measurement`	Object measurements
`MeasurementNoise`	Measurement noise covariance matrix
`SensorIndex`	Unique ID of the sensor
`ObjectClassID`	Object classification
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters
`ObjectAttributes`	Additional information passed to tracker

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 `Frame` as `'spherical'` 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 `Orientation` is given as a frame rotation from the parent coordinate frame to the child coordinate frame. If `false`, then `Orientation` 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 `true` by default.

Parameter	Definition
`HasAzimuth`	A logical scalar that indicates if azimuth is included in the measurements. This parameter is `true` by default. If specified as a field, it must be set to `true`.
`HasRange`	A logical scalar that indicates if range is included in the measurements. This parameter must be specified as a field and set to `false`.
`HasVelocity`	A logical scalar that indicates if velocity is included in the measurements. This parameter is `false` by default. If specified as a field, it must be set to `false`.

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.

Extended Capabilities

C/C++ Code Generation

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

See Also

Classes
`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: 300000000
            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)`.

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`}

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 `HasOccusion` 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 `HasOccusion` 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.
`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 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]`.

Field	Description
`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:

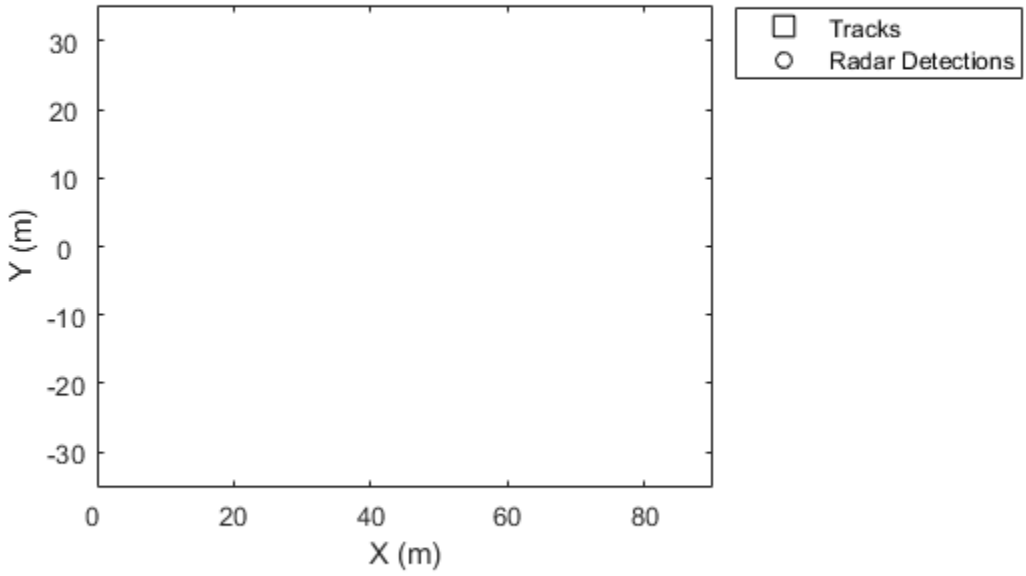
- `detectionPlotter`
- `orientationPlotter`
- `platformPlotter`
- `trackPlotter`
- `trajectoryPlotter`

Examples

Clear Specific Plotter Data

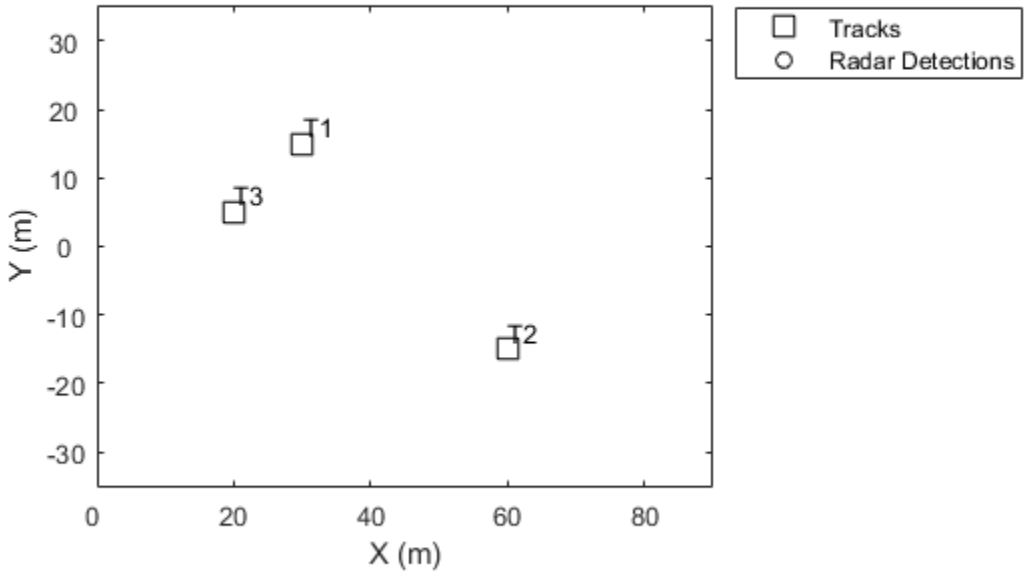
Create a theater plot. Add a track plotter and detection plotter to the theater plot.

```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35]);
tPlotter = trackPlotter(tp,'DisplayName','Tracks');
radarPlotter = detectionPlotter(tp,'DisplayName','Radar Detections');
```



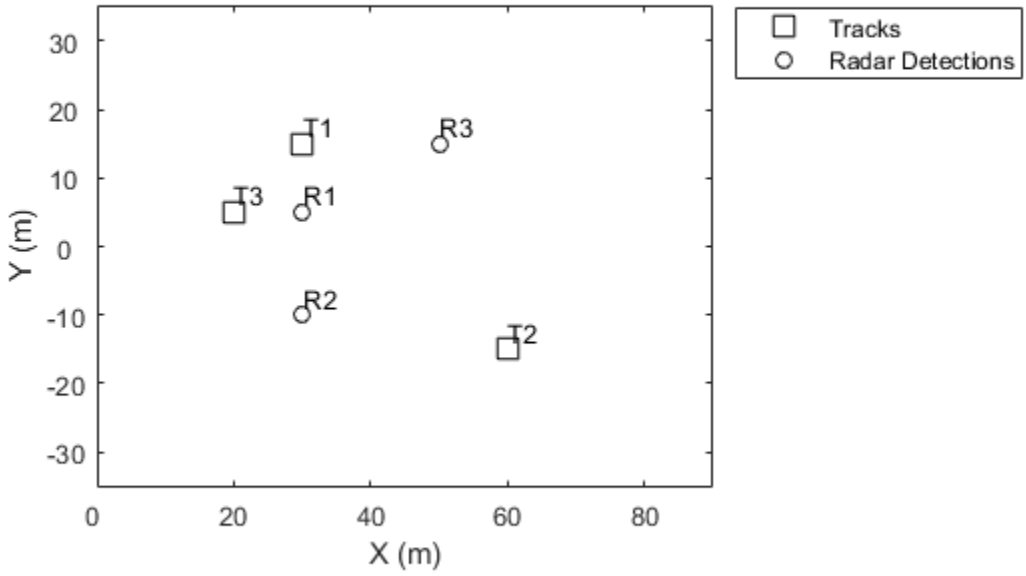
Plot a set of tracks in the track plotter.

```
trackPos = [30, 15, 1; 60, -15, 1; 20, 5, 1];
trackLabels = {'T1','T2','T3'};
plotTrack(tPlotter, trackPos, trackLabels)
```



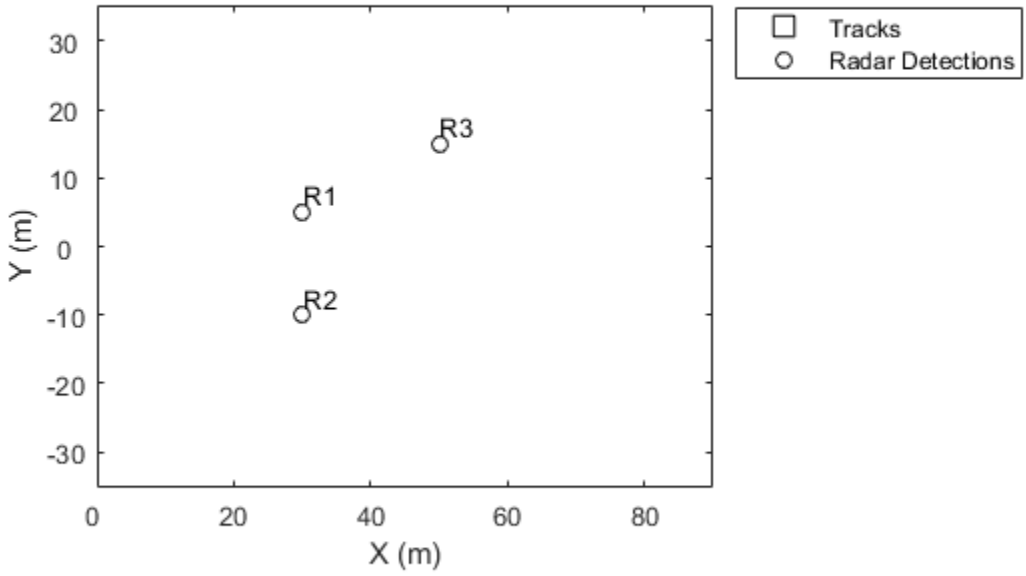
Plot a set of detections in the detection plotter.

```
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:

                   EIRP: 0
                    RCS: 0
             PlatformID: 1
           EmitterIndex: 1
         OriginPosition: [0 0 0]
```

```
          OriginVelocity: [0 0 0]
             Orientation: [1×1 quaternion]
             FieldOfView: [180 180]
         CenterFrequency: 300000000
               Bandwidth: 3000000
           WaveformType: 0
          ProcessingGain: 0
        PropagationRange: 0
    PropagationRangeRate: 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:

                   EIRP: 0
                    RCS: 0
             PlatformID: 1
           EmitterIndex: 1
          OriginPosition: [-7.0711 7.0711 0]
          OriginVelocity: [-7.0711 4.4409e-16 0]
             Orientation: [1×1 quaternion]
             FieldOfView: [180 180]
         CenterFrequency: 300000000
               Bandwidth: 3000000
           WaveformType: 0
          ProcessingGain: 0
        PropagationRange: 0
    PropagationRangeRate: 0
```

Convert Sonar Emission into Body Frame

Convert a sonar emission from scenario coordinates into body coordinates. Use `trackingScenario` to defined 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]`.

Field	Description
`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 `quaternion(1,0,0,0)` or, equivalently, `eye(3)`.

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

2

Classes in Sensor Fusion and Tracking Toolbox

stateinfo

Display state vector information

Syntax

```
stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the `State` property and the associated units.

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

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 2-11 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
`@initcvggiwphd` (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`. 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 `FilterInitilizationFcn`, you must also provide a transform function using the `SensorTransformFcn` property. Other than the default filter initialization function

`initcvggiwphd`, you can also use `initctggiwphd` and `initcaggiwphd` in Sensor Fusion and Tracking Toolbox.

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`.

Note that the default `SensorTransformFcn` is the sensor transform function of the filter returned by `FilterInitilizationFcn`. 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
`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.
`Orientation`	Relative orientation between frames, specified as a 3-by-3 rotation matrix. If the `IsParentToChild` property is set to `false`, then specify `Orientation` as the rotation from the child frame to the parent frame. If the `IsParentToChild` property is set to `true`, then specify `Orientation` as the rotation from the parent frame to the child frame.
`IsParentToChild`	Flag to indicate the direction of rotation between parent and child frame, specified as `true` or `false`. The default is `false`. See description of the `Orientation` field for details.
`HasAzimuth`	Indicates whether outputs contain azimuth components, specified as `true` or `false`.
`HasElevation`	Indicates whether outputs contain elevation components, specified as `true` or `false`.
`HasRange`	Indicates whether outputs contain range components, specified as `true` or `false`.
`HasVelocity`	Indicates whether outputs contains velocity components, specified as `true` or `false`.

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:

Fields	**Struct 1**	**Struct 2**
Frame	`'Spherical'`	`'Rectangular'`
OriginPosition	`[0;0;0]`	`[0;0;0]`

OriginVelocity	`[0;0;0]`	`[0;0;0]`
Orientation	`eye(3)`	`eye(3)`
IsParentToChild	`false`	`false`
HasAzimuth	`true`	`true`
HasElevation	`true`	`true`
HasRange	`true`	`true`
HasVelocity	`false`	`true`

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**

minAz	Minimum detectable azimuth in degrees.	`-10`
maxAz	Maximum detectable azimuth in degrees.	`10`
minEl	Minimum detectable elevation in degrees.	`-2.5`
maxEl	Maximum detectable elevation in degrees.	`2.5`
minRng	Minimum detectable range in meters.	`0`
maxRng	Maximum detectable range in meters.	`1000`
minRr	Minimum detectable range rate in meters per second.	N/A
maxRr	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',@initcvggiwphd)
```

```
config =

  trackingSensorConfiguration with properties:

                 SensorIndex: 3
                 IsValidTime: 0

               SensorLimits: [4x2 double]
           SensorResolution: [4x1 double]
         SensorTransformFcn: @cvmeas
   SensorTransformParameters: [1x2 struct]

     FilterInitializationFcn: @initcvggiwphd
         MaxNumDetsPerObject: Inf

              ClutterDensity: 1.0000e-03
       DetectionProbability: 0.9000
```

```
    MinDetectionProbability: 0.0500
```

Definitions

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

reset

Reset internal states

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

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^2)
N-by-3 matrix

Accelerometer readings in local sensor body coordinate system in m/s^2, 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

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

Syntax

```
fusemag(FUSE,magReadings,magReadingsCovariance)
```

Description

`fusemag(FUSE,magReadings,magReadingsCovariance)` fuses magnetometer data to correct the state estimate.

Input Arguments

`FUSE` — `ahrs10filter` object
object

Object of `ahrs10filter`.

`magReadings` — Magnetometer readings (µT)
3-element row vector

Magnetometer readings in µT, specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

`magReadingsCovariance` — Magnetometer readings error covariance (μT^2)
scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in μT^2, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

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

fusealtimeter

Correct states using altimeter data

Syntax

```
fusealtimeter(FUSE,altimeterReadings,altimeterReadingsCovariance)
```

Description

`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^2)
real scalar

Altimeter readings error covariance in m^2, specified as a real 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

correct

Correct states using direct state measurements

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

State	Units	Index
Delta Angle Bias (XYZ)	rad/s	7:9
Delta Velocity Bias (XYZ)	m/s	10:12
Geomagnetic Field Vector (NED)	µT	13:15
Magnetometer Bias (XYZ)	µT	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

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(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(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)^2)
`[1e-9,1e-9,1e-9]` (default) | scalar | three-element row vector

Multiplicative process noise variance from the gyroscope in (rad/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 (μT^2)
`[1e-6,1e-6,1e-6]` (default) | scalar | three-element row vector

Additive process noise for geomagnetic vector in μT^2, specified as positive real finite numbers.

Data Types: `single` | `double`

`MagnetometerBiasNoise` — Additive process noise for magnetometer bias (μT^2)
`[0.1,0.1,0.1]` (default) | scalar | three-element row vector

Additive process noise for magnetometer bias in μT^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)	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)	µT	13:15
Magnetometer Bias (XYZ)	µT	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

predict	Update states using accelerometer and gyroscope data
fusemag	Correct states using magnetometer data
fusealtimeter	Correct states using altimeter data
correct	Correct states using direct state measurements
pose	Current orientation and position estimate
reset	Reset internal states
stateinfo	Display state vector information

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('fuse10exampledata.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);
estOrient = 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 accelometer 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),estOrient(ii)] = pose(filt);
end
```

Calculate the RMS errors between the known true height and orientation and the output from the AHRS filter.

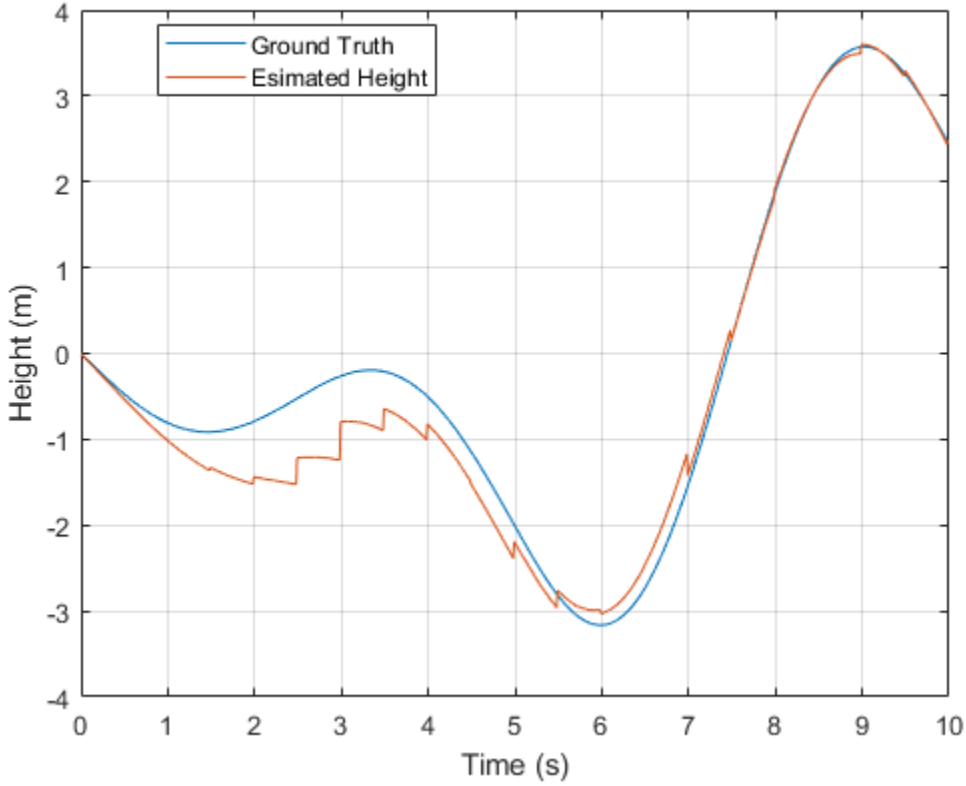
```
pErr = expectedHeight - estHeight;
qErr = rad2deg(dist(expectedOrient,estOrient));

pRMS = sqrt(mean(pErr.^2));
qRMS = sqrt(mean(qErr.^2));

fprintf('Altitude RMS Error\n');
fprintf('\t%.2f (meters)\n\n',pRMS);
```

```
Altitude RMS Error
    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','Esimated Height','location','best')
ylabel('Height (m)')
xlabel('Time (s)')
grid on


fprintf('Quaternion Distance RMS Error\n');
fprintf('\t%.2f (degrees)\n\n',qRMS);
```

```
Quaternion Distance RMS Error
    2.93 (degrees)
```



Extended Capabilities

C/C++ Code Generation

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

See Also

`ahrsfilter` | `insfilter`

Introduced in R2019a

pose

Current orientation and position estimate

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 `ErrorStateIMUGPSFuser` 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
`ErrorStateIMUGPSFuser`

`ErrorStateIMUGPSFuser` object, created by the `insfilter` function.

`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`

Introduced in R2019a

stateinfo

Display state vector information

Syntax

```
stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the `State` property of `FUSE`, an `ErrorStateIMUGPSFuser` object, and the associated units.

Input Arguments

FUSE — INS filter object
`ErrorStateIMUGPSFuser`

`ErrorStateIMUGPSFuser` object, created by the `insfilter` function.

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter`

Introduced in R2019a

reset

Reset internal states

Syntax

```
reset(FUSE)
```

Description

`reset(FUSE)` resets the `State`, `StateCovariance`, and internal integrators of `FUSE`, an `ErrorStateIMUGPSFuser` object, to their default values.

Input Arguments

FUSE — INS filter object
`ErrorStateIMUGPSFuser`

`ErrorStateIMUGPSFuser` object, created by the `insfilter` function.

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter`

Introduced in R2019a

predict

Update states using accelerometer and gyroscope data

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
`ErrorStateIMUGPSFuser`

`ErrorStateIMUGPSFuser` object, created by the `insfilter` function.

accelReadings — Accelerometer readings in local sensor body coordinate system (m/s^2)
3-element row vector

Accelerometer readings in m/s^2, 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`

Introduced in R2019a

fusemvo

Correct states using monocular visual odometry

Syntax

```
fusemvo(FUSE,position,positionCovariance,ornt,orntCovariance)
```

Description

`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
`ErrorStateIMUGPSFuser`

`ErrorStateIMUGPSFuser` object, created by the `insfilter` function.

`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^2)
scalar | 3-element vector | 3-by-3 matrix

Position measurement covariance of MVO in m^2, 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^2)
scalar | 3-element vector | 3-by-3 matrix

Orientation measurement covariance of monocular visual odometry in rad^2, specified as a scalar, 3-element vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter`

Introduced in R2019a

fusegps

Correct states using GPS data

Syntax

```
fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

Description

`fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)` fuses GPS data to correct the state estimate.

Input Arguments

FUSE — INS filter object
`ErrorStateIMUGPSFuser`

`ErrorStateIMUGPSFuser` object, created by the `insfilter` function.

`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^2)
scalar | 3-element row vector | 3-by-3 matrix

Position measurement covariance of GPS receiver in 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 $(m/s)^2$
scalar | 3-element row vector | 3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in $(m/s)^2$, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter`

Introduced in R2019a

correct

Correct states using direct state measurements

Syntax

```
correct(FUSE,idx,measurement,measurementCovariance)
```

Description

`correct(FUSE,idx,measurement,measurementCovariance)` corrects the state and state estimation error covariance of `FUSE`, an `ErrorStateIMUGPSFuser` 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
`ErrorStateIMUGPSFuser`

`ErrorStateIMUGPSFuser` object, created by the `insfilter` function.

`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

State	Units	Index
Velocity (NED)	m/s	8:10
Gyroscope Bias (XYZ)	rad/s	11:13
Accelerometer Bias (XYZ)	m/s^2	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 = numel(idx)-1`. If you do not correct orientation, then `M = numel(idx)`.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`insfilter`

Introduced in R2019a

ErrorStateIMUGPSFuser

Pose from IMU, GPS, and monocular visual odometry (MVO)

Description

The `ErrorStateIMUGPSFuser` object implements sensor fusion of IMU, GPS, and monocular visual odometry (MVO) data to estimate pose in the NED 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 `ErrorStateIMUGPSFuser` object uses an error-state Kalman filter to estimate these quantities.

Creation

Create an `ErrorStateIMUGPSFuser` to fuse IMU, GPS, and MVO data using `insfilter`:

```
filt = insfilter('errorstate');
```

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)^2)
`[1e-6 1e-6 1e-6]` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope in (rad/s)^2, 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)^2)
`[1e-9 1e-9 1e-9]` (default) | scalar | 3-element row vector

Additive process noise variance from the gyroscope bias in (rad/s)^2, 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^2)^2)
`[1e-4 1e-4 1e-4]` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer in (m/s^2)^2, 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/s²)²)

`[1e-4 1e-4 1e-4]` (default) | scalar | 3-element row vector

Additive process noise variance from accelerometer bias in $(m/s^2)^2$, 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)	m	5:7
Velocity (NED)	m/s	8:10
Gyroscope Bias (XYZ)	rad/s	11:13
Accelerometer Bias (XYZ)	m/s^2	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
δ Rotation Vector (XYZ)	1:3
δ Position (NED)	4:6
δ Velocity (NED)	7:9
δ Gyroscope Bias (XYZ)	10:12
δ Accelerometer Bias (XYZ)	13:15
δ Visual Odometry Scale (XYZ)	16

Note that because this is an error-state Kalman filter, it tracks the errors in the states. δ represents the error in the corresponding state.

Data Types: `single` | `double`

Object Functions

predict	Update states using accelerometer and gyroscope data
fusegps	Correct states using GPS data
fusemvo	Correct states using monocular visual odometry
correct	Correct states using direct state measurements
pose	Current orientation and position estimate
reset	Reset internal states
stateinfo	Display state vector information

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 = insfilter('ErrorState');
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.

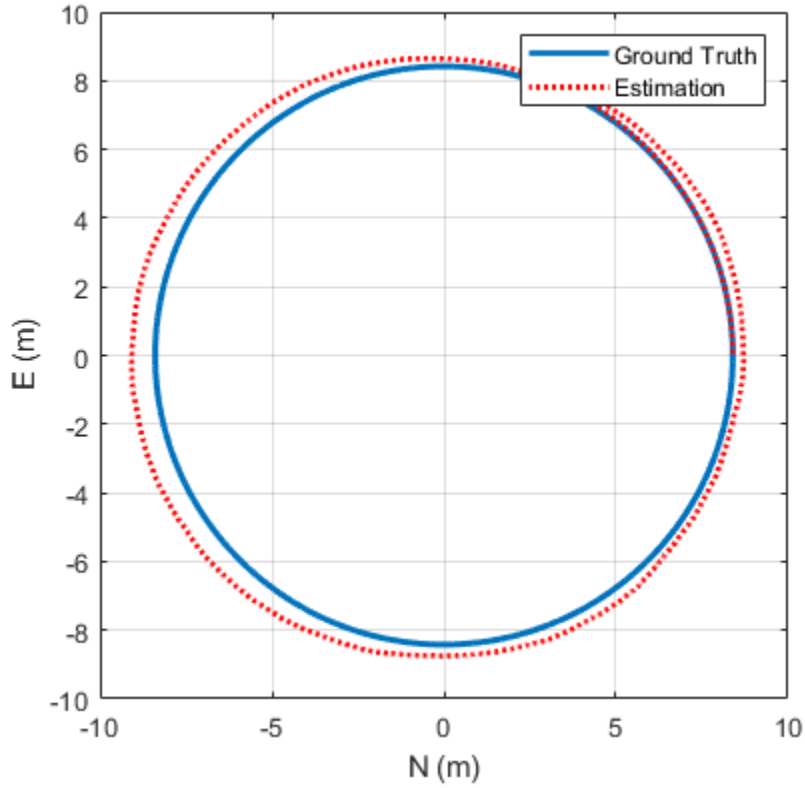
```
pErr = truePos - estPos;
qErr = rad2deg(dist(estOrient,trueOrient));

pRMS = sqrt(mean(pErr.^2));
qRMS = sqrt(mean(qErr.^2));
```

```
fprintf('Position RMS Error\n');
fprintf('\tX: %.2f, Y: %.2f, Z: %.2f (meters)\n\n',pRMS(1),pRMS(2),pRMS(3));

fprintf('Quaternion Distance RMS Error\n');
fprintf('\t%.2f (degrees)\n\n',qRMS);
```

```
Position RMS Error
    X: 0.40, Y: 0.24, Z: 0.05 (meters)

Quaternion Distance RMS Error
    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')
```



Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter`

Introduced in R2019a

stateinfo

Display state vector information

Syntax

```
stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the `State` property of the `AsyncMARGGPSFuser` object and the associated units.

Input Arguments

`FUSE` — `AsyncMARGGPSFuser` object
object

`AsyncMARGGPSFuser` object, created by the `insfilter` function.

Extended Capabilities

C/C++ Code Generation

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

See Also

`AsyncMARGGPSFuser` | `insfilter`

Introduced in R2019a

reset

Reset internal states

Syntax

```
reset(FUSE)
```

Description

`reset(FUSE)` resets the `State` and `StateCovariance` properties of the `AsyncMARGGPSFuser` object to their default values.

Input Arguments

`FUSE` — `AsyncMARGGPSFuser` object
object

`AsyncMARGGPSFuser` object, created by the `insfilter` function.

Extended Capabilities

C/C++ Code Generation

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

See Also

`AsyncMARGGPSFuser` | `insfilter`

Introduced in R2019a

predict

Update states based on motion model

Syntax

```
predict(FUSE,dt)
```

Description

`predict(FUSE,dt)` updates states based on the motion model.

Input Arguments

`FUSE` — `AsyncMARGGPSFuser` object
object

`AsyncMARGGPSFuser` object, created by the `insfilter` function.

`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

`AsyncMARGGPSFuser` | `insfilter`

Introduced in R2019a

pose

Current position, orientation, and velocity estimate

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` — `AsyncMARGGPSFuser` object
object

`AsyncMARGGPSFuser` object, created by the `insfilter` function.

`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` | `quaternion`

Extended Capabilities

C/C++ Code Generation

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

See Also

`AsyncMARGGPSFuser` | `insfilter`

Introduced in R2019a

fusemag

Correct states using magnetometer data

Syntax

```
fusemag(FUSE,magReadings,magReadingsCovariance)
```

Description

`fusemag(FUSE,magReadings,magReadingsCovariance)` fuses magnetometer data to correct the state estimate.

Input Arguments

`FUSE` — `AsyncMARGGPSFuser` object
object

`AsyncMARGGPSFuser` object, created by the `insfilter` function.

`magReadings` — Magnetometer readings (µT)
3-element row vector

Magnetometer readings in µT, specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

`magReadingsCovariance` — Magnetometer readings error covariance (μT^2)
scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in μT^2, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

`AsyncMARGGPSFuser` | `insfilter`

Introduced in R2019a

fusegyro

Correct states using gyroscope data

Syntax

```
fusegyro(FUSE,gyroReadings,gyroCovariance)
```

Description

`fusegyro(FUSE,gyroReadings,gyroCovariance)` fuses gyroscope data to correct the state estimate.

Input Arguments

`FUSE` — `AsyncMARGGPSFuser` object
object

`AsyncMARGGPSFuser` object, created by the `insfilter` function.

`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)^2)
scalar | 3-element row vector | 3-by-3 matrix

Covariance of gyroscope measurement error in (rad/s)^2, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

`AsyncMARGGPSFuser` | `insfilter`

Introduced in R2019a

fusegps

Correct states using GPS data

Syntax

```
fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

Description

`fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)` fuses GPS data to correct the state estimate.

Input Arguments

`FUSE` — `AsyncMARGGPSFuser` object
object

`AsyncMARGGPSFuser` object, created by the `insfilter` function.

`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^2)
3-by-3 matrix

Position measurement covariance of GPS receiver in 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 (m/s^2)
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

`AsyncMARGGPSFuser` | `insfilter`

Introduced in R2019a

fuseaccel

Correct states using accelerometer data

Syntax

```
fuseaccel(FUSE,acceleration,accelerationCovariance)
```

Description

`fuseaccel(FUSE,acceleration,accelerationCovariance)` fuses accelerometer data to correct the state estimate.

Input Arguments

`FUSE` — `AsyncMARGGPSFuser` object
object

`AsyncMARGGPSFuser` object, created by the `insfilter` function.

`acceleration` — Accelerometer readings in local sensor body coordinate system (m/s^2)
3-element row vector

Accelerometer readings in local sensor body coordinate system in m/s^2, specified as a 3-element row vector

Data Types: `single` | `double`

`accelerationCovariance` — Acceleration error covariance of accelerometer measurement ((m/s^2)^2)
scalar | 3-element row vector | 3-by-3 matrix

Acceleration error covariance of the accelerometer measurement in (m/s^2)^2, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

`AsyncMARGGPSFuser` | `insfilter`

Introduced in R2019a

correct

Correct states using direct state measurements

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` — `AsyncMARGGPSFuser` object
object

`AsyncMARGGPSFuser` object, created by the `insfilter` function.

`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

State	Units	Index
Velocity (NED)	m/s	11:13
Acceleration (NED)	m/s^2	14:16
Accelerometer Bias (XYZ)	m/s^2	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
`AsyncMARGGPSFuser` | `insfilter`

Introduced in R2019a

AsyncMARGGPSFuser

Pose from asynchronous MARG and GPS

Description

The `AsyncMARGGPSFuser` object implements sensor fusion of MARG and GPS data to estimate pose in the NED 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 `AsyncMARGGPSFuser` object uses a continuous-discrete extended Kalman filter to estimate these quantities.

Creation

Create an `AsyncMARGGPSFuser` to fuse asynchronous MARG and GPS data using `insfilter`:

```
filt = insfilter('AsyncIMU');
```

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 NED coordinate system ((rad/s)²)
`[0.005 0.005 0.005]` (default) | scalar | three-element row vector

Additive angular velocity process noise in the local NED coordinate system in $(rad/s)^2$, 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 NED 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 NED coordinate system (m^2)
`[1e-6 1e-6 1e-6]` (default) | scalar | three-element row vector

Additive position process noise in the local NED coordinate system in m^2, 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 NED 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 NED coordinate system ((m/s)²)
`[1e-6 1e-6 1e-6]` (default) | scalar | three-element row vector

Additive velocity process noise in the local NED coordinate system in $(m/s)^2$, 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 NED 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 NED coordinate system ((m/s²)²)
`[50 50 50]` (default) | scalar | three-element row vector

Additive acceleration process noise in $(m/s^2)^2$, 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 NED 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)²)
`[1e-10 1e-10 1e-10]` (default) | scalar | three-element row vector

Additive process noise variance from the gyroscope bias in $(rad/s)^2$, 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/s²)²)
`[1e-4 1e-4 1e-4]` (default) | positive scalar | three-element row vector

Additive process noise variance from accelerometer bias in $(m/s^2)^2$, 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 NED coordinate system (μT²)
`[1e-6 1e-6 1e-6]` (default) | positive scalar | three-element row vector

Additive process noise variance of geomagnetic vector in μT^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 NED coordinate system, respectively.
- If `GeomagneticVectorNoise` is a scalar, the single element is applied to each axis.

`MagnetometerBiasNoise` — Additive process noise variance from magnetometer bias (μT^2)
`[0.1 0.1 0.1]` (default) | positive scalar | three-element row vector

Additive process noise variance from magnetometer bias in μT^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
Position (NED)	m	8:10
Velocity (NED)	m/s	11:13
Acceleration (NED)	m/s^2	14:16
Accelerometer Bias (XYZ)	m/s^2	17:19
Gyroscope Bias (XYZ)	rad/s	20:22
Geomagnetic Field Vector (NED)	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
fuseaccel	Correct states using accelerometer data
fusegyro	Correct states using gyroscope data
fusemag	Correct states using magnetometer data
fusegps	Correct states using GPS data
correct	Correct states using direct state measurements
pose	Current position, orientation, and velocity estimate
reset	Reset internal states
stateinfo	Display state vector information

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 = insfilter('AsyncIMU');
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
```

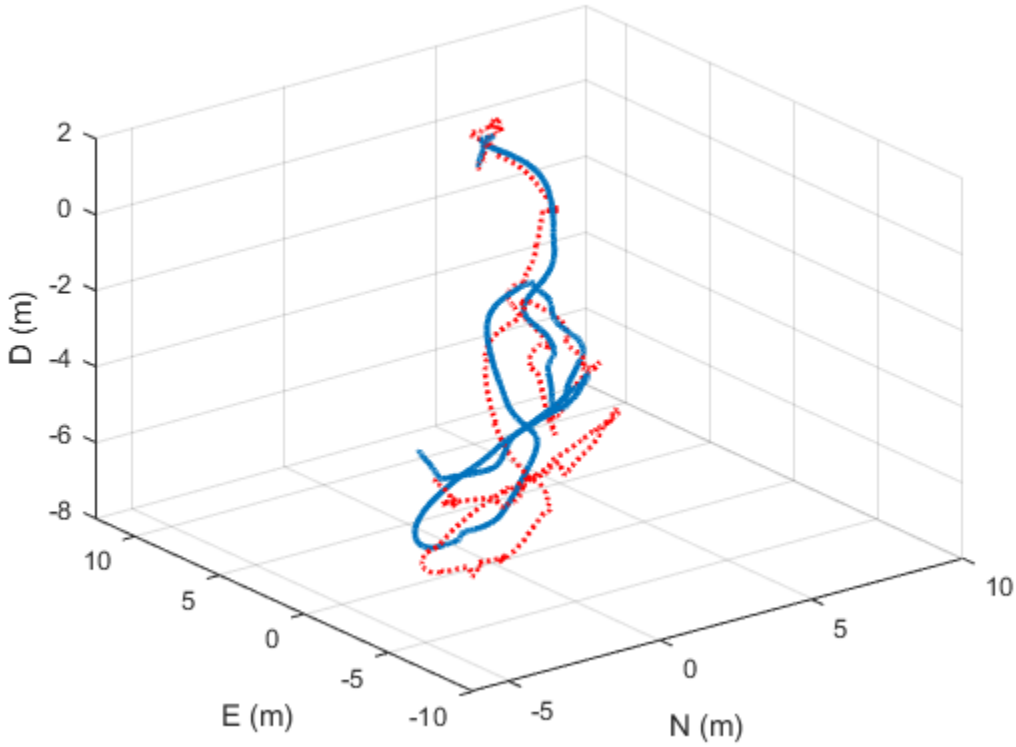
Calculate the RMS errors between the known true position and orientation and the output from the asynchronous IMU filter.

```
posErr = truePos - p;
qErr = rad2deg(dist(trueOrient,q));

pRMS = sqrt(mean(posErr.^2));
qRMS = sqrt(mean(qErr.^2));

fprintf('Position RMS Error\n');
fprintf('\tX: %.2f, Y: %.2f, Z: %.2f (meters)\n\n',pRMS(1),pRMS(2),pRMS(3));

fprintf('Quaternion Distance RMS Error\n');
fprintf('\t%.2f (degrees)\n\n', qRMS);
```

```
Position RMS Error
    X: 0.55, Y: 0.71, Z: 0.74 (meters)

Quaternion Distance RMS Error
    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 `AsyncMARGGPSFuser`

`AsyncMARGGPSFuser` 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.
- ν_N, ν_E, ν_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.
- $geomagneticFieldVector_N$, $geomagneticFieldVector_E$, $geomagneticFieldVector_D$ –– 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 process equation, $\dot{x} = f(x) + w$, w is the process noise, $\dot{x}$ is the derivative of x, and:

$$f(x) = \begin{bmatrix} \frac{-(q_1)(angVel_X) - (q_2)(angVel_Y) - (q_3)(angVel_Z)}{2} \\ \frac{(q_0)(angVel_X) - (q_3)(angVel_Y) + (q_1)(angVel_Z)}{2} \\ \frac{(q_3)(angVel_X) + (q_0)(angVel_Y) - (q_1)(angVel_Z)}{2} \\ \frac{(q_1)(angVel_X) - (q_2)(angVel_Y) + (q_0)(angVel_Z)}{2} \\ 0 \\ 0 \\ 0 \\ \nu_N \\ \nu_E \\ \nu_D \\ accel_N \\ accel_E \\ accel_D \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter`

Introduced in R2019a

correct

Correct states using direct state measurements

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` — `MARGGPSFuser` object

object

Object of `MARGGPSFuser`, created by the `insfilter` function.

`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

State	Units	Index
Delta Angle Bias (XYZ)	rad	11:13
Delta Velocity Bias (XYZ)	m/s	14:16
Geomagnetic Field Vector (NED)	µT	17:19
Magnetometer Bias (XYZ)	µT	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

`MARGGPSFuser` | `insfilter`

Introduced in R2018b

fusegps

Correct states using GPS data

Syntax

```
fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

Description

`fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)` fuses GPS data to correct the state estimate.

Input Arguments

`FUSE` — `MARGGPSFuser` object
object

Object of `MARGGPSFuser`, created by the `insfilter` function.

`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^2)
3-by-3 matrix

Position measurement covariance of GPS receiver in 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 (m/s^2)
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

`MARGGPSFuser` | `insfilter`

Introduced in R2018b

fusemag

Correct states using magnetometer data

Syntax

```
fusemag(FUSE,magReadings,magReadingsCovariance)
```

Description

`fusemag(FUSE,magReadings,magReadingsCovariance)` fuses magnetometer data to correct the state estimate.

Input Arguments

`FUSE` — `MARGGPSFuser` object
object

Object of `MARGGPSFuser`, created by the `insfilter` function.

`magReadings` — Magnetometer readings (µT)
3-element row vector

Magnetometer readings in µT, specified as a 3-element row vector of finite real numbers.

Data Types: `single` | `double`

`magReadingsCovariance` — Magnetometer readings error covariance (μT^2)
scalar | 3-element row vector | 3-by-3 matrix

Magnetometer readings error covariance in μT^2, specified as a scalar, 3-element row vector, or 3-by-3 matrix.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

`MARGGPSFuser` | `insfilter`

Introduced in R2018b

pose

Current orientation and position estimate

Syntax

```
[position,orientation ] = pose(FUSE)
[position,orientation ] = pose(FUSE,format)
```

Description

`[position,orientation ] = pose(FUSE)` returns the current estimate of the pose.

`[position,orientation ] = pose(FUSE,format)` returns the current estimate of the pose with orientation in the specified orientation format.

Input Arguments

`FUSE` — `MARGGPSFuser` object
object

Object of `MARGGPSFuser`, created by the `insfilter` function.

`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`

Extended Capabilities

C/C++ Code Generation

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

See Also

`MARGGPSFuser` | `insfilter`

Introduced in R2018b

predict

Update states using accelerometer and gyroscope data

Syntax

```
predict(FUSE,accelReadings,gyroReadings)
```

Description

`predict(FUSE,accelReadings,gyroReadings)` fuses accelerometer and gyroscope data to update the state estimate.

Input Arguments

`FUSE` — `MARGGPSFuser` object
object

Object of `MARGGPSFuser`, created by the `insfilter` function.

`accelReadings` — Accelerometer readings in local sensor body coordinate system (m/s^2)
3-element row vector

Accelerometer readings in m/s^2, 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

`MARGGPSFuser` | `insfilter`

Introduced in R2018b

reset

Reset internal states

Syntax

```
reset(FUSE)
```

Description

`reset(FUSE)` resets the State, StateCovariance, and internal integrators to their default values.

Input Arguments

`FUSE` — `MARGGPSFuser` object
object

Object of `MARGGPSFuser`, created by the `insfilter` function.

Extended Capabilities

C/C++ Code Generation

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

See Also

`MARGGPSFuser` | `insfilter`

Introduced in R2018b

stateinfo

Display state vector information

Syntax

```
stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the State property and the associated units.

Input Arguments

FUSE — MARGGPSFuser object
object

Object of `MARGGPSFuser`, created by the `insfilter` function.

Extended Capabilities

C/C++ Code Generation

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

See Also

`MARGGPSFuser` | `insfilter`

Introduced in R2018b

MARGGPSFuser

Estimate pose from MARG and GPS data

Description

The `MARGGPSFuser` object implements sensor fusion of MARG and GPS data to estimate pose in the NED 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 `MARGGPSFuser` object uses an extended Kalman filter to estimate these quantities.

Creation

Create a `MARGGPSFuser` using `insfilter`:

```
filt = insfilter('NonholonomicHeading',false,'Magnetometer',true);
```

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 $(rad/s)^2$
`1e-9` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope in $(rad/s)^2$, 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 $(rad/s)^2$
`1e-10` (default) | positive scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope bias in $(rad/s)^2$, 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 $(m/s^2)^2$
`1e-4` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer in $(m/s^2)^2$, 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 $(m/s^2)^2$
`1e-4` (default) | positive scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer bias in $(m/s^2)^2$, 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^2)
`1e-6` (default) | positive scalar | 3-element row vector

Additive process noise for geomagnetic vector in μT^2, 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 (μT^2)
`0.1` (default) | positive scalar | 3-element row vector

Additive process noise for magnetometer bias in μT^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)	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)	µT	17:19
Magnetometer Bias (XYZ)	µT	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

correct	Correct states using direct state measurements
fusegps	Correct states using GPS data
fusemag	Correct states using magnetometer data
pose	Current orientation and position estimate
predict	Update states using accelerometer and gyroscope data
reset	Reset internal states
stateinfo	Display state vector information

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 `MARGGPSFuser` filter object.

```
f = insfilter;
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),
```

```
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

`MARGGPSFuser` uses a 22-axis extended Kalman filter structure to estimate pose in the NED reference frame. The state is defined as:

$$x = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ position_{\mathrm{N}} \\ position_{\mathrm{E}} \\ position_{\mathrm{D}} \\ v_{\mathrm{N}} \\ v_{\mathrm{E}} \\ v_{\mathrm{D}} \\ \Delta\theta bias_{\mathrm{X}} \\ \Delta\theta bias_{\mathrm{Y}} \\ \Delta\theta bias_{\mathrm{Z}} \\ \Delta v bias_{\mathrm{X}} \\ \Delta v bias_{\mathrm{Y}} \\ \Delta v bias_{\mathrm{Z}} \\ geomagneticFieldVector_{\mathrm{N}} \\ geomagneticFieldVector_{\mathrm{E}} \\ geomagneticFieldVector_{\mathrm{D}} \\ magbias_{\mathrm{X}} \\ magbias_{\mathrm{Y}} \\ magbias_{\mathrm{Z}} \end{bmatrix}$$

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.
- $position_{\mathrm{N}}$, $position_{\mathrm{E}}$, $position_{\mathrm{D}}$ –– 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.

- $\Delta\theta bias_X$, $\Delta\theta bias_Y$, $\Delta\theta bias_Z$ –– 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,

$$x_{k|k-1} = f(\widehat{x}_{k-1|k-1}, u_k)$$

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:

$$x_{k|k-1} = \begin{bmatrix} q_0 - q_1\left(\frac{\Delta\theta_X - \Delta\theta bias_X}{2}\right) - q_2\left(\frac{\Delta\theta_Y - \Delta\theta bias_Y}{2}\right) - q_3\left(\frac{\Delta\theta_Z - \Delta\theta bias_Z}{2}\right) \\ q_1 + q_0\left(\frac{\Delta\theta_X - \Delta\theta bias_X}{2}\right) - q_3\left(\frac{\Delta\theta_Y - \Delta\theta bias_Y}{2}\right) + q_2\left(\frac{\Delta\theta_Z - \Delta\theta bias_Z}{2}\right) \\ q_2 + q_3\left(\frac{\Delta\theta_X - \Delta\theta bias_X}{2}\right) + q_0\left(\frac{\Delta\theta_Y - \Delta\theta bias_Y}{2}\right) - q_1\left(\frac{\Delta\theta_Z - \Delta\theta bias_Z}{2}\right) \\ q_3 - q_2\left(\frac{\Delta\theta_X - \Delta\theta bias_X}{2}\right) + q_1\left(\frac{\Delta\theta_Y - \Delta\theta bias_Y}{2}\right) + q_0\left(\frac{\Delta\theta_Z - \Delta\theta bias_Z}{2}\right) \\ position + (\Delta t)(v_{k-1}) \\ v_{k-1} + (\Delta t)\left(R\left(q_0, q_1, q_2, q_3\right)\left(\frac{a_{k-1} + a_k}{2}\right) + g\right) \\ \Delta\theta bias_{k-1} \end{bmatrix}$$

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.
- Δt –– 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

`insfilter`

Topics

“Estimate Position and Orientation of a Ground Vehicle”

Introduced in R2018b

correct

Correct states using direct state measurements

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` — `NHConstrainedIMUGPSFuser` object
object

Object of `NHConstrainedIMUGPSFuser`, created by the `insfilter` function.

`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

State	Units	Index
Velocity (NED)	m/s	11:13
Accelerometer Bias (XYZ)	m/s^2	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
`NHConstrainedIMUGPSFuser` | `insfilter`

Introduced in R2018b

fusegps

Correct states using GPS data

Syntax

```
fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)
```

Description

`fusegps(FUSE,position,positionCovariance,velocity,
velocityCovariance)` fuses GPS data to correct the state estimate.

Input Arguments

`FUSE` — `NHConstrainedIMUGPSFuser` object
object

Object of `NHConstrainedIMUGPSFuser`, created by the `insfilter` function.

`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^2)
3-by-3 matrix

Position measurement covariance of GPS receiver in 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 (m/s^2)
3-by-3 matrix

Velocity measurement covariance of the GPS receiver in the local NED coordinate system in m/s^2, specified as a 3-by-3 matrix.

Data Types: `single` | `double`

Extended Capabilities

C/C++ Code Generation

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

See Also

`NHConstrainedIMUGPSFuser` | `insfilter`

Introduced in R2018b

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` method. 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 (α) and rate (β), are both divided by forgetting factor n:

$$\alpha_{k+1|k} = \frac{\alpha_k}{n_k}$$

$$\beta_{k+1|k} = \frac{\beta_k}{n_k}$$

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

Detections, specified as a *K*-element cell array of `objectDetection`, 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 `HasAdditiveMeasurmentNoise` 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` method. 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

append	Append two ggiwphd filter objects
correct	Correct ggiwphd filter with detections
correctUndetected	Correct ggiwphd filter with no detection hypothesis
extractState	Extract target state estimates from the ggiwphd 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 ggiwphd filter
predict	Predict probability hypothesis density of ggiwphd filter
prune	Prune the filter by removing selected components
scale	Scale weights of components in the density

clone	Create duplicate ggiwphd 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;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 =

    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]
                         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
```

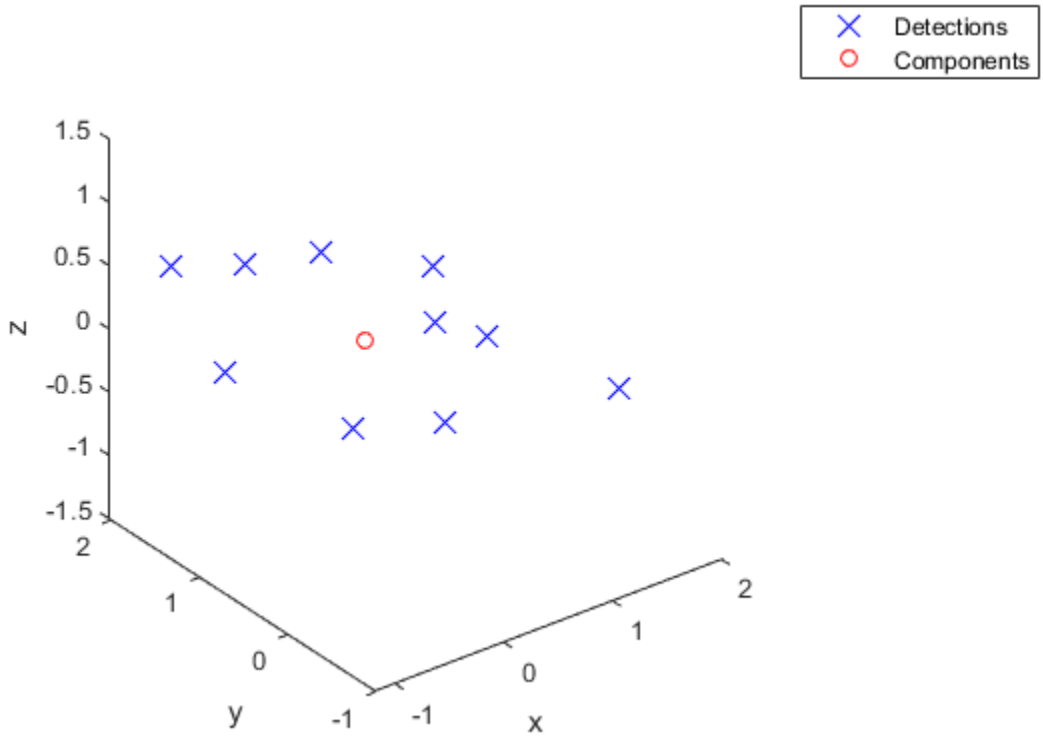
Extract state estimates and detections.

```
targetStates = extractState(phd,0.5);
tStates = targetStates.State

d = [detections{:}];
measurements = [d.Measurement];
```

```
tStates =

    0.1947
    0.9733
    0.8319
    4.1599
   -0.0124
   -0.0621
```

Visualize the results.

```
figure()
plot3(measurements(1,:),measurements(2,:),measurements(3,:),'x','MarkerSize',10,'Marke
hold on;
plot3( tStates(1,:),tStates(3,:),tStates(5,:),'ro');
xlabel('x')
ylabel('y')
zlabel('z')
legend('Detections','Components')
```



References

[1] Granstorm, 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] Granstorm, 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

`partitionDetections` | `trackerPHD` | `trackingSensorConfiguration`

Introduced in R2019a

append

Append two `ggiwphd` 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` — ggiwphd filter
function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

`phd2` — ggiwphd filter
function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

Extended Capabilities

C/C++ Code Generation

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

See Also

`ggiwphd` | `trackerPHD`

Introduced in R2019a

clone

Create duplicate `ggiwphd` filter object

Syntax

```
phd2 = clone(phd1)
```

Description

`phd2 = clone(phd1)` creates a duplicate `ggiwphd` filter, `phd2`, from a `ggiwphd` filter, `phd1`.

Input Arguments

`phd1` — ggiwphd filter
function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

Output Arguments

`phd2` — ggiwphd filter
function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

Extended Capabilities

C/C++ Code Generation

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

See Also

`ggiwphd` | `trackerPHD`

Introduced in R2019a

correct

Correct `ggiwphd` filter with detections

Syntax

```
correct(phd,detectionIndices,likelihood)
```

Description

`correct(phd,detectionIndices,likelihood)` corrects `ggiwphd` filter object, `phd`, using detections specified by `detectionIndices` and corresponding detection likelihoods, `likelihood`.

Input Arguments

`phd` — ggiwphd filter

function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

`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` | `trackerPHD`

Introduced in R2019a

correctUndetected

Correct `ggiwphd` filter with no detection hypothesis

Syntax

```
correctUndetect(phd,Pd)
correctUndetect(phd,Pd,PzeroDets)
```

Description

`correctUndetect(phd,Pd)` corrects the `ggiwphd` filter, `phd`, with the sensor detection probability, `Pd`. The function also calculates the probability of generating zero detections using the current Gamma distribution of the filter.

`correctUndetect(phd,Pd,PzeroDets)` allows you to specify the conditional probability for generating zero detections using `PzeroDets`.

Input Arguments

phd — ggiwphd filter
function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

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` | `trackerPHD`

Introduced in R2019a

extractState

Extract target state estimates from the `ggiwphd` filter

Syntax

```
[States,Indices] = extractState(phd,threshold)
```

Description

`[States,Indices] = extractState(phd,threshold)` returns all sates of components, `States`, whose weights are above the threshold given by `threshold`, and their corresponding indices, `Indices`, in the `ggiwphd` filter, `phd`.

Input Arguments

`phd` — ggiwphd filter
function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

`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 an 1-by-*N* array of structure, where *N* is the number of extracted states. Each structure contains the following fields:

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.

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` | `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` — ggiwphd filter
function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

`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` | `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 `ggiwphd` filter, `phd`.

Input Arguments

`phd` — ggiwphd filter
function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

`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` | `trackerPHD`

Introduced in R2019a

merge

Merge components in the density of `ggiwphd` filter

Syntax

```
merge(phd,mergingThreshold)
```

Description

`merge(phd,mergingThreshold)` merges components whose Kullback-Leibler difference is below the threshold, `mergingThreshold`.

Input Arguments

`phd` — ggiwphd filter
function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

`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` | `trackerPHD`

Introduced in R2019a

predict

Predict probability hypothesis density of ggiwphd filter

Syntax

```
predict(phd,dt)
```

Description

`predict(phd,dt)` predicts the density of the `ggiwphd` filter object, `phd`, forward by time step, `dt`. The function predicts all the three distributions (Gamma, Gaussian,and Inverse-Wishart) of `ggiwphd` filter.

Input Arguments

`phd` — ggiwphd filter
function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

`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` | `trackerPHD`

Introduced in R2019a

prune

Prune the filter by removing selected components

Syntax

```
prune(phd,pruneIndices)
```

Description

`prune(phd,pruneIndices)` removes components in `ggiwphd` filter object, `phd`, specified by `pruneIndices`.

Input Arguments

`phd` — ggiwphd filter
function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

`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` | `trackerPHD`

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 `ggiwphd` filter, `phd`, by factor, `ScaleFactor`.

Input Arguments

phd — ggiwphd filter

function handle

`ggiwphd` filter, specified as a function handle.

Example: `phd`

Data Types: `function_handle`

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` | `trackerPHD`

Introduced in R2019a

pose

Current orientation and position estimate

Syntax

```
[orientation, position] = pose(FUSE)
[orientation, position] = pose(FUSE,format)
```

Description

`[orientation, position] = pose(FUSE)` returns the current estimate of the pose.

`[orientation, position] = pose(FUSE,format)` returns the current estimate of the pose with orientation in the specified orientation format.

Input Arguments

`FUSE` — `NHConstrainedIMUGPSFuser` object
object

Object of `NHConstrainedIMUGPSFuser`, created by the `insfilter` function.

`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

`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`

`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`

Extended Capabilities

C/C++ Code Generation

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

See Also

`NHConstrainedIMUGPSFuser` | `insfilter`

Introduced in R2018b

predict

Update states using accelerometer and gyroscope data

Syntax

```
predict(FUSE,accelReadings,gyroReadings)
```

Description

`predict(FUSE,accelReadings,gyroReadings)` fuses accelerometer and gyroscope data to update the state estimate.

Input Arguments

`FUSE` — `NHConstrainedIMUGPSFuser` object
object

Object of `NHConstrainedIMUGPSFuser`, created by the `insfilter` function.

`accelReadings` — Accelerometer readings in local sensor body coordinate system (m/s^2)
3-element row vector

Accelerometer readings in m/s^2, 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

`NHConstrainedIMUGPSFuser` | `insfilter`

Introduced in R2018b

reset

Reset internal states

Syntax

```
reset(FUSE)
```

Description

`reset(FUSE)` resets the State, StateCovariance, and internal integrators to their default values.

Input Arguments

`FUSE` — `NHConstrainedIMUGPSFuser` object
object

Object of `NHConstrainedIMUGPSFuser`, created by the `insfilter` function.

Extended Capabilities

C/C++ Code Generation

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

See Also

`NHConstrainedIMUGPSFuser` | `insfilter`

Introduced in R2018b

stateinfo

Display state vector information

Syntax

```
stateinfo(FUSE)
```

Description

`stateinfo(FUSE)` displays the meaning of each index of the State property and the associated units.

Input Arguments

`FUSE` — `NHConstrainedIMUGPSFuser` object
object

Object of `NHConstrainedIMUGPSFuser`, created by the `insfilter` function.

Extended Capabilities

C/C++ Code Generation

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

See Also

`NHConstrainedIMUGPSFuser` | `insfilter`

Introduced in R2018b

NHConstrainedIMUGPSFuser

Pose estimation with nonholonomic constraints

Description

The `NHConstrainedIMUGPSFuser` object implements sensor fusion of inertial measurement unit (IMU) and GPS data to estimate pose in the NED 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 `NHConstrainedIMUGPSFuser` object uses an extended Kalman filter to estimate these quantities.

Creation

Create a `NHConstrainedIMUGPSFuser` with nonholonomic constraints using `insfilter`:

```
filt = insfilter('NonholonomicHeading', true, 'Magnetometer', false);
```

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)²
`[4.8e-6 4.8e-6 4.8e-6]` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope in $(rad/s)^2$, 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)²
`[4e-14 4e-14 4e-14]` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the gyroscope bias in $(rad/s)^2$, 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 $(m/s^2)^2$
`[4.8e-2 4.8e-2 4.8e-2]` (default) | scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer in $(m/s^2)^2$, 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 $(m/s^2)^2$
`[4e-14 4e-14 4e-14]` (default) | positive scalar | 3-element row vector

Multiplicative process noise variance from the accelerometer bias in $(m/s^2)^2$, 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)	m	8:10
Velocity (NED)	m/s	11:13
Accelerometer Bias (XYZ)	m/s^2	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 $(m/s)^2$
`1e-2` (default) | nonnegative scalar

Velocity constraints noise in $(m/s)^2$, specified as a nonnegative scalar.

Data Types: `single` | `double`

Object Functions

correct	Correct states using direct state measurements
fusegps	Correct states using GPS data
pose	Current orientation and position estimate
predict	Update states using accelerometer and gyroscope data
reset	Reset internal states
stateinfo	Display state vector information

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','initialState
      'gyroData','gpsFs','gpsLLA','Rpos','gpsVel','Rvel','trueOrient','truePos');
```

Initialize the `NHConstrainedIMUGPSFuser` filter object.

```
filt = insfilter('NonholonomicHeading',true,'Magnetometer',false);
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),
```

```
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)
```

Extended Capabilities

C/C++ Code Generation

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

See Also

`insfilter`

Topics

“Estimate Position and Orientation of a Ground Vehicle”

Introduced in R2018b

accelparams class

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`.

Construction

`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^2)
`inf` (default) | real positive scalar

Maximum sensor reading in m/s^2, specified as a real positive scalar.

Data Types: `single` | `double`

`Resolution` — Resolution of sensor measurements ($(m/s^2)/LSB$)
`0` (default) | real nonnegative scalar

Resolution of sensor measurements in $(m/s^2)/LSB$, specified as a real nonnegative scalar.

Data Types: `single` | `double`

`ConstantBias` — Constant sensor offset bias (m/s^2)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Constant sensor offset bias in 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`

`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 ($m/s^2/\sqrt{Hz}$)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Power spectral density of sensor noise in ($m/s^2/\sqrt{Hz}$), 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^2)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Instability of the bias offset in 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`

`RandomWalk` — Integrated white noise of sensor ($(m/s^2)(\sqrt{Hz})$)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Integrated white noise of sensor in ($(m/s^2)(\sqrt{Hz})$), 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/s²)/°C)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Sensor bias from temperature in $(m/s^2)/°C$, 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 m/s^2 and a resolution of 0.598 $\left(mm/s^2\right)$/LSB. The constant offset bias is 0.49 m/s^2. The sensor has a power spectral density of 3920 $\left(\mu m/s^2\right)/\sqrt{Hz}$. The bias from temperature is 0.294 $\left(m/s^2\right)/^0C$. The scale factor error from temperature is $0.02\%/^0C$. The sensor axes are skewed by 2%.

```
params = accelparams('MeasurementRange',19.6,'Resolution',0.598e-3,'ConstantBias',0.49
```

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;
```

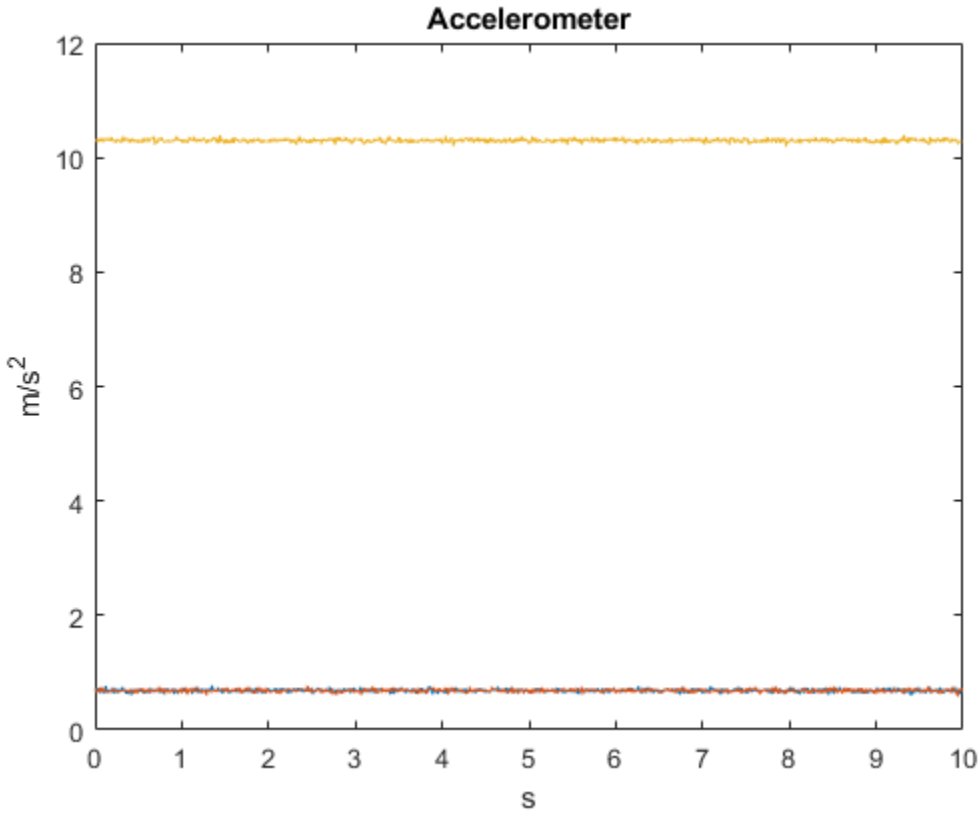
```
t = 0:1/Fs:(numSamples-1)/Fs;

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

System Objects
`imuSensor`

Classes
`gyroparams` | `magparams`

Topics

“Model IMU, GPS, and INS/GPS”

Introduced in R2018b

gyroparams class

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`.

Construction

`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)/℃), 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 (%/℃), 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/s²)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Sensor bias from linear acceleration in (rad/s)/(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`

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 (rad/s)/LSB. The constant offset bias is 0.349 rad/s. The sensor has a power spectral density of 8.727e-4 \rmrad/s/\sqrt \rmHz . *Thebiasfromtemperatureis*0 . 349\rmrad/s/. The bias from temperature is 0.349 $\left(\text{rad/s}^2\right)/^0\text{C}$. The scale factor error from temperature is 0.2%/^{0}C. The

sensor axes are skewed by 2%. The sensor bias from linear acceleration is 0.178e-3 (rad/s)/(m/s^2)

```
params = gyroparams('MeasurementRange',4.363,'Resolution',1.332e-04,'ConstantBias',0.3
```

Use a sample rate of 100 Hz spaced out over 1000 samples. Create the imuSensor object using the gyroscope parameter object.

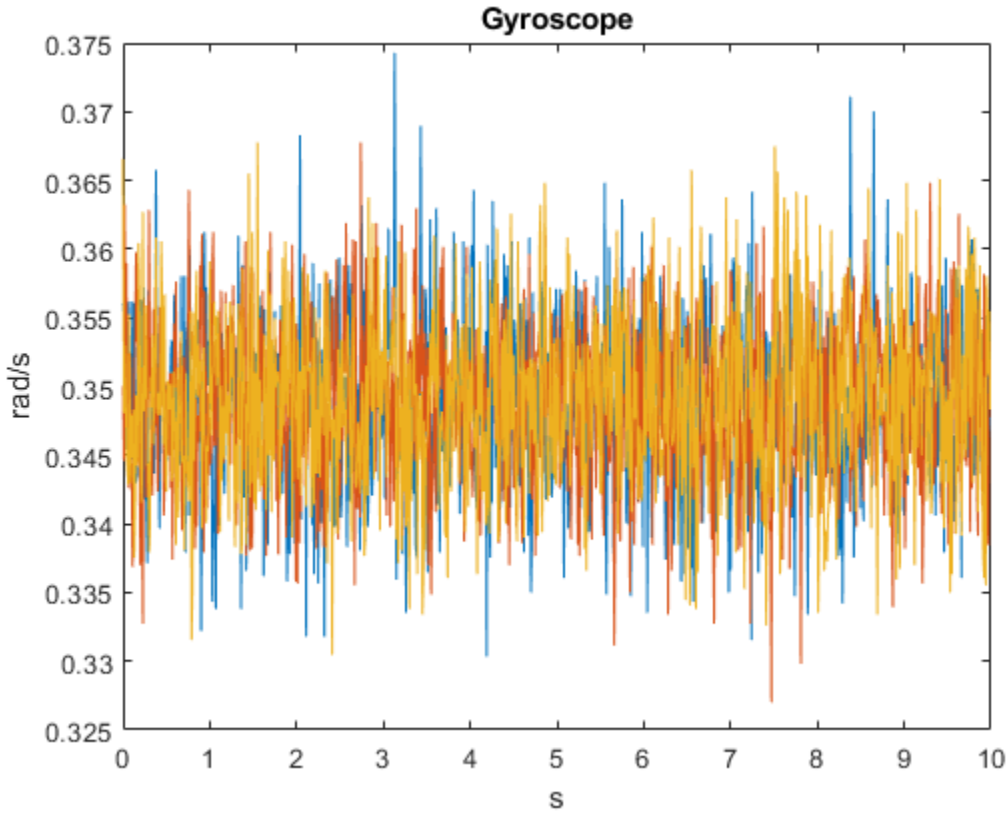
```
Fs = 100;
numSamples = 1000;
t = 0:1/Fs:(numSamples-1)/Fs;

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

Classes
`accelparams` | `magparams`

System Objects
`imuSensor`

Topics
“Model IMU, GPS, and INS/GPS”

Introduced in R2018b

magparams class

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`.

Construction

`params = magarams` 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 (μT)
`Inf` (default) | real positive scalar

Maximum sensor reading in μT, specified as a real positive scalar.

Data Types: `single` | `double`

`Resolution` — 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`

`ConstantBias` — 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`

`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 (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`

`BiasInstability` — Instability of the bias offset (μT)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Instability of the bias offset 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`

`RandomWalk` — Integrated white noise of sensor (μT/√Hz)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Integrated white noise of sensor 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`

`TemperatureBias` — Sensor bias from temperature (μT/°C)
`[0 0 0]` (default) | real scalar | real 3-element row vector

Sensor bias from temperature in (μT/°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`

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 μT and a resolution of 0.1 μT/LSB. The constant offset bias is 1 μT. The sensor has a power spectral density of $\left(\frac{[0.6\ 0.6\ 0.9]}{\sqrt{100}}\right)$ μT/$\sqrt{\text{Hz}}$. The bias from temperature is [0.8 0.8 2.4] μT/^{0}C. The scale factor error from temperature is 0.1 %/^{0}C.

```
params = magparams('MeasurementRange',1200,'Resolution',0.1,'ConstantBias',1,'NoiseDens
```

Use a sample rate of 100 Hz spaced out over 1000 samples. Create the imuSensor object using the magnetometer parameter object.

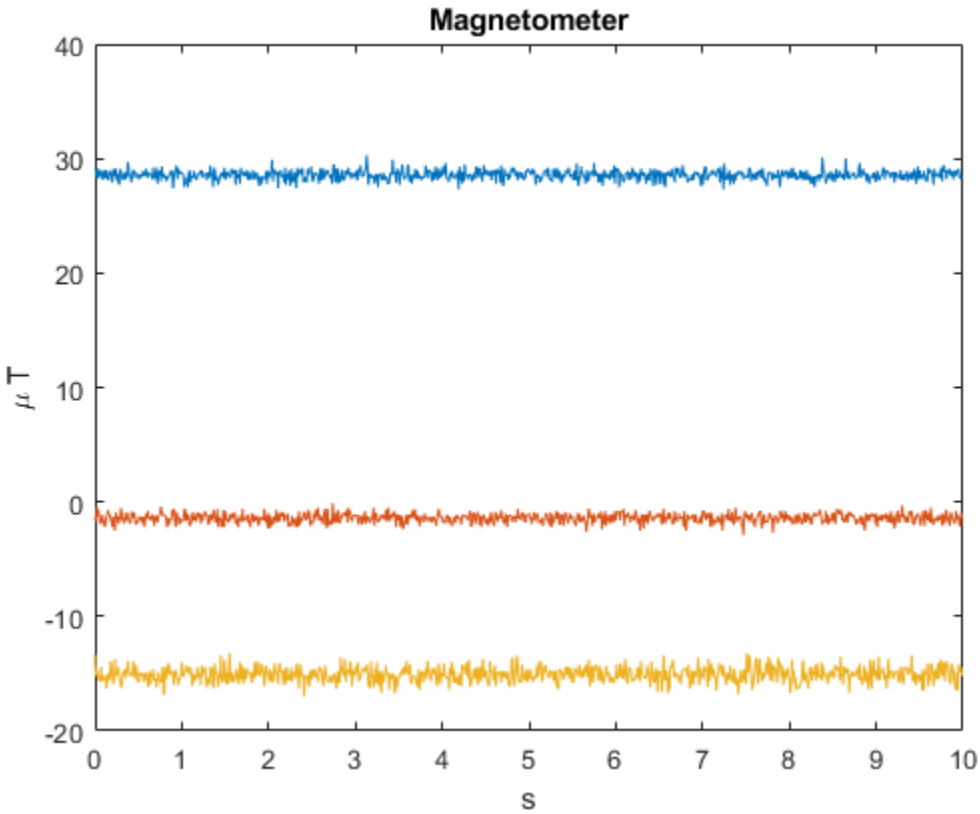
```
Fs = 100;
numSamples = 1000;
t = 0:1/Fs:(numSamples-1)/Fs;

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

Classes

`accelparams` | `gyroparams`

System Objects

`imuSensor`

Topics

“Model IMU, GPS, and INS/GPS”

Introduced in R2018b

objectDetection class

Create object detection report

Description

The `objectDetection` class creates and reports detections of objects in a tracking scenario. Each report contains information obtained by a sensor for a single object. You can use the `objectDetection` output as the input to trackers such as `trackerGNN` or `trackerTOMHT`.

Construction

`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 type of measurement. For example, a measurement of the Cartesian coordinates implies that $N = 3$. A measurement of spherical coordinates and range rate implies that $N = 4$. This argument sets the `Measurement` property.

Output Arguments

`detection` — Detection report

`objectDetection` class object

Detection report, 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
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters
`ObjectAttributes`	Additional information passed to tracker

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.

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.

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, `trackerGNN` or `trackerTOMHT` immediately create a confirmed track from the detection.

Example: `1`

Data Types: `double`

`MeasurementParameters` — Measurement function parameters

`{}` (default) | cell array

Measurement function parameters, specified as a cell array. The cell array contains all the arguments used by the measurement function specified by the `MeasurementFcn` property of a nonlinear tracking filter such as `trackingEKF` or `trackingUKF`. Each cell contains a single argument.

Example: `{[1;0;0],'rectangular'}`

`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 `trackerGNN` and `trackerTOMHT` 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

Classes

`trackingABF` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

System Objects

`irSensor` | `monostaticRadarSensor` | `radarSensor` | `sonarSensor` | `trackerGNN` | `trackerTOMHT`

Introduced in R2018b

quaternion

Create a quaternion array

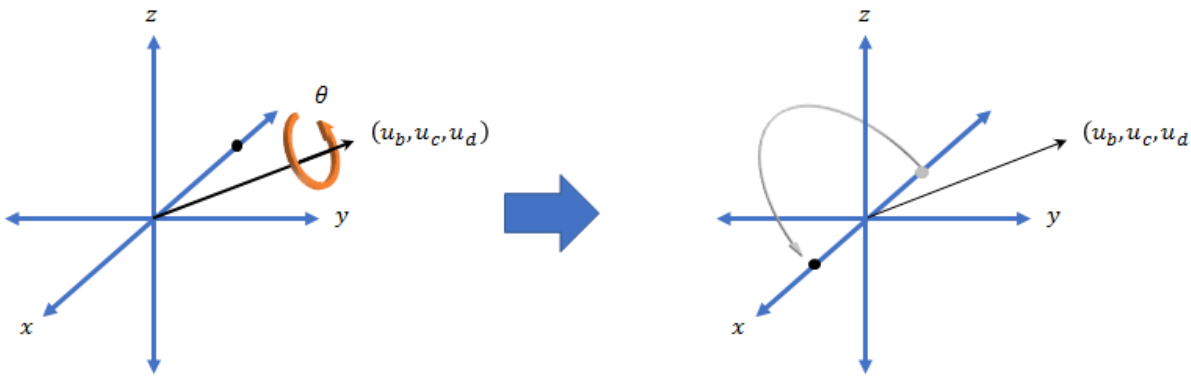
Description

A quaternion is a four-part hyper-complex number used in three-dimensional rotations and orientations.

A quaternion number is represented in the form $a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, where a, b, c, and d parts are real numbers, and i, j, and k are the basis elements, satisfying the equation: $\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1$.

The set of quaternions, denoted by **H**, is defined within a four-dimensional vector space over the real numbers, $\mathbf{R}^4$. Every element of **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 $q = \cos\left(\theta/2\right) + \sin\left(\theta/2\right)(u_b\mathrm{i} + u_c\mathrm{j} + u_d\mathrm{k})$, where θ is the angle of rotation and [u_b, u_c, and u_d] is the axis of rotation.

Creation

Syntax

```
quat = quaternion()
quat = quaternion(A,B,C,D)
quat = quaternion(matrix)
quat = quaternion(RV,'rotvec')
quat = quaternion(RV,'rotvecd')
quat = quaternion(RM,'rotmat',PF)
quat = quaternion(E,'euler',RS,PF)
quat = quaternion(E,'eulerd',RS,PF)
```

Description

`quat = quaternion()` creates an empty quaternion.

`quat = quaternion(A,B,C,D)` creates a quaternion array where the four quaternion parts are taken from the arrays `A`, `B`, `C`, and `D`. All the inputs must have the same size and be of the same data type.

`quat = quaternion(matrix)` creates an *N*-by-1 quaternion array from an *N*-by-4 matrix, where each column becomes one part of the quaternion.

`quat = quaternion(RV,'rotvec')` creates an *N*-by-1 quaternion array from an *N*-by-3 matrix of rotation vectors, `RV`. Each row of `RV` represents a rotation vector in radians.

`quat = quaternion(RV,'rotvecd')` creates an *N*-by-1 quaternion array from an *N*-by-3 matrix of rotation vectors, `RV`. Each row of `RV` represents a rotation vector in degrees.

`quat = quaternion(RM,'rotmat',PF)` creates an *N*-by-1 quaternion array from the 3-by-3-by-*N* array of rotation matrices, `RM`. `PF` can be either `'point'` if the Euler angles represent point rotations or `'frame'` for frame rotations.

`quat = quaternion(E,'euler',RS,PF)` creates an *N*-by-1 quaternion array from the *N*-by-3 matrix, `E`. Each row of `E` represents a set of Euler angles in radians. The angles in `E` are rotations about the axes in sequence `RS`.

`quat = quaternion(E,'eulerd',RS,PF)` creates an *N*-by-1 quaternion array from the *N*-by-3 matrix, `E`. Each row of `E` represents a set of Euler angles in degrees. The angles in `E` are rotations about the axes in sequence `RS`.

Input Arguments

`A,B,C,D` — Quaternion parts

comma-separated arrays of the same size

Parts of a quaternion, specified as four comma-separated scalars, matrices, or multi-dimensional arrays of the same size.

Example: `quat = quaternion(1,2,3,4)` creates a quaternion of the form 1 + 2i + 3j + 4k.

Example: `quat = quaternion([1,5],[2,6],[3,7],[4,8])` creates a 1-by-2 quaternion array where `quat(1,1) = 1 + 2i + 3j + 4k` and `quat(1,2) = 5 + 6i + 7j + 8k`

Data Types: `single` | `double`

`matrix` — Matrix of quaternion parts

N-by-4 matrix

Matrix of quaternion parts, specified as an *N*-by-4 matrix. Each row represents a separate quaternion. Each column represents a separate quaternion part.

Example: `quat = quaternion(rand(10,4))` creates a 10-by-1 quaternion array.

Data Types: `single` | `double`

`RV` — Matrix of rotation vectors

N-by-3 matrix

Matrix of rotation vectors, specified as an *N*-by-3 matrix. Each row of `RV` represents the [X Y Z] elements of a rotation vector. A rotation vector is a unit vector representing the axis of rotation scaled by the angle of rotation in radians or degrees.

To use this syntax, specify the first argument as a matrix of rotation vectors and the second argument as the `'rotvec'` or `'rotvecd'`.

Example: `quat = quaternion(rand(10,3),'rotvec')` creates a 10-by-1 quaternion array.

Data Types: `single` | `double`

`RM` — Rotation matrices
3-by-3 matrix | 3-by-3-by-*N* array

Array of rotation matrices, specified by a 3-by-3 matrix or 3-by-3-by-*N* array. Each page of the array represents a separate rotation matrix.

Example: `quat = quaternion(rand(3),'rotmat','point')`

Example: `quat = quaternion(rand(3),'rotmat','frame')`

Data Types: `single` | `double`

`PF` — Type of rotation matrix
`'point'` | `'frame'`

Type of rotation matrix, specified by `'point'` or `'frame'`.

Example: `quat = quaternion(rand(3),'rotmat','point')`

Example: `quat = quaternion(rand(3),'rotmat','frame')`

Data Types: `char` | `string`

`E` — Matrix of Euler angles
N-by-3 matrix

Matrix of Euler angles, specified by an *N*-by-3 matrix. If using the `'euler'` syntax, specify `E` in radians. If using the `'eulerd'` syntax, specify `E` in degrees.

Example: `quat = quaternion(E,'euler','YZY','point')`

Example: `quat = quaternion(E,'euler','XYZ','frame')`

Data Types: `single` | `double`

RS — Rotation sequence
character vector | scalar string

Rotation sequence, specified as a three-element character vector:

- 'YZY'
- 'YXY'
- 'ZYZ'
- 'ZXZ'
- '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];
```

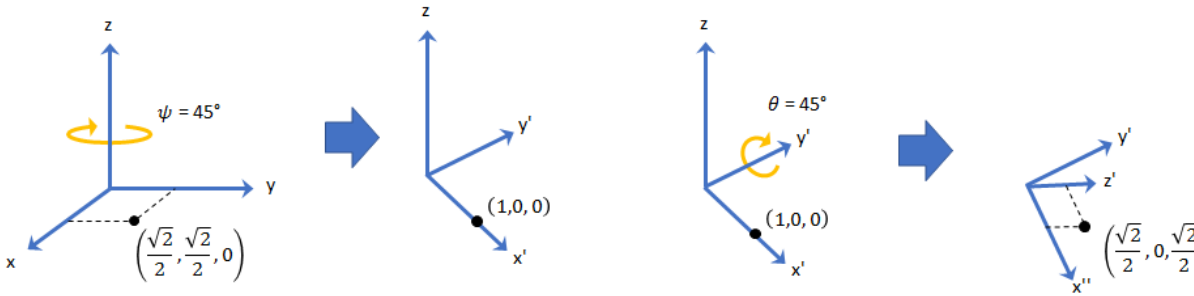
In this representation, the first column represents the *x*-axis, the second column represents the *y*-axis, and the third column represents the *z*-axis.

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)
  ```

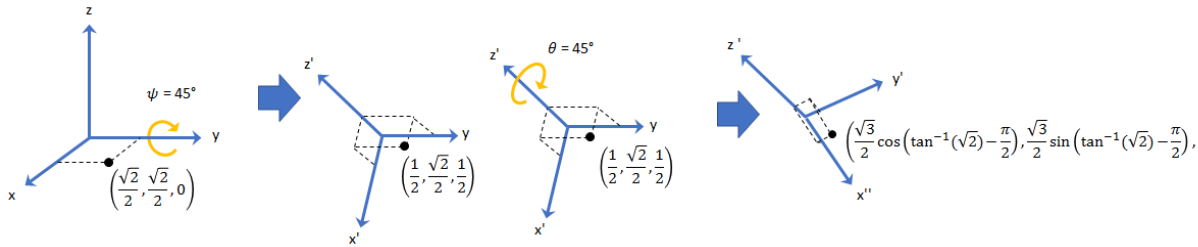
  ```
  newPointCoordinate =

      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

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)
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
prod	Product of a quaternion array
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
ldivide, .\	Element-wise quaternion left division
rdivide, ./	Element-wise quaternion right division
power, .^	Element-wise quaternion power
exp	Exponential of quaternion array
log	Natural logarithm of quaternion array
transpose	Transpose a quaternion array
uminus, -	Quaternion unary minus
zeros	Create quaternion array with all parts set to zero
randrot	Uniformly distributed random rotations

Examples

Create Empty Quaternion

```
quat = quaternion()
```

```
quat =

  0x0 empty quaternion array
```

By default, the underlying class of the quaternion is a double.

```
classUnderlying(quat)
```

```
ans =
'double'
```

Create Quaternion by Specifying Individual Quaternion Parts

You can create a quaternion array by specifying the four parts as comma-separated scalars, matrices, or multidimensional arrays of the same size.

Define quaternion parts as scalars.

```
A = 1.1;
B = 2.1;
C = 3.1;
D = 4.1;
quatScalar = quaternion(A,B,C,D)
```

```
quatScalar = quaternion
     1.1 + 2.1i + 3.1j + 4.1k
```

Define quaternion parts as column vectors.

```
A = [1.1;1.2];
B = [2.1;2.2];
C = [3.1;3.2];
D = [4.1;4.2];
quatVector = quaternion(A,B,C,D)
```

```
quatVector = 2x1 quaternion array
     1.1 + 2.1i + 3.1j + 4.1k
     1.2 + 2.2i + 3.2j + 4.2k
```

Define quaternion parts as matrices.

```
A = [1.1,1.3; ...
     1.2,1.4];
B = [2.1,2.3; ...
     2.2,2.4];
C = [3.1,3.3; ...
     3.2,3.4];
D = [4.1,4.3; ...
```

```
     4.2,4.4];
quatMatrix = quaternion(A,B,C,D)
```

```
quatMatrix = 2x2 quaternion array
     1.1 + 2.1i + 3.1j + 4.1k     1.3 + 2.3i + 3.3j + 4.3k
     1.2 + 2.2i + 3.2j + 4.2k     1.4 + 2.4i + 3.4j + 4.4k
```

Define quaternion parts as three dimensional arrays.

```
A = randn(2,2,2);
B = zeros(2,2,2);
C = zeros(2,2,2);
D = zeros(2,2,2);
quatMultiDimArray = quaternion(A,B,C,D)
```

```
quatMultiDimArray = 2x2x2 quaternion array
quatMultiDimArray(:,:,1) =

     0.53767 +       0i +       0j +       0k     -2.2588 +       0i +       0j +
      1.8339 +       0i +       0j +       0k     0.86217 +       0i +       0j +


quatMultiDimArray(:,:,2) =

     0.31877 +       0i +       0j +       0k    -0.43359 +       0i +       0j +
     -1.3077 +       0i +       0j +       0k     0.34262 +       0i +       0j +
```

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 = 3×4

    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 = 3×4

    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706
```

Create Quaternion by Specifying Rotation Vectors

You can create an *N*-by-1 quaternion array by specifying an *N*-by-3 matrix of rotation vectors in radians or degrees. Rotation vectors are compact spatial representations that have a one-to-one relationship with normalized quaternions.

Rotation Vectors in Radians

Create a scalar quaternion using a rotation vector and verify the resulting quaternion is normalized.

```
rotationVector = [0.3491,0.6283,0.3491];
quat = quaternion(rotationVector,'rotvec')
```

```
quat = quaternion
     0.92124 + 0.16994i + 0.30586j + 0.16994k
```

```
norm(quat)
```

```
ans = 1.0000
```

You can convert from quaternions to rotation vectors in radians using the `rotvec` function. Recover the `rotationVector` from the quaternion, `quat`.

```
rotvec(quat)
```

```
ans = 1×3

    0.3491    0.6283    0.3491
```

Rotation Vectors in Degrees

Create a scalar quaternion using a rotation vector and verify the resulting quaternion is normalized.

```
rotationVector = [20,36,20];
quat = quaternion(rotationVector,'rotvecd')
```

```
quat = quaternion
     0.92125 + 0.16993i + 0.30587j + 0.16993k
```

```
norm(quat)
```

```
ans = 1
```

You can convert from quaternions to rotation vectors in degrees using the `rotvecd` function. Recover the `rotationVector` from the quaternion, `quat`.

```
rotvecd(quat)
```

```
ans = 1×3

   20.0000   36.0000   20.0000
```

Create Quaternion by Specifying Rotation Matrices

You can create an N-by-1 quaternion array by specifying a 3-by-3-by-N array of rotation matrices. Each page of the rotation matrix array corresponds to one element of the quaternion array.

Create a scalar quaternion using a 3-by-3 rotation matrix. Specify whether the rotation matrix should be interpreted as a frame or point rotation.

```
rotationMatrix = [1 0         0; ...
                  0 sqrt(3)/2 0.5; ...
                  0 -0.5      sqrt(3)/2];
quat = quaternion(rotationMatrix,'rotmat','frame')
```

```
quat = quaternion
     0.96593 + 0.25882i +       0j +       0k
```

You can convert from quaternions to rotation matrices using the `rotmat` function. Recover the `rotationMatrix` from the quaternion, `quat`.

```
rotmat(quat,'frame')
```

```
ans = 3×3

    1.0000         0         0
         0    0.8660    0.5000
         0   -0.5000    0.8660
```

Create Quaternion by Specifying Euler Angles

You can create an *N*-by-1 quaternion array by specifying an *N*-by-3 array of Euler angles in radians or degrees.

Euler Angles in Radians

Use the `euler` syntax to create a scalar quaternion using a 1-by-3 vector of Euler angles in radians. Specify the rotation sequence of the Euler angles and whether the angles represent a frame or point rotation.

```
E = [pi/2,0,pi/4];
quat = quaternion(E,'euler','ZYX','frame')
```

```
quat = quaternion
     0.65328 +  0.2706i +  0.2706j + 0.65328k
```

You can convert from quaternions to Euler angles using the `euler` function. Recover the Euler angles, `E`, from the quaternion, `quat`.

```
euler(quat,'ZYX','frame')
```

```
ans = 1×3

    1.5708         0    0.7854
```

Euler Angles in Degrees

Use the `eulerd` syntax to create a scalar quaternion using a 1-by-3 vector of Euler angles in degrees. Specify the rotation sequence of the Euler angles and whether the angles represent a frame or point rotation.

```
E = [90,0,45];
quat = quaternion(E,'eulerd','ZYX','frame')
```

```
quat = quaternion
     0.65328 +  0.2706i +  0.2706j + 0.65328k
```

You can convert from quaternions to Euler angles in degrees using the `eulerd` function. Recover the Euler angles, `E`, from the quaternion, `quat`.

```
eulerd(quat,'ZYX','frame')
```

```
ans = 1×3

   90.0000         0   45.0000
```

Quaternion Algebra

Quaternions form a noncommutative associative algebra over the real numbers. This example illustrates the rules of quaternion algebra.

Addition and Subtraction

Quaternion addition and subtraction occur part-by-part, and are commutative:

```
Q1 = quaternion(1,2,3,4)
```

```
Q1 = quaternion
     1 + 2i + 3j + 4k
```

```
Q2 = quaternion(9,8,7,6)

Q2 = quaternion
     9 + 8i + 7j + 6k


Q1plusQ2 = Q1 + Q2

Q1plusQ2 = quaternion
     10 + 10i + 10j + 10k


Q2plusQ1 = Q2 + Q1

Q2plusQ1 = quaternion
     10 + 10i + 10j + 10k


Q1minusQ2 = Q1 - Q2

Q1minusQ2 = quaternion
    -8 - 6i - 4j - 2k


Q2minusQ1 = Q2 - Q1

Q2minusQ1 = quaternion
     8 + 6i + 4j + 2k
```

You can also perform addition and subtraction of real numbers and quaternions. The first part of a quaternion is referred to as the *real* part, while the second, third, and fourth parts are referred to as the *vector*. Addition and subtraction with real numbers affect only the real part of the quaternion.

```
Q1plusRealNumber = Q1 + 5

Q1plusRealNumber = quaternion
     6 + 2i + 3j + 4k


Q1minusRealNumber = Q1 - 5

Q1minusRealNumber = quaternion
    -4 + 2i + 3j + 4k
```

Multiplication

Quaternion multiplication is determined by the products of the basis elements and the distributive law. Recall that multiplication of the basis elements, *i*, *j*, and *k*, are not commutative, and therefore quaternion multiplication is not commutative.

```
Q1timesQ2 = Q1 * Q2
```

```
Q1timesQ2 = quaternion
    -52 + 16i + 54j + 32k
```

```
Q2timesQ1 = Q2 * Q1
```

```
Q2timesQ1 = quaternion
    -52 + 36i + 14j + 52k
```

```
isequal(Q1timesQ2,Q2timesQ1)
```

```
ans = logical
   0
```

You can also multiply a quaternion by a real number. If you multiply a quaternion by a real number, each part of the quaternion is multiplied by the real number individually:

```
Q1times5 = Q1*5
```

```
Q1times5 = quaternion
      5 + 10i + 15j + 20k
```

Multiplying a quaternion by a real number is commutative.

```
isequal(Q1*5,5*Q1)
```

```
ans = logical
   1
```

Conjugation

The complex conjugate of a quaternion is defined such that each element of the vector portion of the quaternion is negated.

```
Q1
```

```
Q1 = quaternion
     1 + 2i + 3j + 4k
```

```
conj(Q1)
```

```
ans = quaternion
     1 - 2i - 3j - 4k
```

Multiplication between a quaternion and its conjugate is commutative:

```
isequal(Q1*conj(Q1),conj(Q1)*Q1)
```

```
ans = logical
   1
```

Quaternion Array Manipulation

You can organize quaternions into vectors, matrices, and multidimensional arrays. Built-in MATLAB® functions have been enhanced to work with quaternions.

Concatenate

Quaternions are treated as individual objects during concatenation and follow MATLAB rules for array manipulation.

```
Q1 = quaternion(1,2,3,4);
Q2 = quaternion(9,8,7,6);

qVector = [Q1,Q2]
```

```
qVector = 1x2 quaternion array
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
```

```
Q3 = quaternion(-1,-2,-3,-4);
Q4 = quaternion(-9,-8,-7,-6);

qMatrix = [qVector;Q3,Q4]
```

```
qMatrix = 2x2 quaternion array
     1 + 2i + 3j + 4k     9 + 8i + 7j + 6k
    -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 R2018a

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` or `waypointTrajectory` 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.

`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

`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

advance	Advance tracking scenario simulation by one time step
platform	Add platform to tracking scenario
platformPoses	Positions, velocities, and orientations of all platforms in tracking scenario
platformProfiles	Profiles of platforms in tracking scenario
restart	Restart tracking scenario simulation
record	Run tracking scenario and record platform states

Examples

Create Tracking Scenario with Two Platforms

Construct a tracking scenario with two platforms following 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);
```

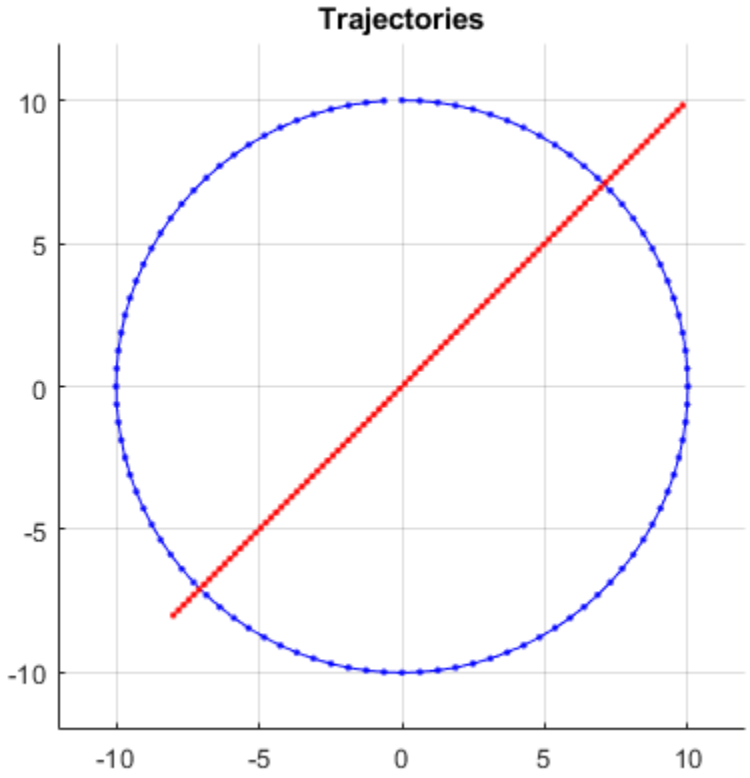
Verify the number of platforms in the scenario.

```
disp(sc.Platforms)
```

```
    [1×1 fusion.scenario.Platform]    [1×1 fusion.scenario.Platform]
```

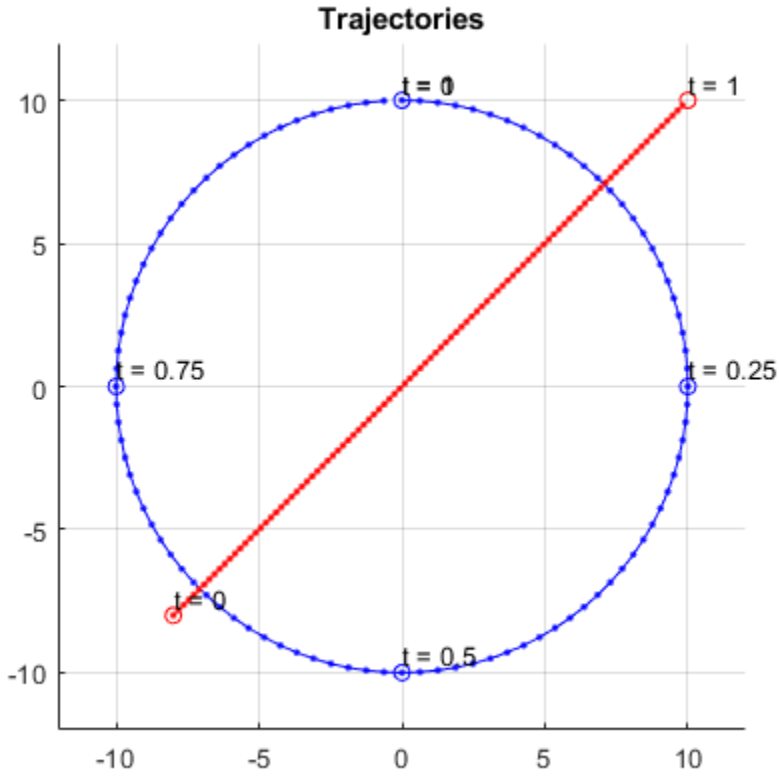
Run the simulation and plot the current position of each platform. Use an animated line to plot the position of each platform

```
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','Vertica
plot(wpts2(:,1),wpts2(:,2),' or')
text(wpts2(:,1),wpts2(:,2),"t = " + string(time2),'HorizontalAlignment','left','Vertica
hold off
```



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.

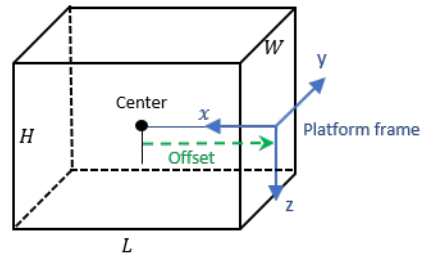
Example: `5`

Data Types: `double` | `single`

`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 with default property values (default) | `waypointTrajectory` object

Platform motion, specified as either a `kinematicTrajectory` object or a `waypointTrajectory` object.

The motion 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.

`Signatures` — Platform signatures
`rcsSignature` with default properties | cell array of signature objects

Platform signatures, specified as a cell array of `irSignature`, `rcsSignature`, and `tsSignature` objects. 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`.

Object Functions

detect	Detect signals using platform-mounted sensors
emit	Radiate signals from emitters mounted on platform

pose	Pose of platform
targetPoses	Target positions and orientations 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
    '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 = [623.561925 626.638509 0.000000 ] Velocity = [11.378941 -9
Time = 100.000000 Position = [1000.000000 0.000000 0.000000 ] Velocity = [1.677836 -15
Time = 150.000000 Position = [724.510494 -677.241693 0.000000 ] Velocity = [-11.268303
Time = 200.000000 Position = [0.000000 -1000.000000 0.000000 ] Velocity = [-16.843065
Time = 250.000000 Position = [-926.594496 -797.673675 0.000000 ] Velocity = [-15.11734
Time = 300.000000 Position = [-1000.000000 0.000000 0.000000 ] Velocity = [9.520786 9.6
Time = 350.000000 Position = [-421.669352 137.708709 0.000000 ] Velocity = [10.727588
Time = 400.000000 Position = [-0.000000 1.000000 0.000000 ] Velocity = [6.118648 -2.38
```

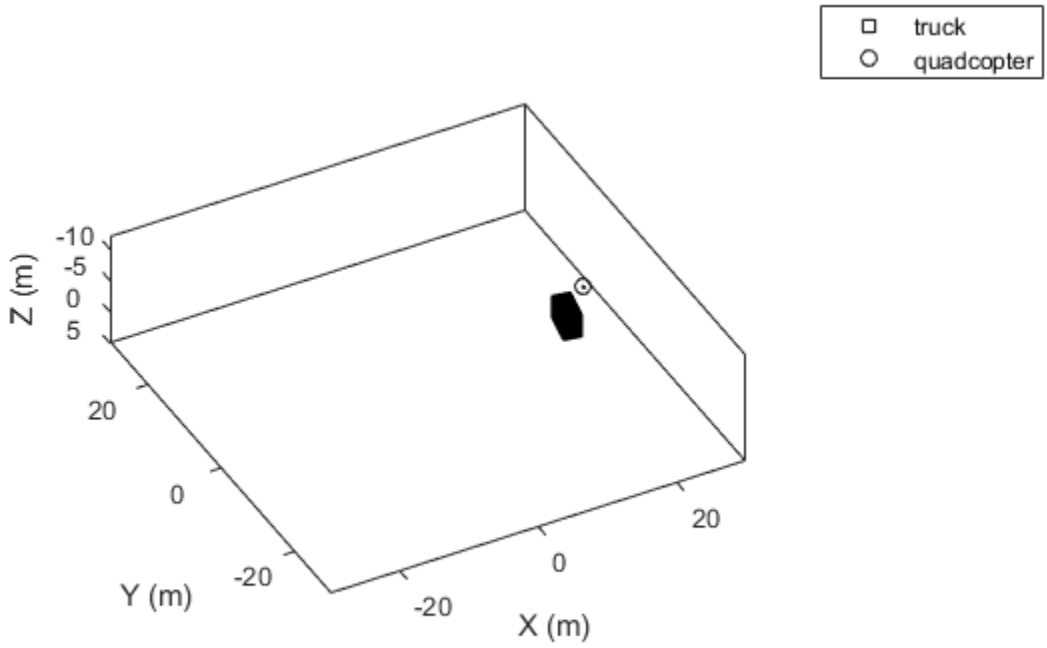
Cuboid Platforms Follow Circular Trajectories

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

Classes
`rcsSignature` | `tsSignature`

System Objects
`insSensor` | `irSensor` | `kinematicTrajectory` | `monotstaticRadarSensor` | `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
`EmitterIndex`	Unique emitter index
`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 `UpdateInterval`.
`IsScanDone`	`IsScanDone` is `true` when the emitter has completed a scan.
`FieldOfView`	Field of view of emitter.
`MeasurementParameters`	`MeasurementParameters` is an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.

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
`EmitterIndex`	Unique emitter index
`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 `UpdateInterval`.
`IsScanDone`	`IsScanDone` is `true` when the emitter has completed a scan.
`FieldOfView`	Field of view of emitter.

`MeasurementParameters`	`MeasurementParameters` is an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.

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
`SensorIndex`	Unique sensor index
`IsValidTime`	Valid detection time, returned as `0` or `1`. `IsValidTime` is `0` when detection updates are requested at times that are between update intervals specified by `UpdateInterval`.
`IsScanDone`	`IsScanDone` is `true` when the sensor has completed a scan.
`FieldOfView`	Field of view of sensor determines which objects fall within the sensor beam during object execution. The field of view is defined as a 2-by-1 vector of positive real values, [azfov;elfov].

`MeasurementParameters`	`MeasurementParameters` is an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current sensor frame.

Data Types: `struct`

`numDets` — Number of detections
nonnegative integer

Number of detections reported, returned as a nonnegative integer.

Data Types: `double`

Definitions

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

<table>
<tr><th>DetectionCoordinates</th><th>Measurement and Measurement Noise Coordinates</th></tr>
<tr><td>'Scenario'
'Body'
'Sensor rectangular'</td><td>Coordinate Dependence on HasRangeRate<table><tr><th>HasRangeRate</th><th>Coordinates</th></tr><tr><td>true</td><td>[x; y; z; vx; vy; vz]</td></tr><tr><td>false</td><td>[x; y; z]</td></tr></table></td></tr>
<tr><td>'Sensor spherical'</td><td>Coordinate Dependence on HasRangeRate and HasElevation<table><tr><th>HasRangeRate</th><th>HasElevation</th><th>Coordinates</th></tr><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></table></td></tr>
</table>

Measurement Parameters

The `MeasurementParameters` field 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”). The longest possible sequence of transformations is Sensor → Platform → Scenario. For example, if the detections are reported in sensor spherical coordinates and `HasINS` is set to `false`, then the sequence consists of one transformation from sensor to platform. If `HasINS` is `true`, the sequence of transformations consists of two transformations – first to platform coordinates then to scenario coordinates. Trivially, if the detections are reported in platform rectangular coordinates and `HasINS` is set to `false`, the transformation consists only of the identity.

The structure fields 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 `'rectangular'`. When detections are reported in spherical coordinates, `Frame` is set `'spherical'` for the first `struct`.
`OriginPosition`	Position offset of the origin of the frame relative to the parent frame, represented as a 3-by-1 vector.
`OriginVelocity`	Velocity offset of the origin of the frame relative to the parent frame, represented as a 3-by-1 vector.
`Orientation`	3-by-3 real-valued orthonormal frame rotation matrix.
`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 indication if azimuth is included in the measurement.
`HasRange`	A logical scalar indication 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.
`SNR`	Detection signal-to-noise ratio in dB.

See Also

`emit` | `pose` | `targetPoses`

Introduced in R2018b

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]`.
`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

Platform.pose

Pose of platform

Syntax

```
pse = pose(ptfm,type)
```

Description

`pse = pose(ptfm,type)` returns the 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.

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`

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 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 scenario 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 platform in scenario coordinates, specified as a real-valued 1-by-3 vector. Units are in 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]`.

Field	Description
`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. The format is specified by the `fmt` input argument. Orientation defines the frame rotation from the local NED 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 in degrees per second. The default is value `[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.

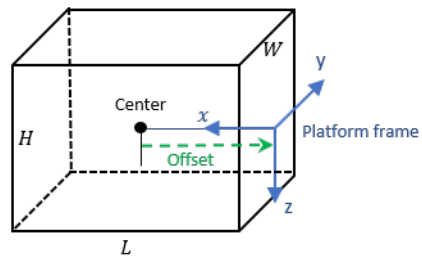
Example: 5

Data Types: `double`

`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 with default property values (default) | `waypointTrajectory` object

Platform motion, specified as either a `kinematicTrajectory` object or a `waypointTrajectory` object.

The motion 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.

`Signatures` — Platform signatures
`rcsSignature` with default properties | cell array of signature objects

Platform signatures, specified as a cell array of `rcsSignature`, `irSignature`, or `tsSignature` objects. A signature represents the reflection or emission pattern of a platform, such as its radar cross-section, target strength, or IR emission.

`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.

Object Functions

detect	Detect signals using platform-mounted sensors
emit	Radiate signals from emitters mounted on platform
pose	Pose of platform
targetPoses	Target positions and orientations 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
    '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 = [623.561925 626.638509 0.000000 ] Velocity = [11.378941 -9
Time = 100.000000 Position = [1000.000000 0.000000 0.000000 ] Velocity = [1.677836 -15
Time = 150.000000 Position = [724.510494 -677.241693 0.000000 ] Velocity = [-11.268303
Time = 200.000000 Position = [0.000000 -1000.000000 0.000000 ] Velocity = [-16.843065
Time = 250.000000 Position = [-926.594496 -797.673675 0.000000 ] Velocity = [-15.11734
Time = 300.000000 Position = [-1000.000000 0.000000 0.000000 ] Velocity = [9.520786 9.6
Time = 350.000000 Position = [-421.669352 137.708709 0.000000 ] Velocity = [10.727588
Time = 400.000000 Position = [-0.000000 1.000000 0.000000 ] Velocity = [6.118648 -2.380
```

Cuboid Platforms Follow Circular Trajectories

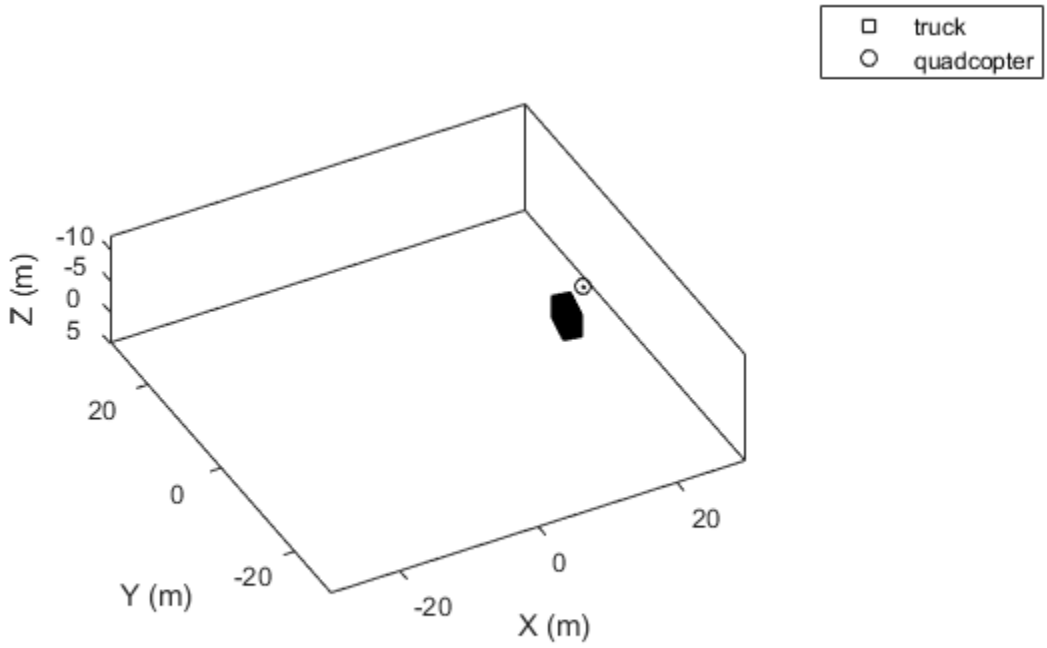
Create a tracking scenario with two cuboid platforms following circular trajectories.

```
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

platformPoses

Positions, velocities, and orientations of all platforms in tracking scenario

Syntax

```
poses = platformPoses(sc)
poses = platformPoses(sc,fmt)
```

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.

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`

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 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 scenario 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 platform in scenario coordinates, specified as a real-valued 1-by-3 vector. Units are in 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]`.

Field	Description
`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. The format is specified by the `fmt` input argument. Orientation defines the frame rotation from the local NED 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 in degrees per second. The default is value `[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. 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

Field	Description
`ClassID`	User-defined platform classification identifier, defined as a nonnegative integer
`Signatures`	Platform signatures defined as a cell array of radar cross-section (`rcsSignature`), IR emission pattern (`irSignature`), and sonar target strength (`tsSignature`) objects.

See `Platform` for more completed definitions of the fields.

Introduced in R2018b

record

Run tracking scenario and record platform states

Syntax

```
rec = record(sc)
rec = record(sc,fmt)
```

Description

`rec = record(sc)` returns a record, `rec`, of the evolution of the tracking scenario simulation, `sc`. The `record` function starts the simulation from the beginning,

`rec = record(sc,fmt)` also specifies the format, `fmt`, of the returned platform orientation.

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`

Output Arguments

`rec` — Records of platform states during simulation

M-by-1 vector of structures

Records of platform states during the simulation, returned as an *M*-by-1 vector of structures. *M* is the number of time steps in the simulation. Each record corresponds to a simulation time step and contains the poses of all the platforms at that time. The record structure has these fields:

```
SimulationTime
Poses
```

The `SimulationTime` field contains the simulation time of the record. `Poses` is an *N*-by-1 vector of structures, where *N* is the number of platforms. Each `Poses` structure contains 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 scenario 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 platform in scenario coordinates, specified as a real-valued 1-by-3 vector. Units are in 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]`.

Field	Description
`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. The format is specified by the `fmt` input argument. Orientation defines the frame rotation from the local NED 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 in degrees per second. The default is value `[0 0 0]`.

Data Types: `struct`

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

rcsSignature class

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.

Construction

`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.

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`

Methods

value	Radar cross-section at specified angle and frequency

Examples

Radar Cross-Section of Ellipsoid

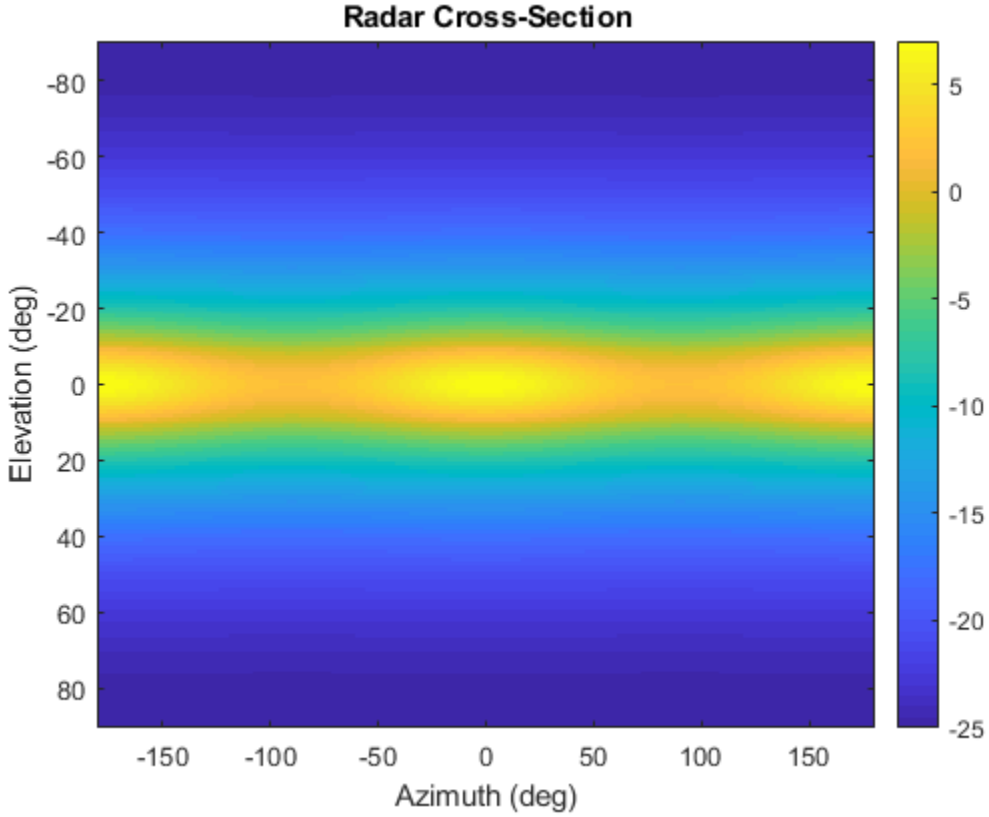
Specify the radar cross-section (RCS) of a triaxial ellipsoid and plot RCS values along an azimuth cut.

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.

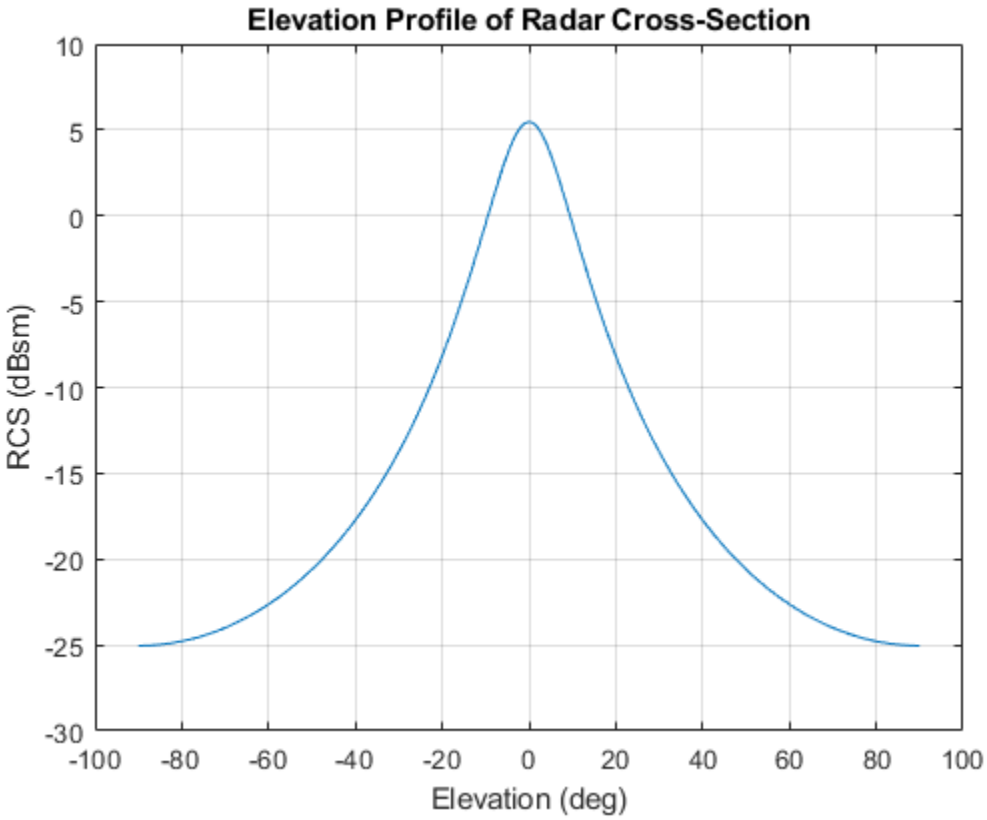
```
az = [-180:1:180];
el = [-90:1:90];
rcs = rcs_ellipsoid(a,b,c,az,el);
rcsdb = 10*log10(rcs);
imagesc(az,el,rcsdb)
title('Radar Cross-Section')
xlabel('Azimuth (deg)')
ylabel('Elevation (deg)')
colorbar
```



Create an `rcsSignature` object and plot an elevation cut at 30° azimuth.

```
rcssig = rcsSignature('Pattern',rcsdb,'Azimuth',az,'Elevation',el,'Frequency',[300e6 3(
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)
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

Class: rcsSignature

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`.

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

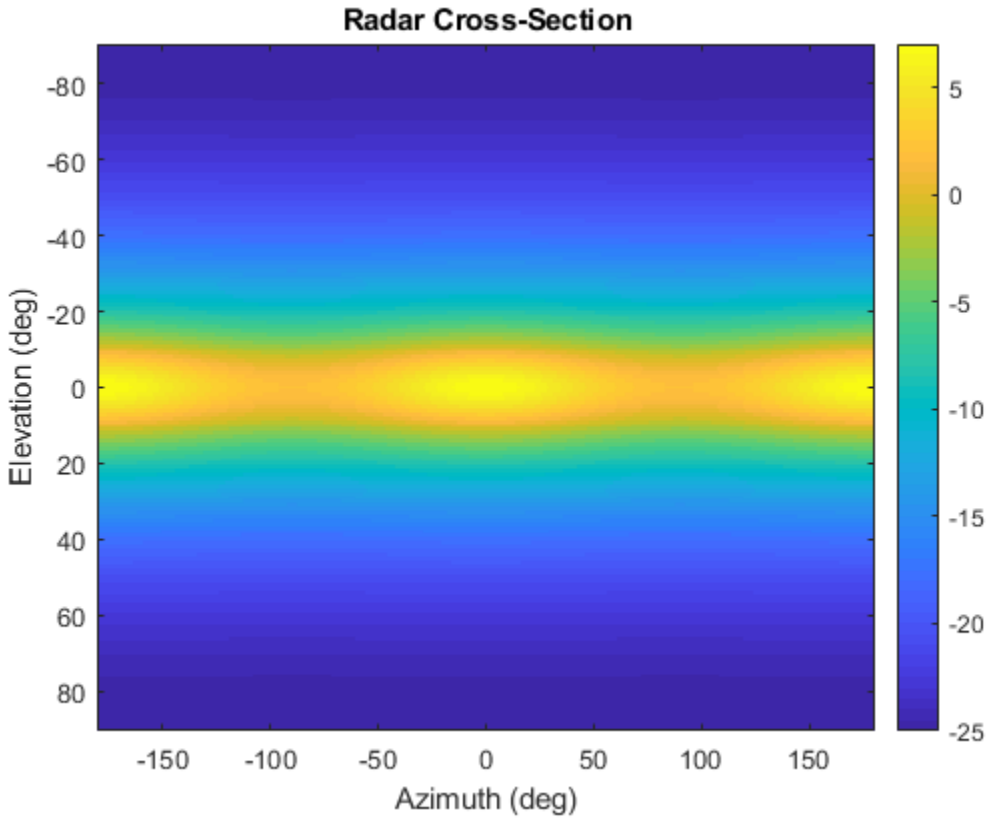
Specify the radar cross-section (RCS) of a triaxial ellipsoid and plot RCS values along an azimuth cut.

Specify the lengths of the axes of the ellipsoid. Units are in meters.

```
a = 0.15;
b = 0.20;
c = 0.95;
```

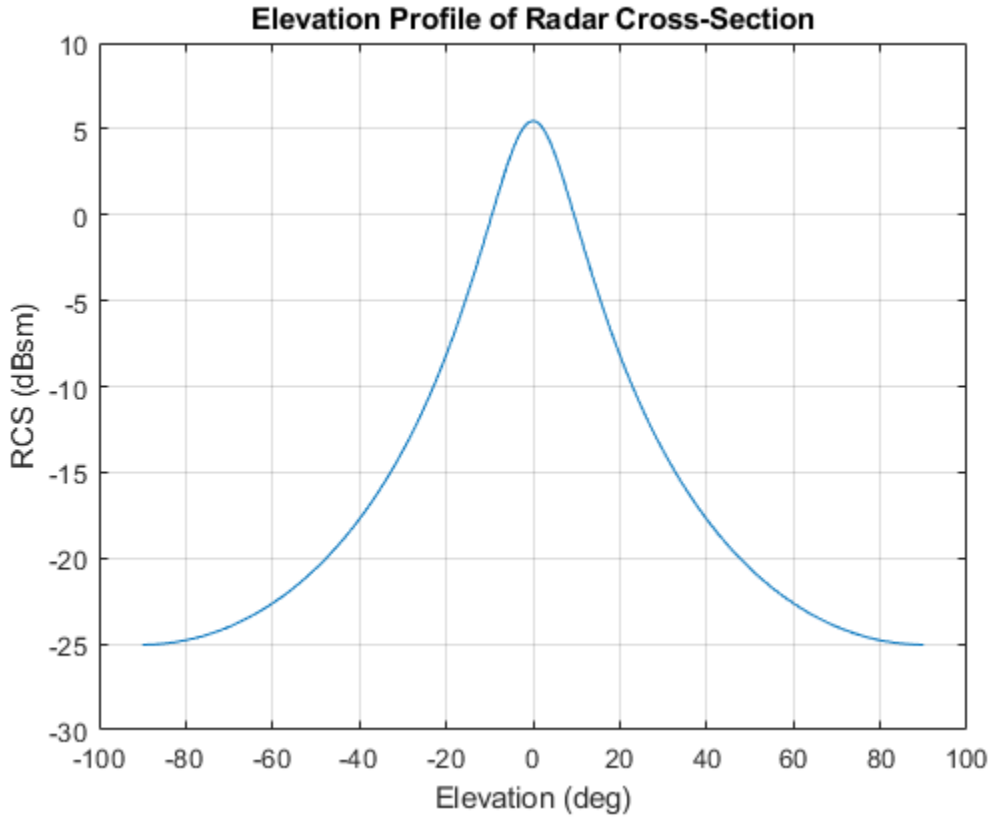
Create an RCS array. Specify the range of azimuth and elevation angles over which RCS is defined. Then, use an analytical model to compute the radar cross-section of the ellipsoid. Create an image of the RCS.

```
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 30
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
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 class

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.

Construction

`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.

Properties

`Pattern` — Target strength pattern
`[10 10; 10 10]` (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
`[-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`

Methods

value	Target strength at specified angle and frequency

Examples

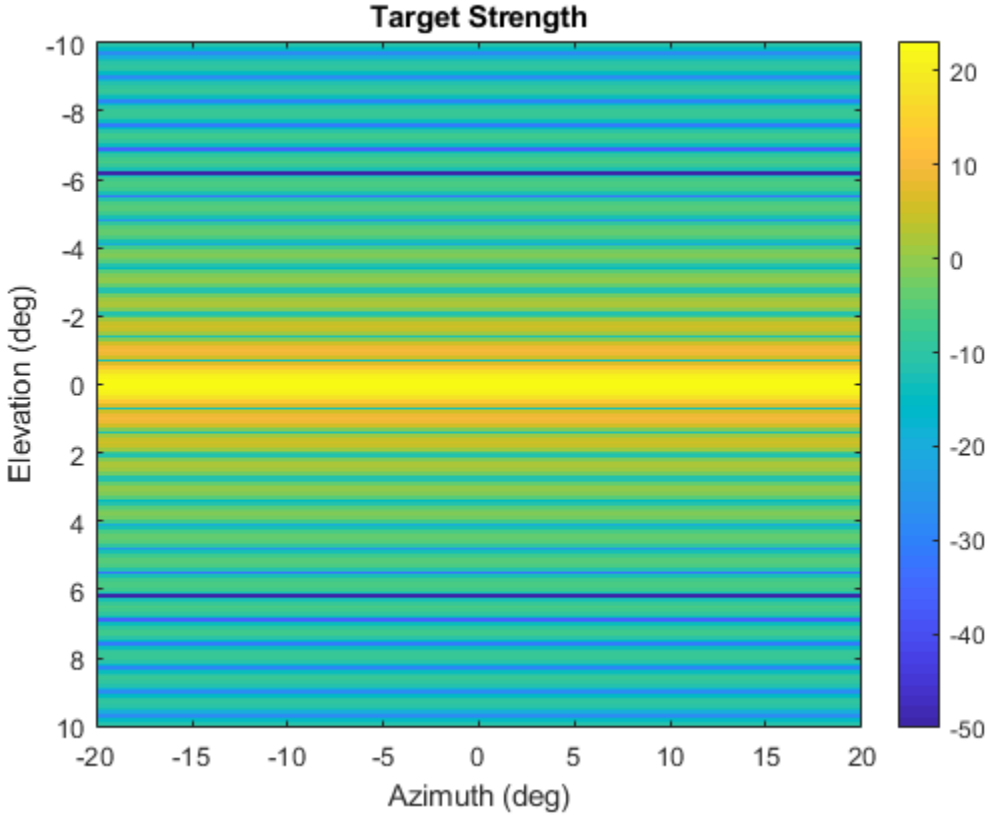
Target Strength of Rigid Cylinder

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.

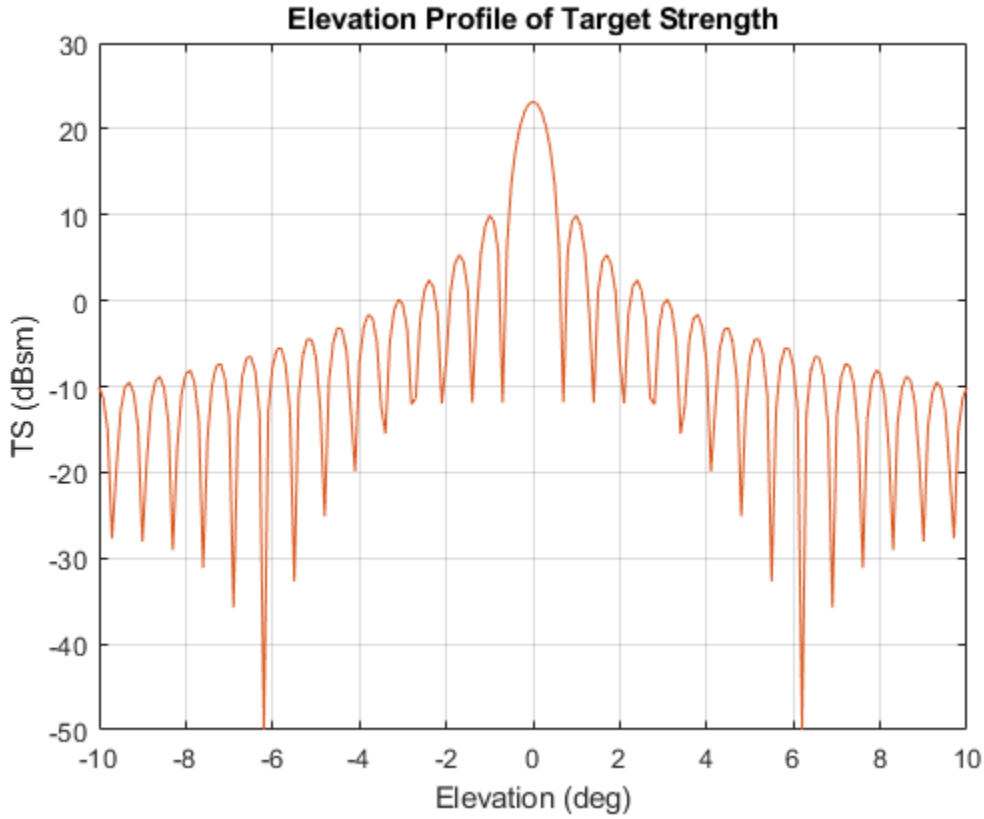
```
lambda = [0.12, .1];
c = 1520.0;
az = [-20:0.1:20];
el = [-10:0.1:10];
ts1 = ts_cylinder(L,a,az,el,lambda(1));
ts2 = ts_cylinder(L,a,az,el,lambda(2));
tsdb1 = 10*log10(ts1);
tsdb2 = 10*log10(ts2);
imagesc(az,el,tsdb1)
title('Target Strength')
xlabel('Azimuth (deg)')
ylabel('Elevation (deg)')
colorbar
```



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

Class: tsSignature

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`.

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.

Examples

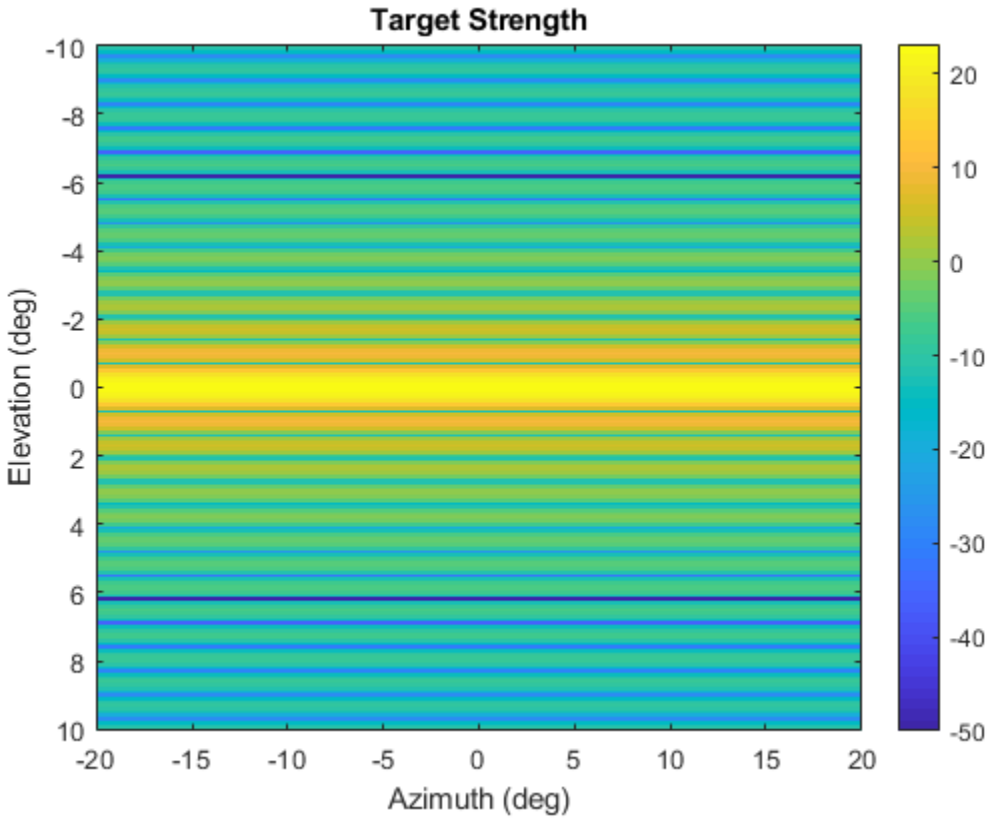
Target Strength of Rigid Cylinder

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;
```

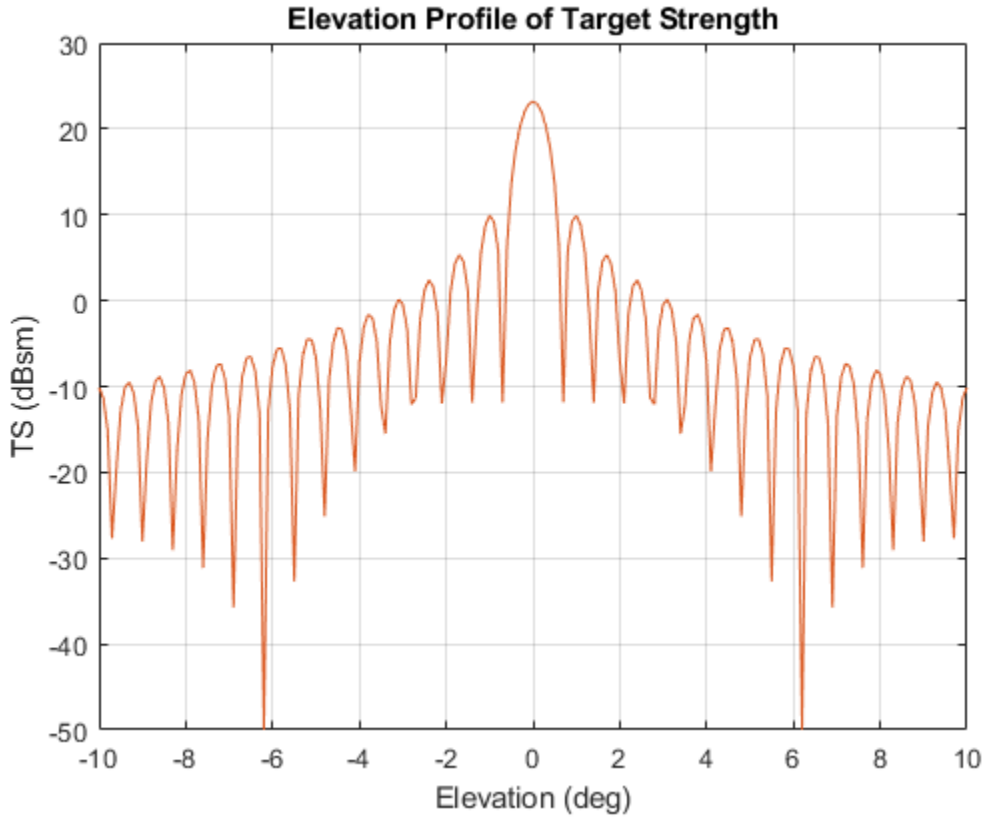
Create an array of target strengths at two wavelengths. First, specify the range of azimuth and elevation angles over which TS is defined. Then, use an analytical model to compute the target strength. Create an image of the TS.

```
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 class

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.

Construction

`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.

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, 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`

Methods

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.

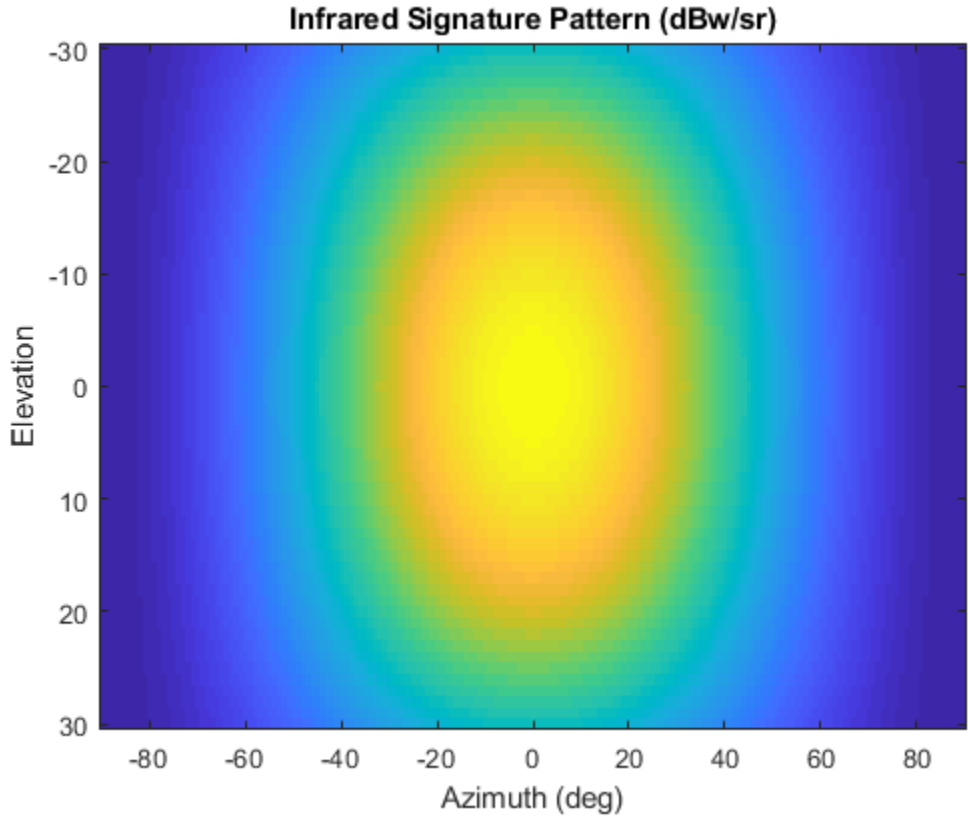
```
az = -90:90;
el = [-30:30];
```

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

Class: irSignature

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`.

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.

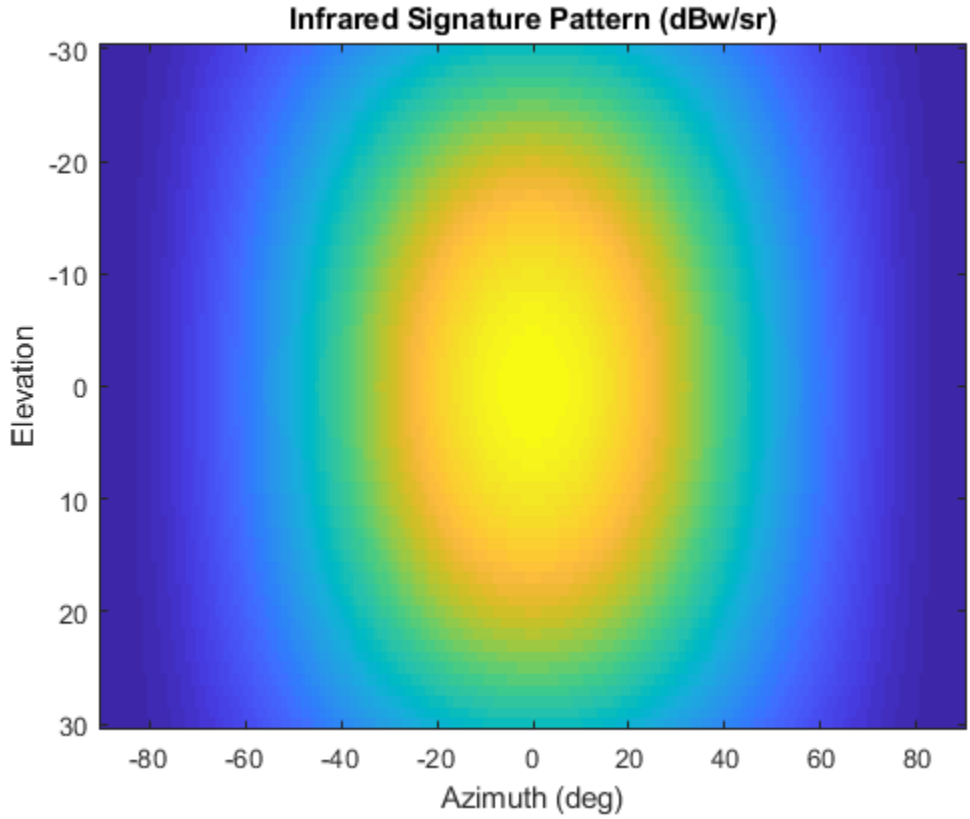
```
az = -90:90;
el = [-30:30];
```

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

trackingKF class

Linear Kalman filter

Description

The `trackingKF` class creates a discrete-time linear Kalman filter used for tracking 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. Both the process and the measurements can have additive noise. The filter also allows for optional controls or forces to act on the vehicle. 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 two ways:

- The first way is to specify explicitly the motion model. Set the motion model property, `MotionModel`, to `Custom` and then use the `StateTransitionModel` property to set the state transition matrix.
- The second way is to 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'`

Construction

`filter = trackingKF` returns 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 `MotionModel` property is set to `'2D Constant Velocity'`.

`filter = trackingKF(F,H)` specifies the state transition model, `F`, and the measurement model, `H`. The `MotionModel` property is set to `'Custom'`.

`filter = trackingKF(F,H,G)` also specifies the control model, `G`. The `MotionModel` property is set 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 using one or more `Name`,`Value` pair arguments. 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.

MotionModel	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]`

MotionModel	Form of State Vector	Form of State Transition Model
`'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.

Example: `[1.0 0.05; 0.05 2]`

Data Types: `double`

`MeasurementModel` — Measurements model from state vector

`[1 0 0 0; 0 0 1 0]` (default) | real-valued *N*-by-*M* matrix

Measurement model, 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`

Methods

clone	Create Linear Kalman filter object with identical property values
correct	Correct Kalman state vector and state covariance matrix
correctjpda	Correct state and state estimation error covariance using JPDA
distance	Distance from measurements to predicted measurement
predict	Predict linear Kalman filter state
initialize	Initialize Kalman filter
likelihood	Measurement likelihood
residual	Measurement residual and residual covariance

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.

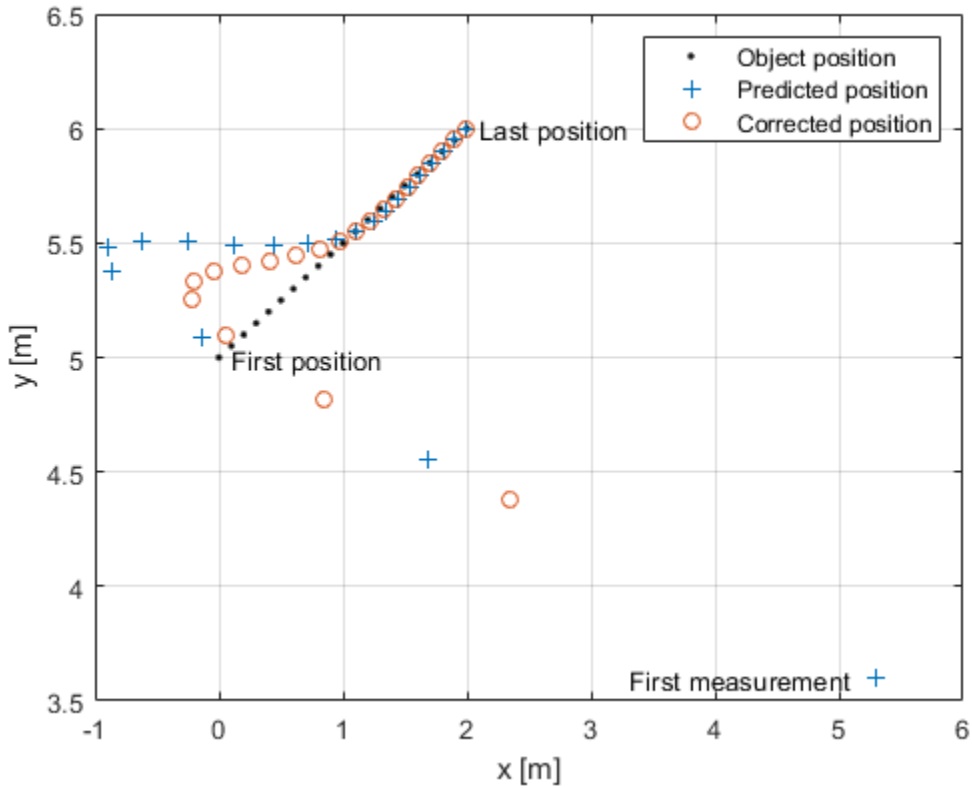
```
vx = 0.2;
vy = 0.1;
T  = 0.5;
pos = [0:vx*T:2;5:vy*T:6]';
```

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')
```



Definitions

Filter Parameters

This table relates the filter model parameters to the object properties. M is the size of the state vector and N is the size of the measurement vector. L is the size of the control model.

Model Parameter	Meaning	Specified in 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
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-

Model Parameter	Meaning	Specified in Property	Size
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.

You can read a brief description of the linear Kalman filter algorithm in “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 a value other than `Custom` for the `MotionModel` value, you must specify the state vector explicitly at construction time using the `State` property. The choice of motion model determines the size of the state vector but does 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`

Classes

`trackingABF` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

System Objects

`trackerGNN` | `trackerTOMHT`

Topics

“Linear Kalman Filters”

Introduced in R2018b

clone

Class: trackingKF

Create Linear Kalman filter object with identical property values

Syntax

```
filter2 = clone(filter)
```

Description

`filter2 = clone(filter)` creates another instance of the object, `filter`, having identical property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

Input Arguments

`filter` — Linear Kalman filter

`trackingKF` object

Linear Kalman filter, specified as a `trackingKF` object.

Example: `filter = trackingKF`

Output Arguments

`filter2` — Linear Kalman filter

`trackingKF` object

Linear Kalman filter, returned as a `trackingKF` object.

Introduced in R2018b

correct

Class: trackingKF

Correct Kalman state vector and state covariance matrix

Syntax

```
[xcorr,Pcorr] = correct(filter,z)
[xcorr,Pcorr] = correct(filter,z,zcov)
```

Description

`[xcorr,Pcorr] = correct(filter,z)` returns the corrected state vector, `xcorr`, and the corrected state error covariance matrix, `Pcorr`, of the tracking filter, `filter`, based on the current measurement, `z`. The internal state and covariance of the Kalman filter are overwritten by the corrected values.

`[xcorr,Pcorr] = correct(filter,z,zcov)` also specifies the measurement error covariance matrix, `zcov`. When specified, `zcov` is used as the measurement noise. Otherwise, measurement noise will have the value of the `MeasurementNoise` property.

The corrected state and covariance replaces the internal values of the Kalman filter.

Input Arguments

`filter` — Kalman filter

`trackingKF` object

Kalman filter, specified as a `trackingKF` object.

Example: `filter = trackingKF`

`z` — Object measurement

real-valued *N*-element vector

Object measurement, specified as a real-valued *N*-element vector.

Example: `[2;1]`

Data Types: `double`

`zcov` — Error covariance matrix of measurements
positive-definite real-valued *N*-by-*N* matrix

Error covariance matrix of measurements, specified as a positive-definite real-valued *N*-by-*N* matrix.

Example: `[2,1;1,20]`

Data Types: `double`

Output Arguments

`xcorr` — Corrected state
real-valued *M*-element vector

Corrected state, returned as a real-valued *M*-element vector. The corrected state represents the *a posteriori* estimate of the state vector, taking into account the current measurement.

`Pcorr` — Corrected state error covariance matrix
positive-definite real-valued *M*-by-*M* matrix

Corrected state error covariance matrix, returned as a positive-definite real-valued *M*-by-*M* matrix. The corrected covariance matrix represents the *a posteriori* estimate of the state error covariance matrix, taking into account the current measurement.

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.

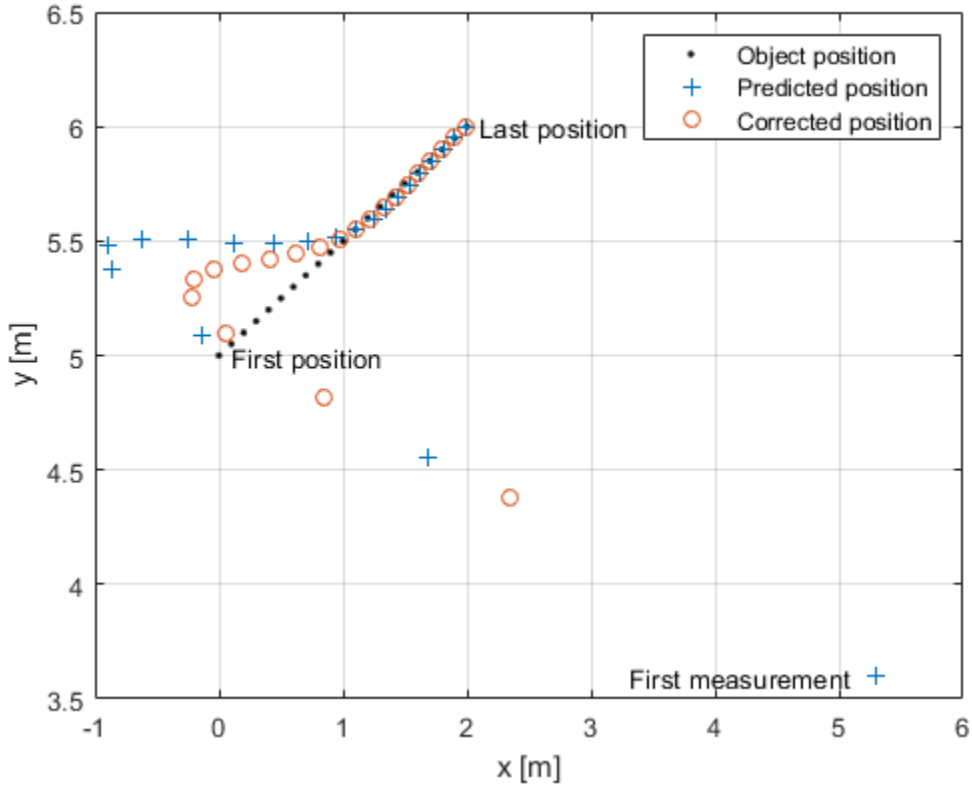
```
vx = 0.2;
vy = 0.1;
T  = 0.5;
pos = [0:vx*T:2;5:vy*T:6]';
```

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')
```



Introduced in R2018b

correctjpda

Class: trackingKF

Correct state and state estimation error covariance using JPDA

Syntax

```
[x_corr,P_corr] = correctjpda(filterObj,z,jp)
[x_corr,P_corr] = correctjpda(filterObj,z,jp,varargin)
[x_corr,P_corr] = correctjpda(filterObj,z,jp,zcov)
[x_corr,P_corr,z_corr] = correctjpda(filterObj,z,jp)
```

Description

`[x_corr,P_corr] = correctjpda(filterObj,z,jp)` returns the correction of state, `x_corr`, and state estimate error covariance, `P_corr`, using a set of measurements `z` and their joint probabilistic data association coefficients `jp`.

This syntax supports a filter object, `filterObj`, created by `trackingKF`, `trackingEKF`, `trackingMSCEKF`, `trackingUKF`, `trackingABF`, `trackingCKF`, `trackingGSF`, `trackingPF`, or `trackingIMM`.

`[x_corr,P_corr] = correctjpda(filterObj,z,jp,varargin)` specifies additional parameters used by the measurement function defined in the `MeasurementFcn` property of the tracking filter object.

This syntax supports a filter object, `filterObj`, created by `trackingEKF`, `trackingMSCEKF`, `trackingUKF`, `trackingCKF`, `trackingGSF`, `trackingPF`, or `trackingIMM` only.

`[x_corr,P_corr] = correctjpda(filterObj,z,jp,zcov)` specifies additional measurement covariance `zcov` used in the `MeasurementNoise` property of a `trackingKF` filter object.

This syntax supports a filter object, `filterObj`, created by `trackingKF` only.

`[x_corr,P_corr,z_corr] = correctjpda(filterObj,z,jp)` also returns the correction of measurements, `z_corr`.

This syntax supports a filter object, `filterObj`, created by `trackingABF` only.

Input Arguments

`filterObj` — Tracking filter

object

Tracking filter, specified as an object. For example, you can create a `trackingEKF` object as

```
EKF = trackingEKF
```

and use `EKF` as the value of `filterObj`.

`z` — 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`

`jp` — 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 `jp` is the joint probability that the *i*th measurement in `z` is associated with the filter. The last element of `jp` corresponds to the probability that no measurement is associated with the filter. The sum of all elements of `jp` equals 1.

Data Types: `single` | `double`

`zcov` — Measurement covariance

M-by-*M* matrix

Measurement covariance matrix, 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 `z`.

Data Types: `single` | `double`

`varargin` — Measurement function arguments
comma-separated list of argument names

Measurement function arguments, specified as a comma-separated list. These arguments are the same ones that are passed into the measurement function specified by the `MeasurementFcn` property of the tracking filter. For example, if you set `MeasurementFcn` to `@cameas`, and then call

```
[x_corr,P_corr] = correctjpda(filter,frame,sensorpos,sensorvel)
```

The `correctjpda` method will internally call

```
meas = cameas(state,frame,sensorpos,sensorvel)
```

Output Arguments

`x_corr` — 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 association probabilities.

`P_corr` — Corrected state error covariance matrix
positive-definite *P*-by-*P* matrix

Corrected state error covariance matrix, 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 association probabilities.

`z_corr` — 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.

Definitions

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 ($i = 1, \ldots, N$) with varied probabilities of association β_i ($i = 0, 1, \ldots, N$). Note that β_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

$$x_k{+} = x_k{-} + K_k \sum_{i=1}^{N} \beta_i (y_i - h(x_k{-}))$$
$$P_k{+} = P_k{-} - (1 - \beta_0) K_k S_k K_k{}^T + P_k$$

where

$$P_k = K_k \sum_{i=1}^{N} \left[\beta_i (y_i - h(x_k{-}))(y_i - h(x_k{-}))^T - (\delta y)(\delta y)^T \right] K_k{}^T$$

and

$$\delta y = \sum_{j=1}^{N} \beta_j \left(y_j - h(x_k^-) \right)$$

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

`trackerJPDA` | `trackingABF` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

Introduced in R2019a

distance

Class: trackingKF

Distance from measurements to predicted measurement

Syntax

```
dist = distance(filter,zmat)
```

Description

`dist = distance(filter,zmat)` computes the Mahalanobis distances, `dist`, between multiple candidate measurements, `zmat`, of an object and the measurement predicted from the state of the tracking filter, `filter`. The `distance` method is useful for associating measurements to tracks.

The distance computation uses the covariance of the predicted state and the covariance of the process noise. You can call the `distance` method only after calling the `predict` method.

Input Arguments

`filter` — Linear Kalman filter

`trackingKF` object

Linear Kalman filter, specified as a `trackingKF` object.

Example: `filter = trackingKF`

`zmat` — Object measurements

real-valued *K*-by-*N* matrix

Object measurements, specified as a real-valued *K*-by-*N* matrix. *N* is the number of rows in the `MeasurementModel` property. *K* is the number of candidate measurement vectors. Each row forms a single measurement vector.

Example: `[2,1;3,0]`

Data Types: `double`

Output Arguments

`dist` — Mahalanobis distances

positive real-valued *K*-element vector

Mahalanobis distances between candidate measurements and a predicted measurement, returned as a real-valued *K*-element vector. *K* is the number of candidate measurement vectors. The method computes one distance value for each measurement vector.

Introduced in R2018b

predict

Class: trackingKF

Predict linear Kalman filter state

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)
```

Description

`[xpred,Ppred] = predict(filter)` returns the predicted state vector and the predicted state error covariance matrix for the next time step based on the current time step. The predicted values overwrite the internal state vector and covariance matrix of the filter.

This syntax applies when you set the `ControlModel` to an empty matrix.

`[xpred,Ppred] = predict(filter,u)` also specifies a control input or force, `u`.

This syntax applies when you set the `ControlModel` to a non-empty matrix.

`[xpred,Ppred] = predict(filter,F)` also 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` to an empty matrix.

`[xpred,Ppred] = predict(filter,F,Q)` also specifies the process noise covariance, `Q`. Use this syntax to change the state transition model and the process noise covariance during a simulation.

This syntax applies when you set the `ControlModel` to an empty matrix.

`[xpred,Ppred] = predict(filter,u,F,G)` also specifies 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` to a non-empty 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` to a non-empty matrix.

`[xpred,Ppred] = predict(filter,dt)` returns the predicted state and state estimation error covariance after the time step, `dt`.

This syntax applies when the `MotionModel` property is not set to `'Custom'` and the `ControlModel` property is set to an empty matrix.

`[xpred,Ppred] = predict(filter,u,dt)` also specifies a control input, `u`.

This syntax applies when the `MotionModel` property is not set to `'Custom'` and the `ControlModel` property is set to a non-empty matrix.

Input Arguments

`filter` — Kalman filter

`trackingKF` object

Kalman filter, specified as `trackingKF` object.

Example: `filter = trackingKF`

`u` — Control vector

real-valued *L*-element vector

Control vector, real-valued *L*-element vector.

Data Types: `double`

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.

Data Types: `double`

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.

Data Types: `double`

G — Control model
real-valued *M*-by-*L* matrix

Control model, specified as a real-valued *M*-by-*L* matrix, where *M* is the size of the state vector and *L* is the number of independent controls.

dt — Time step
positive scalar

Time step, specified as a positive scalar. Units are in seconds.

Data Types: `double`

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.

Data Types: `double`

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.

Data Types: `double`

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]';
```

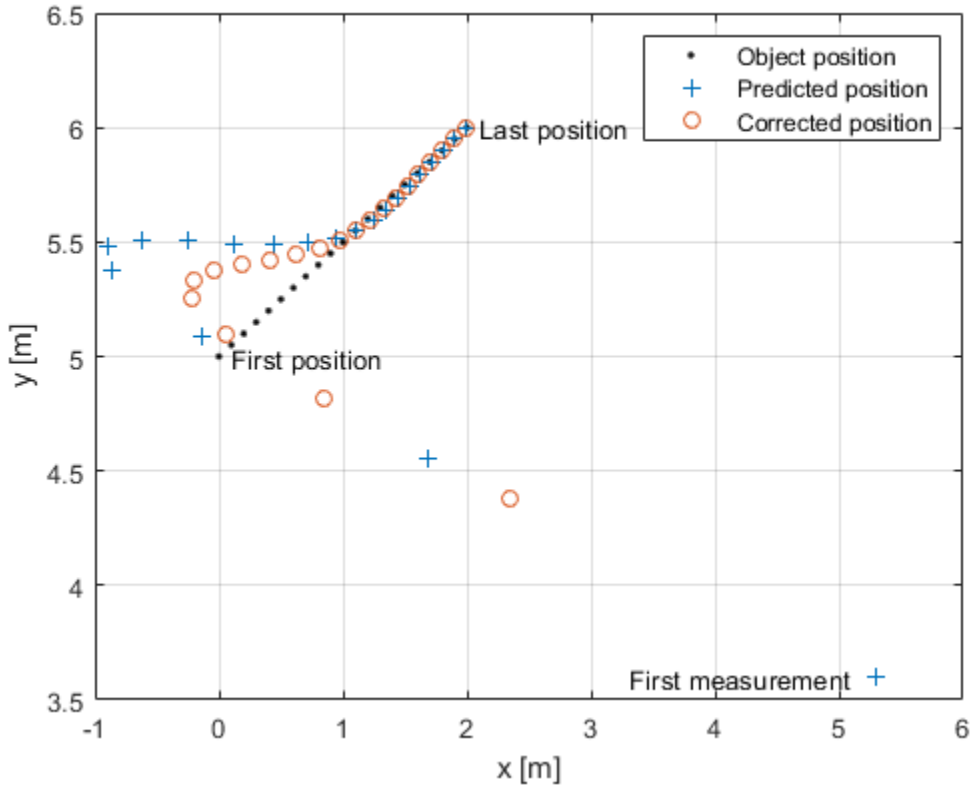
Predict and correct the state of the object.

```
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')
```



Introduced in R2018b

initialize

Class: trackingKF

Initialize Kalman filter

Syntax

```
initialize(filter,X,P)
initialize(filter,X,P,Name,Value)
```

Description

`initialize(filter,X,P)` initializes the Kalman filter, `filter`, using the state, `x`, and the state covariance, `P`.

`initialize(filter,X,P,Name,Value)` initializes the Kalman filter properties using one of more name-value pairs of the filter.

Note: you cannot change the size or type of properties you are initializing.

Input Arguments

`filter` — Kalman tracking filter

`Kalman filter object`

Kalman tracking filter, specified as a Kalman filter object.

`X` — Initial Kalman filter state

vector | matrix

Initial Kalman filter state, specified as a vector or matrix.

`P` — Initial Kalman filter state covariance

matrix

Initial Kalman filter state covariance, specified as a matrix.

Introduced in R2018b

likelihood

Class: trackingKF

Measurement likelihood

Syntax

```
measlikelihood = likelihood(filter,zmeas)
```

Description

`measlikelihood = likelihood(filter,zmeas)` returns the likelihood of the measurement, `zmeas`, of an object tracked by the Kalman filter, `filter`.

Input Arguments

`filter` — Kalman tracking filter

`Kalman filter object`

Kalman tracking filter, specified as a Kalman filter object.

`zmeas` — Measurement of tracked object

vector | matrix

Measurement of the tracked object, specified as a vector or matrix.

Output Arguments

`measlikelihood` — Likelihood of measurement

scalar

Likelihood of measurement, returned as a scalar.

See Also

Introduced in R2018b

residual

Class: trackingKF

Measurement residual and residual covariance

Syntax

```
[zres,rescov] = residual(filter,zmeas)
```

Description

`[zres,rescov] = residual(filter,zmeas)` computes the residual, `zres`, between a measurement, `zmeas`, and a predicted measurement derived from the state of the Kalman filter, `filter`. The function also returns the covariance of the residual, `rescov`.

Input Arguments

`filter` — Linear Kalman tracking filter

`Linear Kalman filter object`

Linear Kalman tracking filter, specified as a Kalman filter object.

`zmeas` — Measurement of tracked object

vector | matrix

Measurement of the tracked object, specified as a vector or matrix.

Output Arguments

`zres` — Residual between measurement and predicted measurement

matrix

Residual between measurement and predicted measurement, returned as a matrix.

`rescov` — Covariance of residuals
matrix

Covariance of the residuals, returned as a matrix.

Algorithms

- The residual is the difference between a measurement and the value predicted by the filter. The residual d is defined as $d = z - Hx$. H is the measurement model set by the `MeasurementModel` property, x is the current filter state, and z is the current measurement.
- The covariance of the residual, S, is defined as $S = HPH' + R$ where P is the state covariance matrix, R is the measurement noise matrix set by the `MeasurementNoise` property.

Introduced in R2018b

trackingEKF class

Extended Kalman filter

Description

The `trackingEKF` class creates a discrete-time extended Kalman filter used for tracking 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 that follows a nonlinear motion model, or when the measurements are nonlinear functions of the state, or both. The filter also allows for optional controls or forces to act on the object. 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 the measurements can have Gaussian noise which can be included in two ways:

- Noise can be added 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.
- Noises can be included in the state transition function, the measurement model function, or both. In these cases, the corresponding noise sizes are not restricted.

Construction

`filter = trackingEKF` creates an extended Kalman filter object for a discrete-time system 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 using one or more `Name,Value` pair arguments. 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.

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.

- If `HasAdditiveProcessNoise` is `true`, specify the function using one of these syntaxes:

  ```
  x(k) = transitionfcn(x(k-1))
  ```

  ```
  x(k) = transitionfcn(x(k-1),parameters)
  ```

  where `x(k)` is the state at time `k`. The `parameters` term stands for all additional arguments required by the state transition function.
- 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),parameters)
```

where `x(k)` is the state at time `k` and `w(k)` is a value for the process noise at time `k`. The `parameters` argument stands for all additional arguments required by the state transition function.

Example: `@constacc`

Data Types: `function_handle`

`StateTransitionJacobianFcn` — State transition function Jacobian
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),parameters)
  ```

  where `x(k)` is the state at time `k`. The `parameters` argument stands for all additional arguments required by the state transition function.

  `Jx(k)` denotes the Jacobian of the predicted state with respect to the previous state. The Jacobian is an *M*-by-*M* matrix at time `k`. The Jacobian function can take additional input parameters, such as control inputs or time step size.
- 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),parameters)
  ```

  where `x(k)` is the state at time `k` and `w(k)` is a sample *Q*-element vector of the process noise at time `k`. *Q* is the size of the process noise covariance. Unlike the case of additive process noise, the process noise vector in the non-additive noise case need not have the same dimensions as the state vector.

`Jx(k)` denotes the Jacobian of the predicted state with respect to the previous state. This Jacobian is an *M*-by-*M* matrix at time *k*. The Jacobian function can take additional input parameters, such as control inputs or time step size.

`Jw(k)` denotes the *M*-by-*Q* Jacobian of the predicted state with respect to the process noise elements.

If not specified, the Jacobians are computed by numerical differencing at each call of the `predict` method. This computation can increase the processing time and numerical inaccuracy.

Example: `@constaccjac`

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 an *Q*-by-*Q* matrix. *Q* is the size of the process noise vector.

  You must specify `ProcessNoise` before any call to the `predict` method. 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 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
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`

`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 `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`. `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 `HasAdditiveMeasurmentNoise` 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 *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` method. 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` method. 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`

`HasAdditiveMeasurmentNoise` — 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.

Methods

clone	Create extended Kalman filter object with identical property values
correct	Correct Kalman state vector and state error covariance matrix
correctjpda	Correct state and state estimation error covariance using JPDA
distance	Distance from measurements to predicted measurement
predict	Predict extended Kalman state vector and state error covariance matrix
initialize	Initialize extended Kalman filter
likelihood	Measurement likelihood
residual	Measurement residual and residual covariance

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` methods 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
```

Definitions

Filter Parameters

This table relates the filter model parameters to the object properties. In this table, M is the size of the state vector and N is the size of the measurement vector.

Filter Parameter	Meaning	Specified in 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

Filter Parameter	Meaning	Specified in Property	Size
Q_k	Estimate of the process noise covariance matrix at step k. Process noise is a measure of the uncertainty in the dynamic model. It is assumed to be zero-mean white Gaussian noise.	`ProcessNoise`	M-by-M matrix when `HasAdditiveProcessNoise` is `true`. Q-by-Q matrix when `HasAdditiveProcessNoise` is `false`
R_k	Estimate of the measurement noise covariance at step k. Measurement noise reflects the uncertainty of the measurement. It is assumed to be zero-mean white Gaussian noise.	`MeasurementNoise`	N-by-N matrix when `HasAdditiveMeasurementNoise` is `true`. R-by-R when `HasAdditiveMeasurementNoise` is `false`.
F	Function determining Jacobian of propagated state with respect to previous state.	`StateTransitionJacobianFcn`	M-by-M matrix
H	Function determining Jacobians of measurement with respect to the state and measurement noise.	`MeasurementJacobianFcn`	N-by-M for state vector Jacobian and N-by-R for measurement vector Jacobian

Algorithms

The extended Kalman filter estimates the state of a process governed by this nonlinear stochastic equation:

$$x_{k+1} = f(x_k, u_k, w_k, t)$$

x_k is the state at step k. *f()* is the state transition function. Random noise perturbations, w_k, can affect the object motion. The filter also supports a simplified form,

$$x_{k+1} = f(x_k, u_k, t) + w_k$$

To use the simplified form, set `HasAdditiveProcessNoise` to `true`.

In the extended Kalman filter, the measurements are also general functions of the state:

$$z_k = h(x_k, v_k, t)$$

h(x_k,v_k,t) is the measurement function that determines the measurements as functions of the state. Typical measurements are position and velocity or some function of position and velocity. The measurements can also include noise, represented by v_k. Again, the filter offers a simpler formulation.

$$z_k = h(x_k, t) + v_k$$

To use the simplified form, set `HasAdditiveMeasurmentNoise` 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`

Classes

`trackingABF` | `trackingCKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

System Objects

`trackerGNN` | `trackerTOMHT`

Topics

"Extended Kalman Filters"

Introduced in R2018b

clone

Class: trackingEKF

Create extended Kalman filter object with identical property values

Syntax

```
filter2 = clone(filter)
```

Description

`filter2 = clone(filter)` creates another instance of the object, `trackingEKF`, having identical property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

Input Arguments

`filter` — Extended Kalman filter

`trackingEKF` object

Extended Kalman filter, specified as a `trackingEKF` object.

Example: `filter = trackingEKF`

Output Arguments

`filter2` — Extended Kalman filter

`trackingEKF` object

Extended Kalman filter, returned as a `trackingEKF` object.

Introduced in R2018b

correct

Class: trackingEKF

Correct Kalman state vector and state error covariance matrix

Syntax

```
[xcorr,Pcorr] = correct(filter,z)
[xcorr,Pcorr] = correct(filter,z,varargin)
```

Description

`[xcorr,Pcorr] = correct(filter,z)` returns the corrected state vector, `xcorr`, and the corrected state error covariance matrix, `Pcorr`, for the extended Kalman filter defined in `filter`, based on the current measurement, `z`. The internal state and covariance of the Kalman filter are overwritten by the corrected values.

`[xcorr,Pcorr] = correct(filter,z,varargin)` also specifies any input arguments to the measurement function. These arguments are used as input to the measurement function specified in the `MeasurementFcn` property.

Input Arguments

`filter` — Extended Kalman filter

`trackingEKF` object

Extended Kalman filter, specified as a `trackingEKF` object.

Example: `filter = trackingEKF`

`z` — Object measurement

real-valued *N*-element vector

Object measurement, specified as a real-valued *N*-element vector.

Example: `[2;1]`

`varargin` — Measurement function arguments

comma-separated list

Measurement function arguments, specified as a comma-separated list. These arguments are the same ones that are passed into the measurement function specified by the `MeasurementFcn` property. For example, if you set `MeasurementFcn` to `@cameas`, and then call

```
[xcorr,Pcorr] = correct(filter,frame,sensorpos,sensorvel)
```

the `correct` method will internally call

```
meas = cameas(state,frame,sensorpos,sensorvel)
```

.

Output Arguments

`xcorr` — Corrected state

real-valued *M*-element vector

Corrected state, returned as a real-valued *M*-element vector. The corrected state represents the *a posteriori* estimate of the state vector, taking into account the current measurement.

`Pcorr` — Corrected state error covariance matrix

positive-definite real-valued *M*-by-*M* matrix

Corrected state error covariance matrix, returned as a positive-definite real-valued *M*-by-*M* matrix. The corrected state covariance matrix represents the *a posteriori* estimate of the state covariance matrix, taking into account the current measurement.

Introduced in R2018b

correctjpda

Class: trackingEKF

Correct state and state estimation error covariance using JPDA

Syntax

```
[x_corr,P_corr] = correctjpda(filterObj,z,jp)
[x_corr,P_corr] = correctjpda(filterObj,z,jp,varargin)
[x_corr,P_corr] = correctjpda(filterObj,z,jp,zcov)
[x_corr,P_corr,z_corr] = correctjpda(filterObj,z,jp)
```

Description

`[x_corr,P_corr] = correctjpda(filterObj,z,jp)` returns the correction of state, `x_corr`, and state estimate error covariance, `P_corr`, using a set of measurements `z` and their joint probabilistic data association coefficients `jp`.

This syntax supports a filter object, `filterObj`, created by `trackingKF`, `trackingEKF`, `trackingMSCEKF`, `trackingUKF`, `trackingABF`, `trackingCKF`, `trackingGSF`, `trackingPF`, or `trackingIMM`.

`[x_corr,P_corr] = correctjpda(filterObj,z,jp,varargin)` specifies additional parameters used by the measurement function defined in the `MeasurementFcn` property of the tracking filter object.

This syntax supports a filter object, `filterObj`, created by `trackingEKF`, `trackingMSCEKF`, `trackingUKF`, `trackingCKF`, `trackingGSF`, `trackingPF`, or `trackingIMM` only.

`[x_corr,P_corr] = correctjpda(filterObj,z,jp,zcov)` specifies additional measurement covariance `zcov` used in the `MeasurementNoise` property of a `trackingKF` filter object.

This syntax supports a filter object, `filterObj`, created by `trackingKF` only.

`[x_corr,P_corr,z_corr] = correctjpda(filterObj,z,jp)` also returns the correction of measurements, `z_corr`.

This syntax supports a filter object, `filterObj`, created by `trackingABF` only.

Input Arguments

`filterObj` — Tracking filter

object

Tracking filter, specified as an object. For example, you can create a `trackingEKF` object as

```
EKF = trackingEKF
```

and use `EKF` as the value of `filterObj`.

`z` — 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`

`jp` — 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 `jp` is the joint probability that the *i*th measurement in `z` is associated with the filter. The last element of `jp` corresponds to the probability that no measurement is associated with the filter. The sum of all elements of `jp` equals 1.

Data Types: `single` | `double`

`zcov` — Measurement covariance

M-by-*M* matrix

Measurement covariance matrix, 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 `z`.

Data Types: `single` | `double`

`varargin` — Measurement function arguments
comma-separated list of argument names

Measurement function arguments, specified as a comma-separated list. These arguments are the same ones that are passed into the measurement function specified by the `MeasurementFcn` property of the tracking filter. For example, if you set `MeasurementFcn` to `@cameas`, and then call

```
[x_corr,P_corr] = correctjpda(filter,frame,sensorpos,sensorvel)
```

The `correctjpda` method will internally call

```
meas = cameas(state,frame,sensorpos,sensorvel)
```

Output Arguments

`x_corr` — 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 association probabilities.

`P_corr` — Corrected state error covariance matrix
positive-definite *P*-by-*P* matrix

Corrected state error covariance matrix, 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 association probabilities.

`z_corr` — 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.

Definitions

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 ($i = 1, \ldots, N$) with varied probabilities of association β_i ($i = 0, 1, \ldots, N$). Note that β_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

$$x_k{+} = x_k{-} + K_k \sum_{i=1}^{N} \beta_i (y_i - h(x_k{-}))$$
$$P_k{+} = P_k{-} - (1 - \beta_0) K_k S_k K_k{}^T + P_k$$

where

$$P_k = K_k \sum_{i=1}^{N} \left[\beta_i (y_i - h(x_k{-}))(y_i - h(x_k{-}))^T - (\delta y)(\delta y)^T \right] K_k{}^T$$

and

$$\delta y = \sum_{j=1}^{N} \beta_j \left(y_j - h(x_k^-) \right)$$

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

`trackerJPDA` | `trackingABF` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

Introduced in R2019a

distance

Class: trackingEKF

Distance from measurements to predicted measurement

Syntax

```
dist = distance(filter,zmat)
dist = distance(filter,zmat,measurementParams)
```

Description

`dist = distance(filter,zmat)` computes the Mahalanobis distances between multiple candidate measurements of an object, `zmat`, and the predicted measurement computed by the `trackingEKF` object. The distance method is used to assign measurements to tracks.

This distance computation takes into account the covariance of the predicted state and the covariance of the process noise. You can call the `distance` method only after calling the `predict` method.

`dist = distance(filter,zmat,measurementParams)` also specifies the parameters used by the measurement function set in the `MeasurementFcn` property.

Input Arguments

`filter` — Extended Kalman filter
`trackingEKF` object

Extended Kalman filter, specified as a `trackingEKF` object.

Example: `filter = trackingEKF`

`zmat` — Object measurements
real-valued *K*-by-*N* matrix

Measurements, specified as a real-valued *K*-by-*N* matrix. *K* is the number of candidate measurement vectors. Each row corresponds to a candidate measurement vector.*N* is the number of rows in the output of the function specified by the `MeasurementFcn` property.

Example: `[2,1;3,0]`

Data Types: `double`

`measurementParams` — Measurement function parameters
`{}` (default) | cell array

Measurement function parameters, specified as a cell array containing arguments to the measurement function specified by the `MeasurementFcn` property. Suppose you set `MeasurementFcn` to `@cameas`, and then set these values:

```
measurementParams = {frame,sensorpos,sensorpos)
```

The `distance` method internally calls the following:

```
cameas(state,frame,sensorpos,sensorvel)
```

Data Types: `cell`

Output Arguments

`dist` — Mahalanobis distances
real-valued *K*-element vector of positive values

Mahalanobis distances between candidate measurements and the predicted measurement, returned as a real-valued *K*-element vector of positive values. There is one distance value per measurement vector.

Data Types: `double`

Introduced in R2018b

predict

Class: trackingEKF

Predict extended Kalman state vector and state error covariance matrix

Syntax

```
[xpred,Ppred] = predict(filter)
[xpred,Ppred] = predict(filter,varargin)
[xpred,Ppred] = predict( ___ ,dt)
```

Description

`[xpred,Ppred] = predict(filter)` returns the predicted state vector, `xpred`, and state error covariance matrix, `Ppred`, at the next time step based on the current time step. The predicted values overwrite the internal state vector and state error covariance matrix of the extended Kalman filter.

`[xpred,Ppred] = predict(filter,varargin)` specifies input arguments, `varargin`, for the state transition function set in the `StateTransitionFcn` property.

`[xpred,Ppred] = predict( ___ ,dt)` also specifies the time step, `dt`.

Input Arguments

`filter` — Extended Kalman filter

`trackingEKF` object

Extended Kalman filter, specified as a `trackingEKF` object.

Example: `filter = trackingEKF`

`varargin` — State transition function arguments

comma-separated list

State transition function arguments, specified as a comma-separated list. These arguments are the same ones that are passed into the state transition function specified by the `StateTransitionFcn` property. For example, if you set the `StateTransitionFcn` property to `@constacc`, and then call

```
[xpred,Ppred] = predict(filter,dt)
```

the `predict` method will internally call

```
state = constacc(state,dt)
```

`dt` — Time step
positive scalar

Time step, specified as a positive scalar. Units are in seconds.

Data Types: `double`

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 *a priori* estimate of the state vector propagated from the previous state. The prediction uses the state transition function specified in the `StateTransitionFcn` property.

Data Types: `double`

`Ppred` — Predicted state error covariance matrix
real-valued *M*-by-*M* matrix

Predicted state error covariance matrix, returned as a real-valued *M*-by-*M* matrix. This predicted error is the *a priori* estimate of the state error covariance matrix. `predict` uses the state transition function Jacobian specified in the `StateTransitionJacobianFcn` property.

Data Types: `double`

Introduced in R2018b

initialize

Class: trackingEKF

Initialize extended Kalman filter

Syntax

```
initialize(filterobj,X,P)
initialize(filterobj,X,P,Name,Value)
```

Description

`initialize(filterobj,X,P)` initializes the extended Kalman filter, `filterobj`, using the state, `x`, and the state covariance, `P`.

`initialize(filterobj,X,P,Name,Value)` initializes Kalman filter properties using name-value pairs.

Note: you cannot change the size or type of properties you are initializing.

Input Arguments

`filterobj` — Extended Kalman tracking filter

`Extended Kalman filter object`

Kalman tracking filter, specified as a Kalman filter object.

X — Initial extended Kalman filter state

vector | matrix

Initial extended Kalman filter state, specified as a vector or matrix.

P — Initial extended Kalman filter state covariance

matrix

Initial extended Kalman filter state covariance, specified as a matrix.

Introduced in R2018b

likelihood

Class: trackingEKF

Measurement likelihood

Syntax

```
measlikelihood = likelihood(filterobj,zmeas)
measlikelihood = likelihood(filterobj,zmeas,measparams)
```

Description

`measlikelihood = likelihood(filterobj,zmeas)` returns the likelihood of the measurement, `zmeas`, of an object tracked by the extended Kalman filter, `filterobj`.

`measlikelihood = likelihood(filterobj,zmeas,measparams)` also specifies measurement parameters, `measparams`.

Input Arguments

`filterobj` — Extended Kalman tracking filter
`Kalman filter object`

Extended Kalman tracking filter, specified as an extended Kalman filter object.

`zmeas` — Measurement of tracked object
vector | matrix

Measurement of the 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 defined in the `MeasurementFcn` property of the Extended Kalman filter, `filterobj`.

Output Arguments

`measlikelihood` — Likelihood of measurement
scalar

Likelihood of measurement, returned as a scalar.

See Also

Introduced in R2018b

residual

Class: trackingEKF

Measurement residual and residual covariance

Syntax

```
[zres,rescov] = residual(filterobj,zmeas)
[zres,rescov] = residual(filterobj,zmeas,measparams)
```

Description

`[zres,rescov] = residual(filterobj,zmeas)` computes the residual, `zres`, between a measurement, `zmeas`, and a predicted measurement produced by the Kalman filter, `filterobj`. The function also returns the covariance of the residual, `zres`.

`[zres,rescov] = residual(filterobj,zmeas,measparams)` also specifies measurement parameters, `measparams`.

Input Arguments

`filterobj` — Kalman tracking filter

`Kalman filter object`

Kalman tracking filter, specified as a Kalman filter object.

`zmeas` — Measurement of tracked object

vector | matrix

Measurement of the 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 defined in the `MeasurementFcn` property of the `filterobj`

Output Arguments

`zres` — Residual between measurement and predicted measurement

matrix

Residual between measurement and predicted measurement, returned as a matrix.

`rescov` — Covariance of residuals

matrix

Covariance of the residuals, returned as a matrix.

Algorithms

- The residual is the difference between a measurement and the value predicted by the filter. The residual d is defined as $d = z - h(x)$. h is the measurement function set by the `MeasurementFcn` property, x is the current filter state, and z is the current measurement.
- The covariance of the residual, S, is defined as $S = HPH' + R$ where P is the state covariance matrix, R is the measurement noise matrix set by the `MeasurementNoise` property.

Introduced in R2018b

trackingUKF class

Unscented Kalman filter

Description

The `trackingUKF` class creates a discrete-time unscented Kalman filter used for tracking positions and velocities of 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. In addition, the process and the measurements can have noise. Use an unscented Kalman filter when the current state is a nonlinear function of the previous state or when the measurements are nonlinear functions of the state or when both conditions apply. The unscented Kalman filter estimates the uncertainty about the state, and its propagation through the nonlinear state and measurement equations, using a fixed number of sigma points. Sigma points are chosen using the unscented transformation as parameterized by the `Alpha`, `Beta`, and `Kappa` properties.

Construction

`filter = trackingUKF` creates an unscented Kalman filter object for a discrete-time system 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. 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.

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.

- If `HasAdditiveProcessNoise` is `true`, specify the function using one of these syntaxes:

  ```
  x(k) = transitionfcn(x(k-1))
  ```

  ```
  x(k) = transitionfcn(x(k-1),parameters)
  ```

  where `x(k)` is the state at time `k`. The `parameters` term stands for all additional arguments required by the state transition function.
- 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),parameters)
  ```

where `x(k)` is the state at time `k` and `w(k)` is a value for the process noise at time `k`. The `parameters` argument stands for all additional arguments required by the state transition function.

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 an *Q*-by-*Q* matrix. *Q* is the size of the process noise vector.

  You must specify `ProcessNoise` before any call to the `predict` method. 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 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

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, 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` method. 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`

`HasAdditiveMeasurmentNoise` — 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 zero and less than or equal to one.

`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.

Methods

clone	Create unscented Kalman filter object with identical property values
correct	Correct Kalman state vector and state error covariance matrix
correctjpda	Correct state and state estimation error covariance using JPDA
distance	Distance from measurements to predicted measurement
predict	Predict unscented Kalman state vector and state error covariance matrix
initialize	Initialize unscented Kalman filter
likelihood	Measurement likelihood
residual	Measurement residual and residual covariance

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` methods to propagate the state. You can call `predict` and `correct` in any order and as many times as you want.

```
meas = [1;1;0];
[xpred, Ppred] = predict(filter);
[xcorr, Pcorr] = correct(filter,meas);
[xpred, Ppred] = predict(filter);
[xpred, Ppred] = predict(filter)
```

```
xpred = 4×1

    1.2500
    0.2500
    1.2500
    0.2500


Ppred = 4×4

   11.7500    4.7500   -0.0000    0.0000
    4.7500    3.7500   -0.0000    0.0000
   -0.0000   -0.0000   11.7500    4.7500
    0.0000    0.0000    4.7500    3.7500
```

Definitions

Filter parameters and dimensions

This table relates the filter model parameters to the object properties. M is the size of the state vector and N is the size of the measurement vector.

Filter Parameter	Meaning	Specified in 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
P_k	State error covariance matrix representing the uncertainty in the values of the state	`StateCovariance`	M-by-M

Filter Parameter	Meaning	Specified in Property	Size
Q_k	Estimate of the process noise covariance matrix at step k. Process noise is measure of the uncertainty in your dynamic model and is assumed to be zero-mean white Gaussian noise	`ProcessNoise`	M-by-M when `HasAdditiveProcessNoise` is `true`. Q-by-Q when `HasAdditiveProcessNoise`is `false`.
R_k	Estimate of the measurement noise covariance at step k. Measurement noise reflects the uncertainty of the measurement and is assumed to be zero-mean white Gaussian noise.	`MeasurementNoise`	N-by-N when `HasAdditiveMeasurementNoise` is `true`. R-by-R when `HasAdditiveMeasurementNoise` is `false`.
α	Determines spread of sigma points.	`Alpha`	scalar
β	*A priori* knowledge of sigma point distribution.	`Beta`	scalar
κ	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 `HasAdditiveMeasurmentNoise` 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`

Classes

`trackingABF` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF`

System Objects

`trackerGNN` | `trackerTOMHT`

Introduced in R2018b

clone

Class: trackingUKF

Create unscented Kalman filter object with identical property values

Syntax

```
filter2 = clone(filter)
```

Description

`filter2 = clone(filter)` creates another instance of the object, `trackingUKF`, having identical property values. If an object is locked, the `clone` method creates a copy that is also locked and has states initialized to the same values as the original. If an object is not locked, the `clone` method creates a new unlocked object with uninitialized states.

Input Arguments

`filter` — Unscented Kalman filter

`trackingUKF` object

Unscented Kalman filter, specified as a `trackingUKF` object.

Example: `filter = trackingEKF`

Output Arguments

`filter2` — Unscented Kalman filter

`trackingUKF` object

Unscented Kalman filter, returned as a `trackingUKF` object.

Introduced in R2018b

correct

Class: trackingUKF

Correct Kalman state vector and state error covariance matrix

Syntax

```
[xcorr,Pcorr] = correct(filter,z)
[xcorr,Pcorr] = correct(filter,z,varargin)
```

Description

`[xcorr,Pcorr] = correct(filter,z)` returns the corrected state vector, `xcorr`, and the corrected state error covariance matrix, `Pcorr`, for the unscented Kalman filter defined in `filter`, based on the current measurement, `z`. The internal state and covariance of the Kalman filter are overwritten by the corrected values.

`[xcorr,Pcorr] = correct(filter,z,varargin)` also specifies any input arguments to the measurement function. These arguments are used as input to the measurement function specified in the `MeasurementFcn` property.

Input Arguments

`filter` — Unscented Kalman filter
`trackingUKF` object

Unscented Kalman filter, specified as a `trackingUKF` object.

Example: `filter = trackingUKF`

`z` — Object measurement
real-valued *N*-element vector

Object measurement, specified as a real-valued *N*-element vector.

Example: `[2;1]`

`varargin` — Measurement function arguments

comma-separated list

Measurement function arguments, specified as a comma-separated list. These arguments are the same ones that are passed into the measurement function specified by the `MeasurementFcn` property. For example, if you set `MeasurementFcn` to `@cameas`, and then call

```
[xcorr,Pcorr] = correct(filter,frame,sensorpos,sensorvel)
```

the `correct` method will internally call

```
meas = cameas(state,frame,sensorpos,sensorvel)
```

.

Output Arguments

`xcorr` — Corrected state

real-valued *M*-element vector

Corrected state, returned as a real-valued *M*-element vector. The corrected state represents the *a posteriori* estimate of the state vector, taking into account the current measurement.

`Pcorr` — Corrected state error covariance matrix

positive-definite real-valued *M*-by-*M* matrix

Corrected state error covariance matrix, returned as a positive-definite real-valued *M*-by-*M* matrix. The corrected state covariance matrix represents the *a posteriori* estimate of the state covariance matrix, taking into account the current measurement.

Introduced in R2018b

correctjpda

Class: trackingUKF

Correct state and state estimation error covariance using JPDA

Syntax

```
[x_corr,P_corr] = correctjpda(filterObj,z,jp)
[x_corr,P_corr] = correctjpda(filterObj,z,jp,varargin)
[x_corr,P_corr] = correctjpda(filterObj,z,jp,zcov)
[x_corr,P_corr,z_corr] = correctjpda(filterObj,z,jp)
```

Description

`[x_corr,P_corr] = correctjpda(filterObj,z,jp)` returns the correction of state, `x_corr`, and state estimate error covariance, `P_corr`, using a set of measurements `z` and their joint probabilistic data association coefficients `jp`.

This syntax supports a filter object, `filterObj`, created by `trackingKF`, `trackingEKF`, `trackingMSCEKF`, `trackingUKF`, `trackingABF`, `trackingCKF`, `trackingGSF`, `trackingPF`, or `trackingIMM`.

`[x_corr,P_corr] = correctjpda(filterObj,z,jp,varargin)` specifies additional parameters used by the measurement function defined in the `MeasurementFcn` property of the tracking filter object.

This syntax supports a filter object, `filterObj`, created by `trackingEKF`, `trackingMSCEKF`, `trackingUKF`, `trackingCKF`, `trackingGSF`, `trackingPF`, or `trackingIMM` only.

`[x_corr,P_corr] = correctjpda(filterObj,z,jp,zcov)` specifies additional measurement covariance `zcov` used in the `MeasurementNoise` property of a `trackingKF` filter object.

This syntax supports a filter object, `filterObj`, created by `trackingKF` only.

`[x_corr,P_corr,z_corr] = correctjpda(filterObj,z,jp)` also returns the correction of measurements, `z_corr`.

This syntax supports a filter object, `filterObj`, created by `trackingABF` only.

Input Arguments

`filterObj` — Tracking filter

object

Tracking filter, specified as an object. For example, you can create a `trackingEKF` object as

```
EKF = trackingEKF
```

and use `EKF` as the value of `filterObj`.

`z` — 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`

`jp` — 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 `jp` is the joint probability that the *i*th measurement in `z` is associated with the filter. The last element of `jp` corresponds to the probability that no measurement is associated with the filter. The sum of all elements of `jp` equals 1.

Data Types: `single` | `double`

`zcov` — Measurement covariance

M-by-*M* matrix

Measurement covariance matrix, 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 `z`.

Data Types: `single` | `double`

`varargin` — Measurement function arguments
comma-separated list of argument names

Measurement function arguments, specified as a comma-separated list. These arguments are the same ones that are passed into the measurement function specified by the `MeasurementFcn` property of the tracking filter. For example, if you set `MeasurementFcn` to `@cameas`, and then call

```
[x_corr,P_corr] = correctjpda(filter,frame,sensorpos,sensorvel)
```

The `correctjpda` method will internally call

```
meas = cameas(state,frame,sensorpos,sensorvel)
```

Output Arguments

`x_corr` — 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 association probabilities.

`P_corr` — Corrected state error covariance matrix
positive-definite *P*-by-*P* matrix

Corrected state error covariance matrix, 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 association probabilities.

`z_corr` — 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.

Definitions

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 ($i = 1, \ldots, N$) with varied probabilities of association β_i ($i = 0, 1, \ldots, N$). Note that β_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

$$x_k{+} = x_k{-} + K_k \sum_{i=1}^{N} \beta_i (y_i - h(x_k{-}))$$
$$P_k{+} = P_k{-} - (1 - \beta_0) K_k S_k K_k{}^T + P_k$$

where

$$P_k = K_k \sum_{i=1}^{N} \left[\beta_i (y_i - h(x_k{-}))(y_i - h(x_k{-}))^T - (\delta y)(\delta y)^T \right] K_k{}^T$$

and

$$\delta y = \sum_{j=1}^{N} \beta_j \left(y_j - h(x_k^-) \right)$$

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

`trackerJPDA` | `trackingABF` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

Introduced in R2019a

distance

Class: trackingUKF

Distance from measurements to predicted measurement

Syntax

```
dist = distance(filter,zmat)
dist = distance(filter,zmat,measurementParams)
```

Description

`dist = distance(filter,zmat)` computes the Mahalanobis distances between multiple candidate measurements of an object, `zmat`, and the predicted measurement computed by the `trackingUKF` object. The distance method is used to assign measurements to tracks.

This distance computation takes into account the covariance of the predicted state and the covariance of the process noise. You can call the `distance` method only after calling the `predict` method.

`dist = distance(filter,zmat,measurementParams)` also specifies the parameters used by the measurement function set in the `MeasurementFcn` property.

Input Arguments

`filter` — Unscented Kalman filter
`trackingUKF`object

Unscented Kalman filter, specified as a `trackingUKF` object.

Example: `filter = trackingUKF`

`zmat` — Object measurements
real-valued *K*-by-*N* matrix

Measurements, specified as a real-valued *K*-by-*N* matrix. *K* is the number of candidate measurement vectors. Each row corresponds to a candidate measurement vector.*N* is the number of rows in the output of the function specified by the `MeasurementFcn` property.

Example: `[2,1;3,0]`

Data Types: `double`

`measurementParams` — Measurement function parameters
`{}` (default) | cell array

Measurement function parameters, specified as a cell array containing arguments to the measurement function specified by the `MeasurementFcn` property. Suppose you set `MeasurementFcn` to `@cameas`, and then set these values:

```
measurementParams = {frame,sensorpos,sensorpos)
```

The `distance` method internally calls the following:

```
cameas(state,frame,sensorpos,sensorvel)
```

Data Types: `cell`

Output Arguments

`dist` — Mahalanobis distances
real-valued *K*-element vector of positive values

Mahalanobis distances between candidate measurements and the predicted measurement, returned as a real-valued *K*-element vector of positive values. There is one distance value per measurement vector.

Data Types: `double`

Introduced in R2018b

predict

Class: trackingUKF

Predict unscented Kalman state vector and state error covariance matrix

Syntax

```
[xpred,Ppred] = predict(filter)
[xpred,Ppred] = predict(filter,varargin)
[xpred,Ppred] = predict( ___ ,dt)
```

Description

`[xpred,Ppred] = predict(filter)` returns the predicted state vector, `xpred`, and state error covariance matrix, `Ppred`, at the next time step based on the current time step. The predicted values overwrite the internal state vector and state error covariance matrix of the unscented Kalman filter.

`[xpred,Ppred] = predict(filter,varargin)` specifies in `varargin` input arguments of the state transition function set in the `StateTransitionFcn` property.

`[xpred,Ppred] = predict( ___ ,dt)` also specifies the time step, `dt`.

Input Arguments

`filter` — Unscented Kalman filter

`trackingUKF` object

Unscented Kalman filter, specified as a `trackingUKF` object.

Example: `filter = trackingUKF`

`varargin` — State transition function arguments

comma-separated list

State transition function arguments, specified as a comma-separated list. These arguments are the same ones that are passed into the state transition function specified by the `StateTransitionFcn` property. For example, if you set the `StateTransitionFcn` property to `@constacc`, and then call

```
[xpred,Ppred] = predict(filter,dt)
```

the `predict` method will internally call

```
state = constacc(state,dt)
```

`dt` — Time step
positive scalar

Time step, specified as a positive scalar. Units are in seconds.

Data Types: `double`

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 *a priori* estimate of the state vector propagated from the previous state. The prediction uses the state transition function specified in the `StateTransitionFcn` property.

Data Types: `double`

`Ppred` — Predicted state error covariance matrix
real-valued *M*-by-*M* matrix

Predicted state error covariance matrix, returned as a real-valued *M*-by-*M* matrix. This predicted error is the *a priori* estimate of the state error covariance matrix. `predict` uses the state transition function Jacobian specified in the `StateTransitionJacobianFcn` property.

Data Types: `double`

Introduced in R2018b

initialize

Class: trackingUKF

Initialize unscented Kalman filter

Syntax

```
initialize(filter,X,P)
initialize(filter,X,P,Name,Value)
```

Description

`initialize(filter,X,P)` initializes the unscented Kalman filter, `filter`, using the state, `X`, and the state covariance, `P`.

`initialize(filter,X,P,Name,Value)` initializes the Kalman filter properties using name-value pairs.

Note: you cannot change the size or type of properties you are initializing.

Input Arguments

`filter` — Unscented Kalman tracking filter

`Unscented Kalman filter object`

Kalman tracking filter, specified as an unscented Kalman filter object.

`X` — Initial unscented Kalman filter state

vector | matrix

Initial unscented Kalman filter state, specified as a vector or matrix.

`P` — Initial unscented Kalman filter state covariance

matrix

Initial unscented Kalman filter state covariance, specified as a matrix.

Introduced in R2018b

likelihood

Class: trackingUKF

Measurement likelihood

Syntax

```
measlikelihood = likelihood(filter,zmeas)
measlikelihood = likelihood(filter,zmeas,measparams)
```

Description

`measlikelihood = likelihood(filter,zmeas)` returns the likelihood of the measurement, `zmeas`, of an object tracked by the unscented Kalman filter, `filter`.

`measlikelihood = likelihood(filter,zmeas,measparams)` also specifies measurement parameters, `measparams`.

Input Arguments

`filter` — Unscented Kalman tracking filter
`Unscented Kalman filter object`

Unscented Kalman tracking filter, specified as an unscented Kalman filter object.

`zmeas` — Measurement of tracked object
vector | matrix

Measurement of the 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 defined in the `MeasurementFcn` property of the unscented Kalman filter, `filter`.

Output Arguments

`measlikelihood` — Likelihood of measurement
scalar

Likelihood of measurement, returned as a scalar.

Introduced in R2018b

residual

Class: trackingUKF

Measurement residual and residual covariance

Syntax

```
[zres,rescov] = residual(filterobj,zmeas)
```

Description

`[zres,rescov] = residual(filterobj,zmeas)` computes the residual, `zres`, between a measurement, `zmeas`, and a predicted measurement produced by the Kalman filter, `filterobj`. The function also returns the covariance of the residual, `zres`.

Input Arguments

`filterobj` — Unscented Kalman tracking filter

`Kalman filter object`

Unscented Kalman tracking filter, specified as a Kalman filter object.

`zmeas` — Measurement of tracked object

vector | matrix

Measurement of the tracked object, specified as a vector or matrix.

Output Arguments

`zres` — Residual between measurement and predicted measurement

matrix

Residual between measurement and predicted measurement, returned as a matrix.

`rescov` — Covariance of residuals
matrix

Covariance of the residuals, returned as a matrix.

Algorithms

- The residual is the difference between a measurement and the value predicted by the filter. The residual d is defined as $d = z - h(x)$. h is the measurement function set by the `MeasurementFcn` property, x is the current filter state, and z is the current measurement.
- The covariance of the residual, S, is computed as $S = R + R_p$. R_p is the state covariance matrix projected onto the measurement space and R is the measurement noise matrix set by the `MeasurementNoise` property.

Introduced in R2018b

radarEmission class

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.

Construction

`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 =

  1×1 cell array

    {1×1 objectDetection}


numDets =

     1
```

```
config =

  struct with fields:

             SensorIndex: 1
             IsValidTime: 1
              IsScanDone: 0
             FieldOfView: [1 5]
    MeasurementParameters: [1×1 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 class

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.

Construction

`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 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`

`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 =

  1×1 cell array

    {1×1 objectDetection}


numDets =
```

```
     1


config =

  struct with fields:

             SensorIndex: 1
             IsValidTime: 1
              IsScanDone: 1
             FieldOfView: [1 5]
    MeasurementParameters: [1×1 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.

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 2-385 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 Systems" 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 Systems" 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 Systems" for coordinate system definitions.

Data Types: `double`

Object Functions

Plotter Objects

clearData	Clear data from specific plotter of theater plot
clearPlotterData	Clear plotter data from theater plot

detectionPlotter	Create detection plotter
findPlotter	Return array of plotters associated with theater plot
orientationPlotter	Create orientation plotter
platformPlotter	Create platform plotter
trackPlotter	Create track plotter
trajectoryPlotter	Create trajectory plotter

Plotting Functions

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

Examples

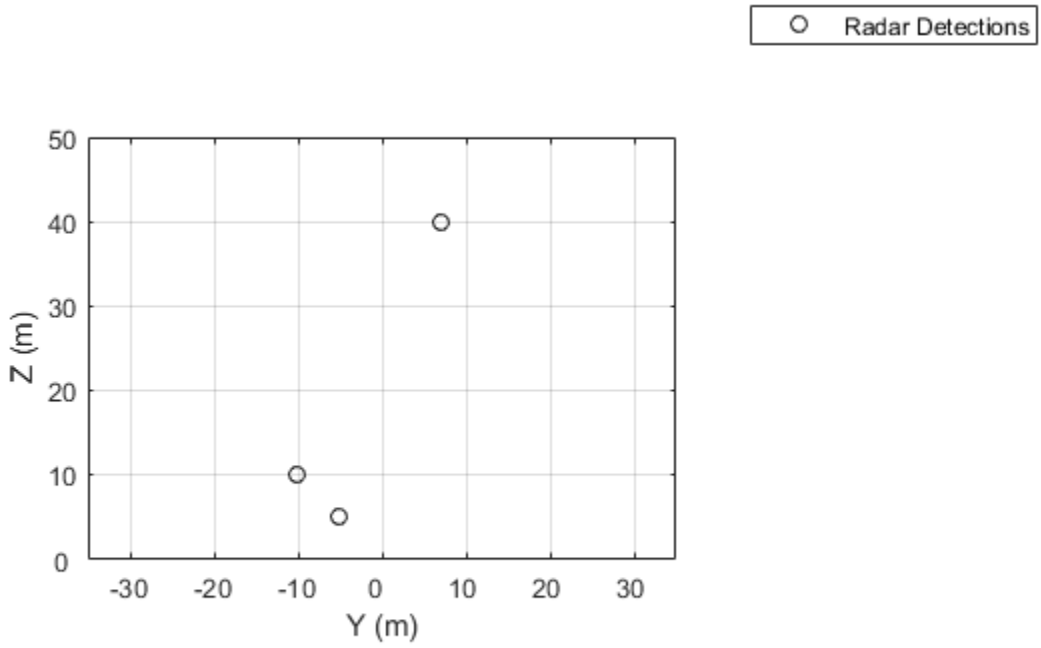
Create and Display Theater Plot

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

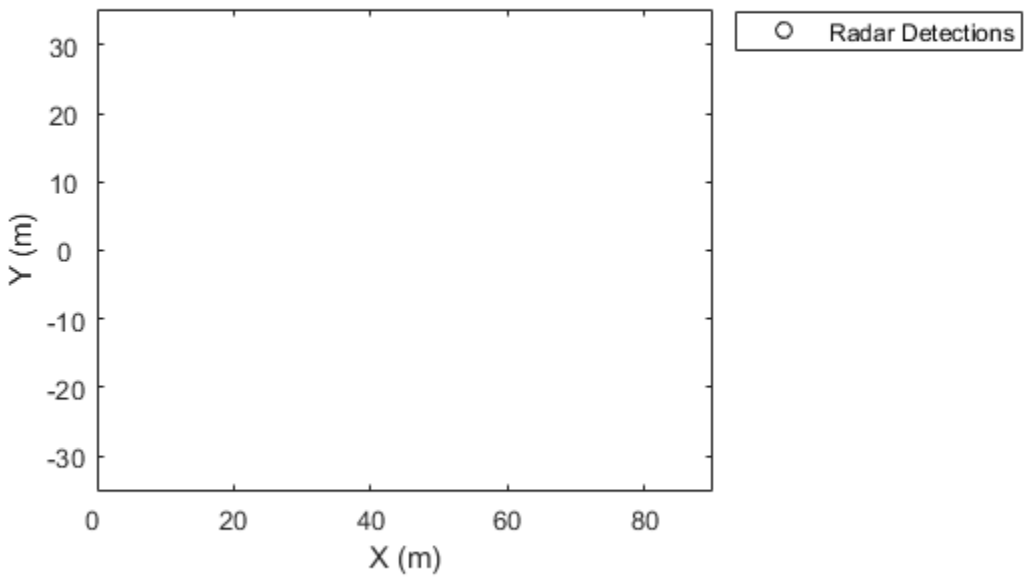
`clearPlotterData(tp)` clears data shown in the plot from all the plotters used in the theater plot, `tp`. Legend entries and coverage areas are not cleared from the plot.

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');
```

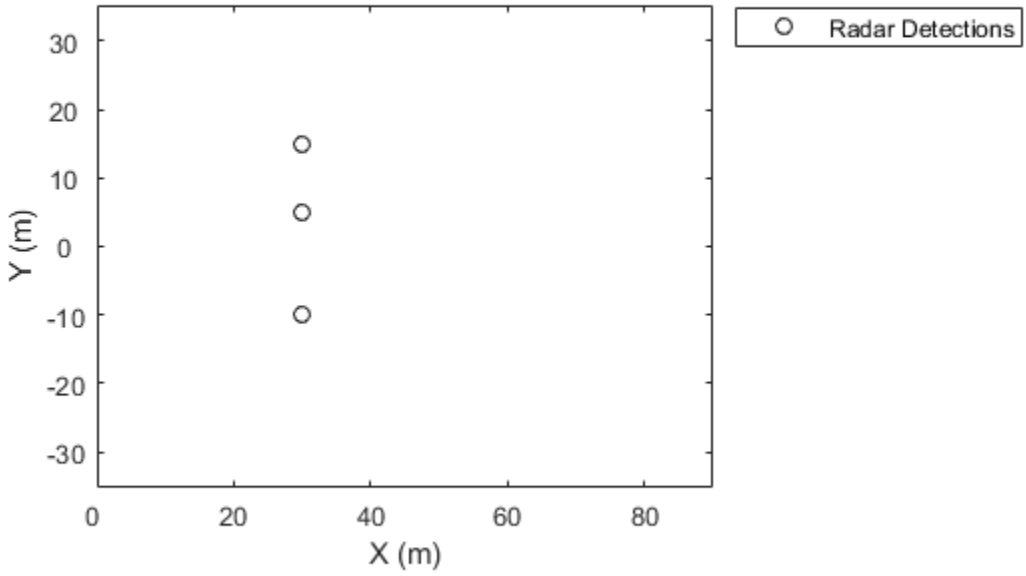


Use `findPlotter` to locate the plotter by its display name.

```
radarPlotter = findPlotter(tp,'DisplayName','Radar Detections');
```

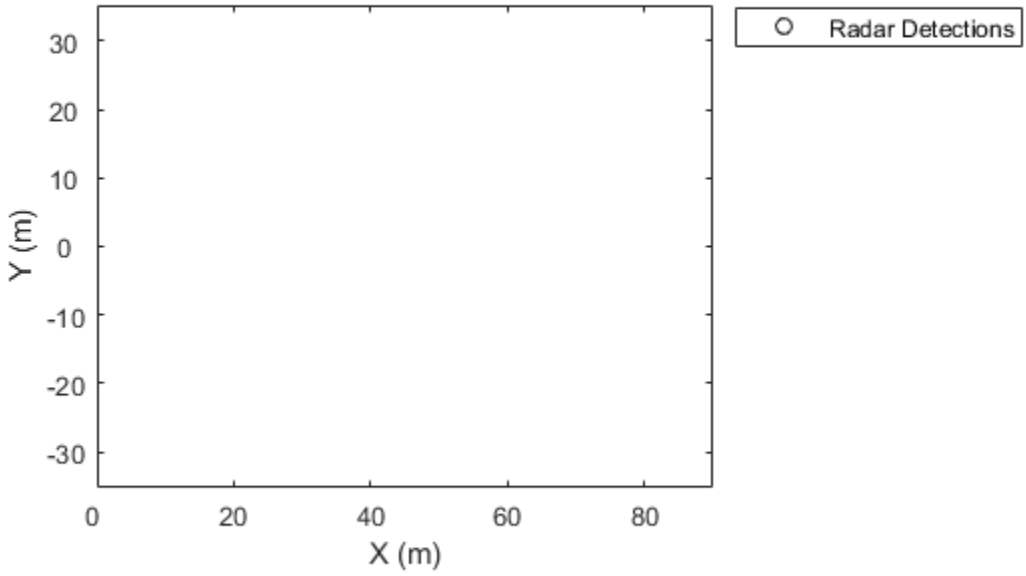
Plot three detections.

```
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

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]);
```

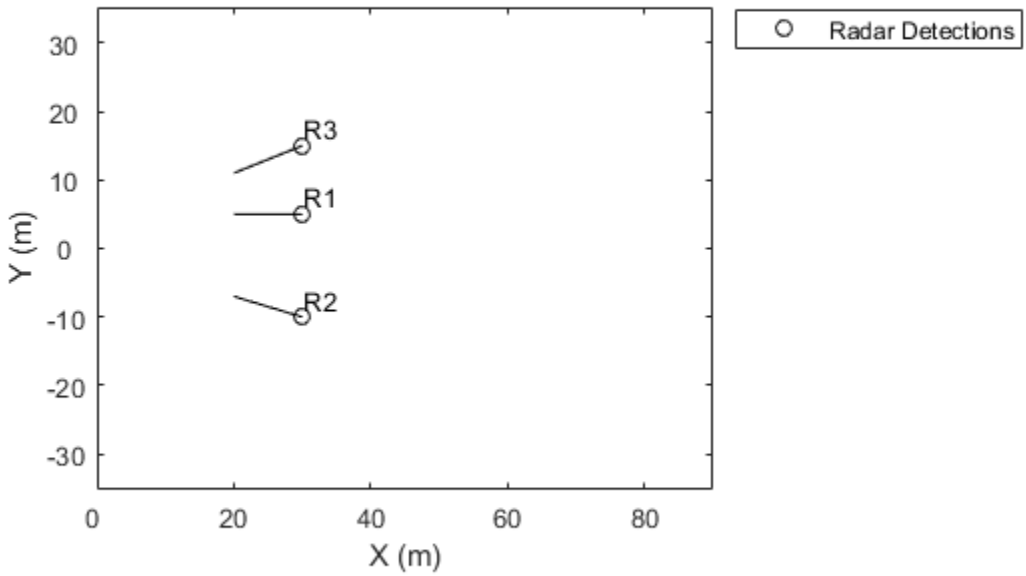
Create a detection plotter with the name `Radar Detections`.

```
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
`'+'`	Plus sign
`'o'`	Circle (default)
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'s'` or `'square'`	Square
`'d'` or `'diamond'`	Diamond

Value	Description
'v'	Downward-pointing triangle
'^'	Upward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'p' or 'pentagram'	Five-pointed star (pentagram)
'h' or 'hexagram'	Six-pointed star (hexagram)
'none'	No marker symbol

`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
`'Plotter`*N*`'` (default) | character vector | string scalar

Tag to associate with the plotter, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. The default value is `'Plotter`*N*`'`, where *N* is an integer that corresponds to the *N*th plotter associated with the `theaterPlot`.

Tags provide a way to identify plotter objects, for example when searching using `findPlotter`.

See Also

`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');
```

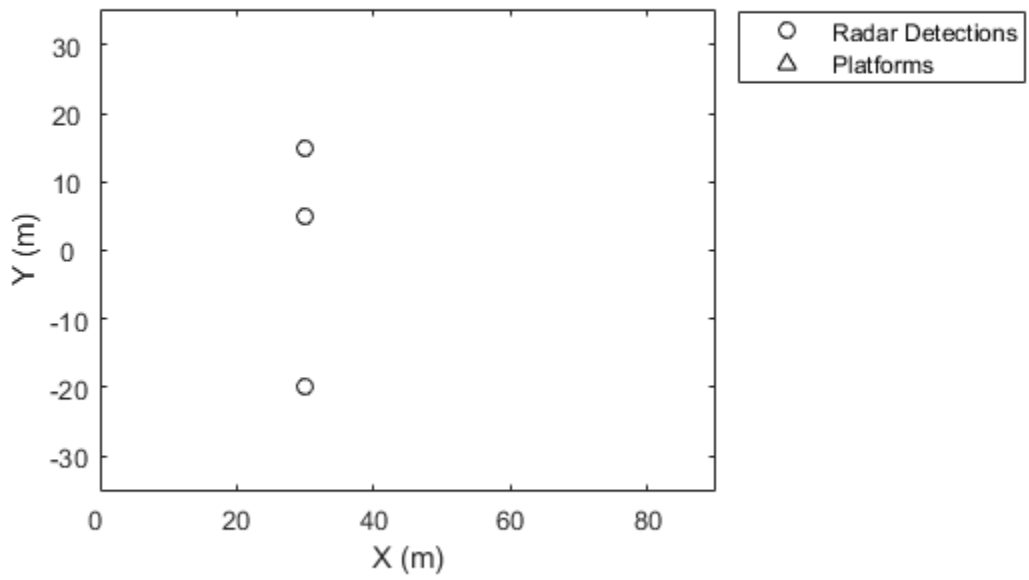
Use `findPlotter` to locate the detection plotter based on its `Tag` property.

```
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 `'Plotter`*`N`*`'`, 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.

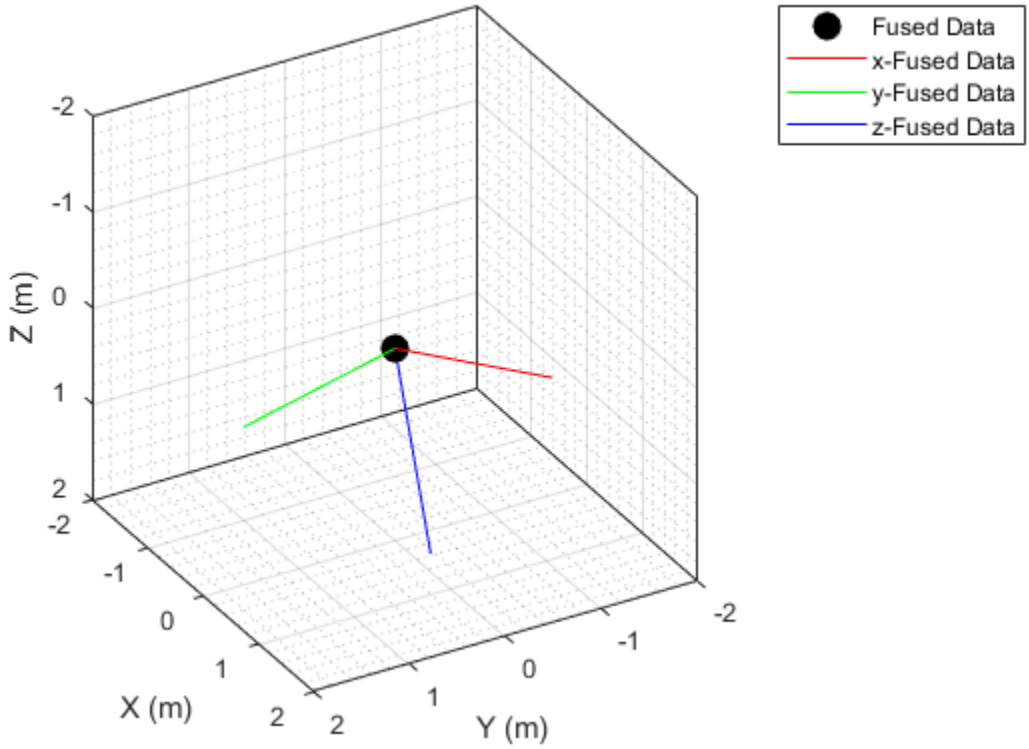
```
pose = fuse(accel,gyro);
```

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
`'+'`	Plus sign
`'o'`	Circle (default)
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'s'` or `'square'`	Square
`'d'` or `'diamond'`	Diamond
`'v'`	Downward-pointing triangle
`'^'`	Upward-pointing triangle
`'>'`	Right-pointing triangle

Value	Description
'<'	Left-pointing triangle
'p' or 'pentagram'	Five-pointed star (pentagram)
'h' or 'hexagram'	Six-pointed star (hexagram)
'none'	No marker symbol

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
`'Plotter`*N*`'` (default) | character vector | string scalar

Tag to associate with the plotter, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. The default value is `'Plotter`*N*`'`, where *N* is an integer that corresponds to the *N*th plotter associated with the `theaterPlot`.

Tags provide a way to identify plotter objects, for example when searching using `findPlotter`.

See Also

`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]);
```

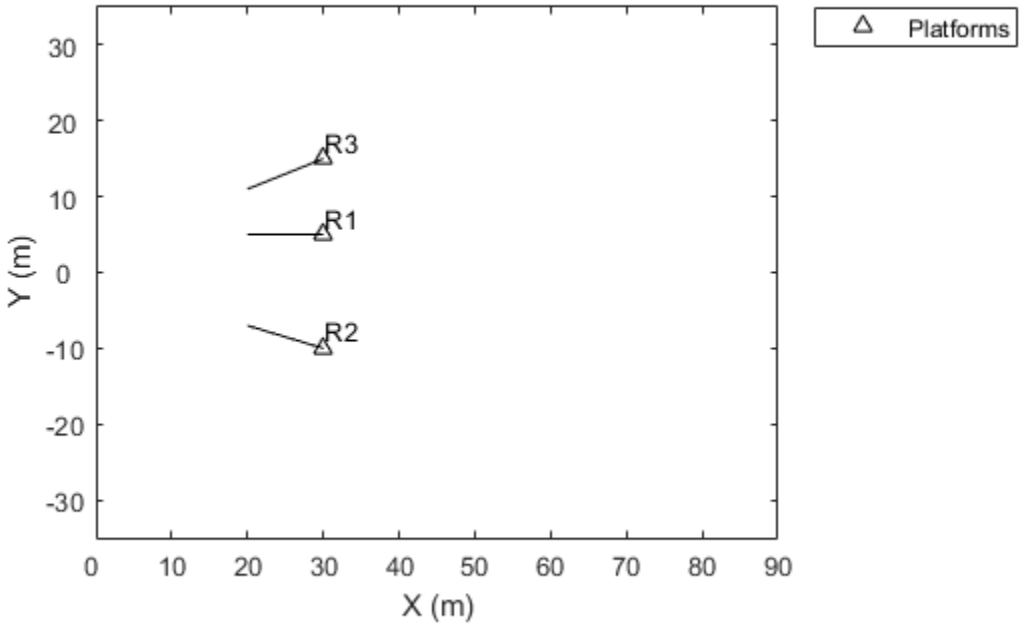
Create a platform plotter with the name `'Platforms'`.

```
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
`'+'`	Plus sign
`'o'`	Circle
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'s'` or `'square'`	Square
`'d'` or `'diamond'`	Diamond
`'v'`	Downward-pointing triangle
`'^'`	Upward-pointing triangle (default)
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'p'` or `'pentagram'`	Five-pointed star (pentagram)

Value	Description
`'h'` or `'hexagram'`	Six-pointed star (hexagram)
`'none'`	No marker symbol

`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` | `plotPatform` | `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.

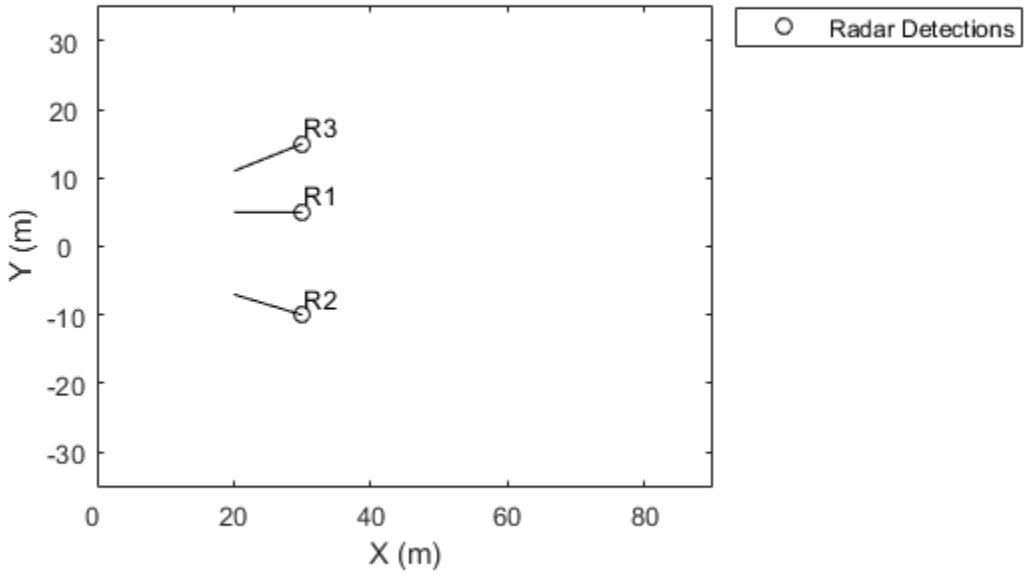
```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35],'ZLim',[1,10]);
```

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.

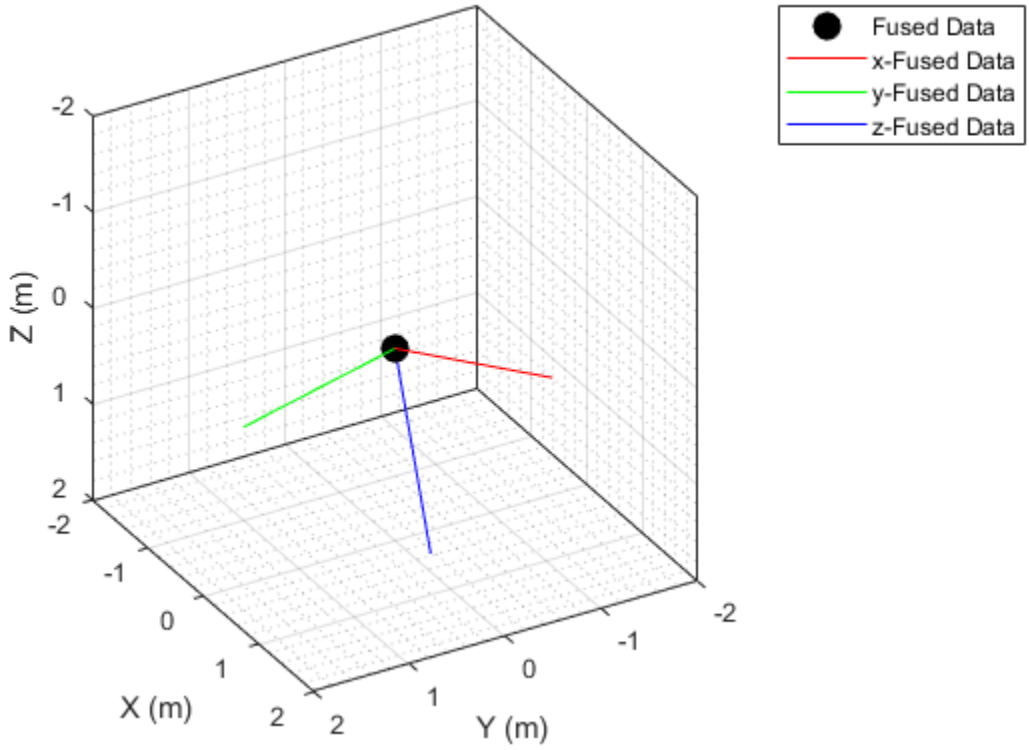
```
pose = fuse(accel,gyro);
```

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)
```

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.

Examples

Create and Update Theater Plot Platforms

Create a theater plot.

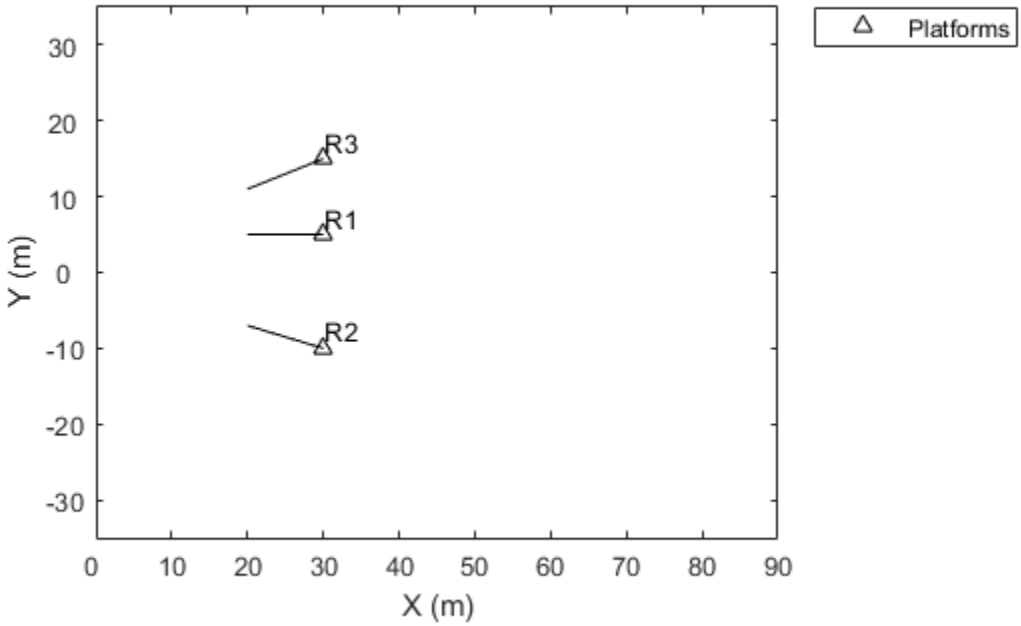
```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35],'ZLim',[1,10]);
```

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
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.

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)
```

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.

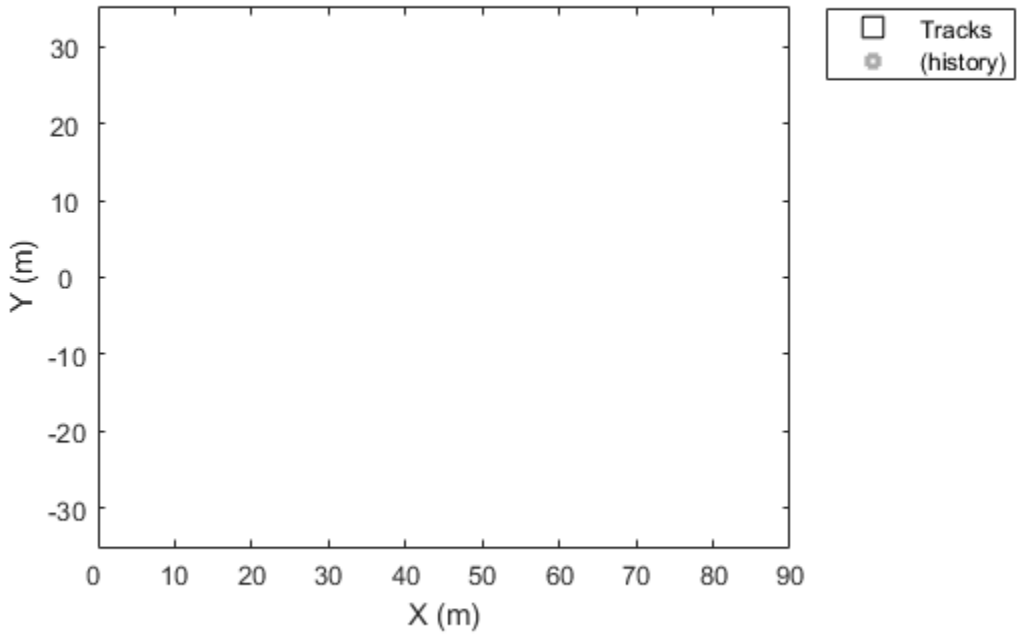
If `trackIDs` is omitted when `'ConnectHistory'` is `'on'`, then the track identifiers are derived from the labels input instead. The `trackIDs` input is ignored when `'ConnectHistory'` is `'off'`.

Examples

Plot Tracks in Theater Plot

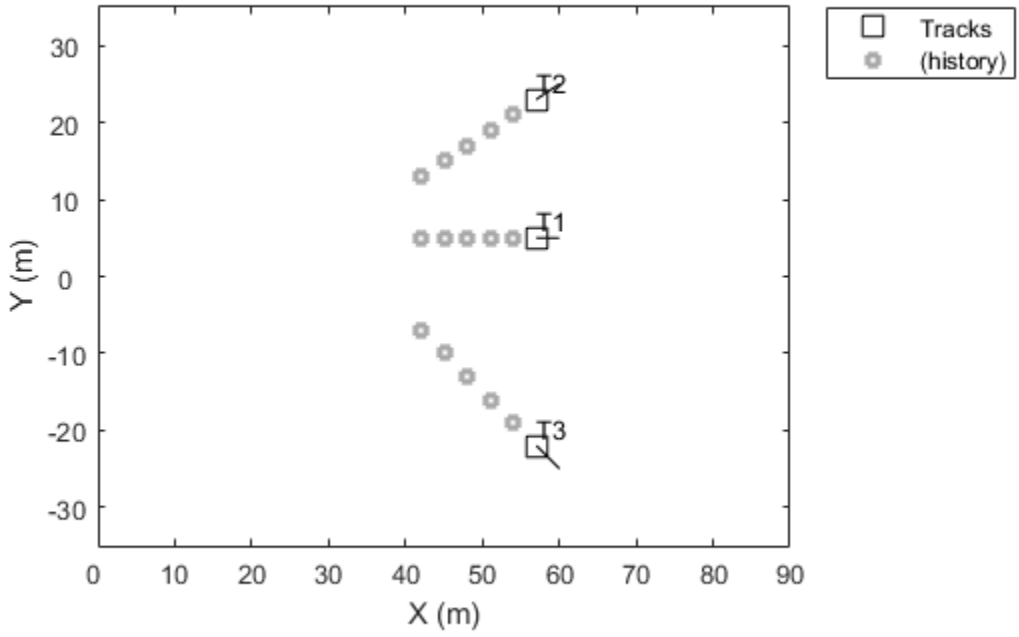
Create a theater plot. Create a track plotter with `DisplayName` set to `'Tracks'` and with `HistoryDepth` set to `5`.

```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35]);
tPlotter = trackPlotter(tp,'DisplayName','Tracks','HistoryDepth',5);
```

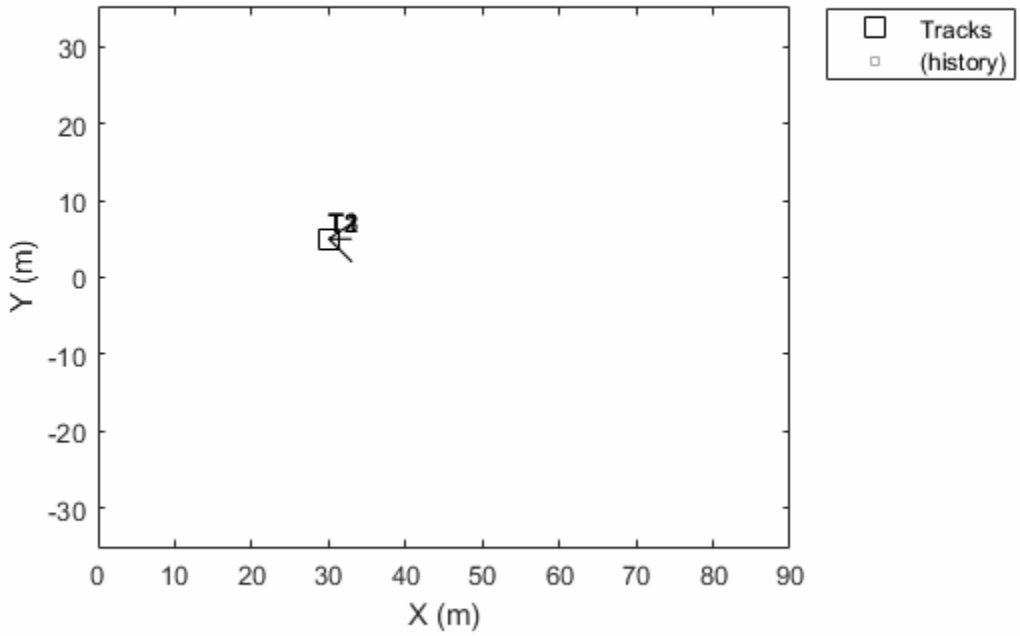


Update the track plotter with three tracks labeled `'T1'`, `'T2'`, and `'T3'` with start positions in units of meters all starting at (30, 5, 1) with corresponding velocities (in m/s) of (3, 0, 1), (3, 2, 2) and (3, -3, 5), respectively. Update the tracks with the velocities for ten iterations.

```
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'`.

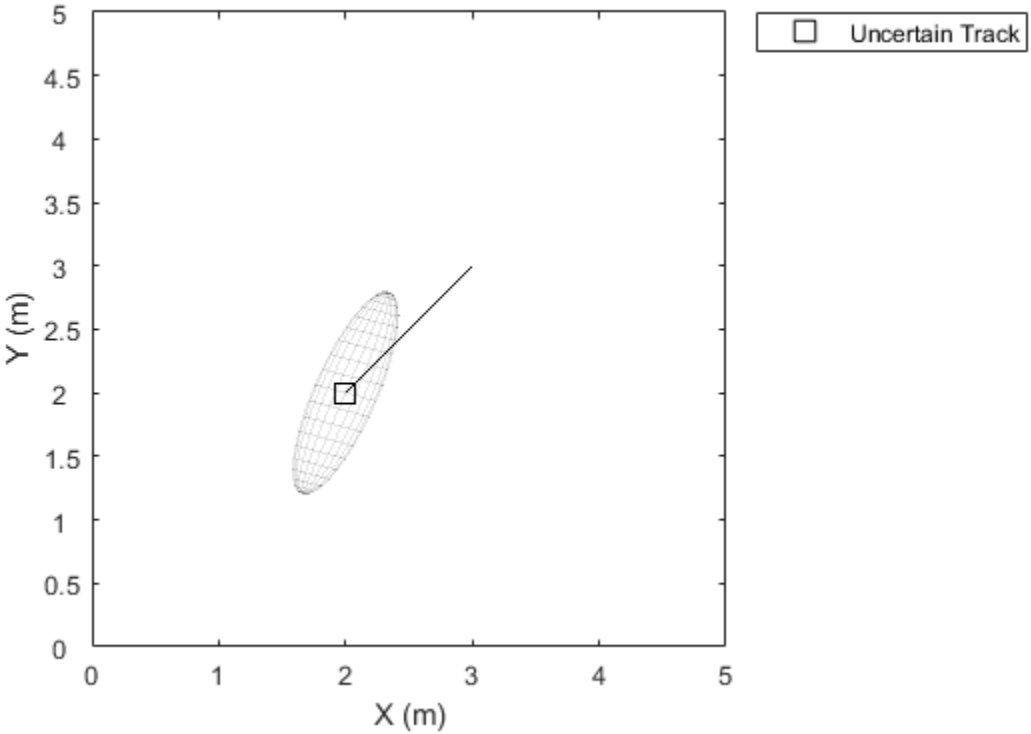
```
tp = theaterPlot('Xlim',[0 5],'Ylim',[0 5]);
tPlotter = trackPlotter(tp,'DisplayName','Uncertain Track');
```

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
covariences = randn(3,3);
```

Plot the track with the covariances plotted as an ellipsoid.

```
plotTrack(tPlotter,positions,velocities,covariences)
```



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.

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.

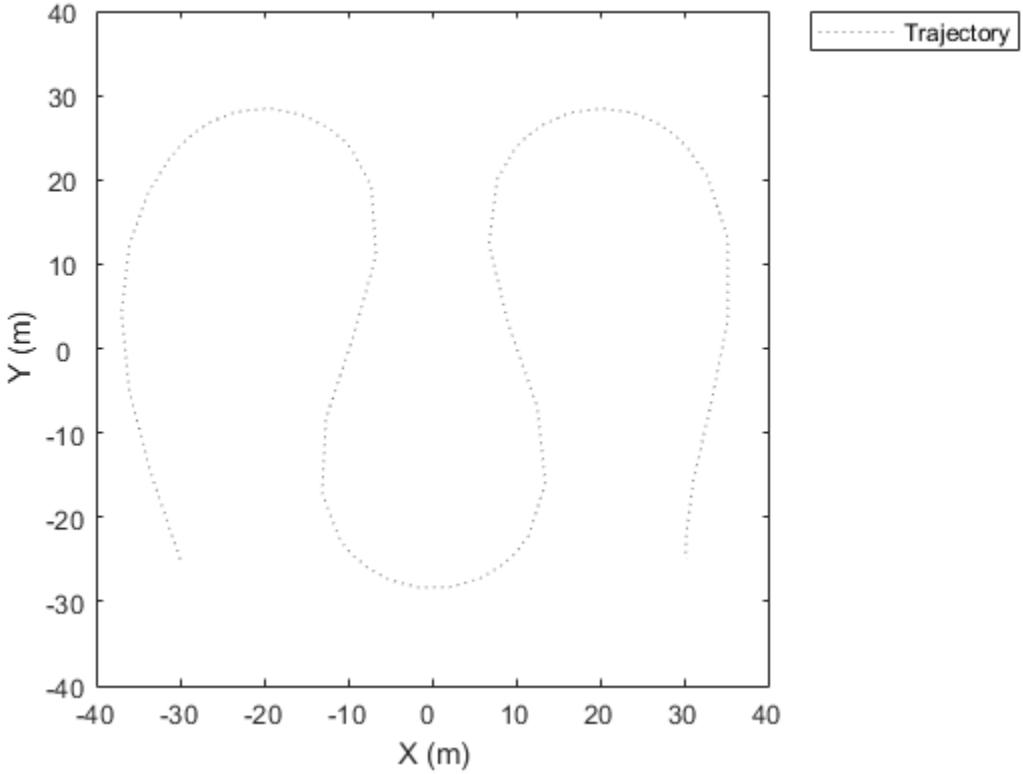
```
target = platform(ts);
traj = waypointTrajectory('Waypoints',wayPoints,'TimeOfArrival',elapsedTime);
target.Trajectory = traj;
```

Record the tracking scenario to retrieve the platform's trajectory.

```
r = record(ts);
pposes = [r(:).Poses];
pposition = vertcat(pposes.Position);
```

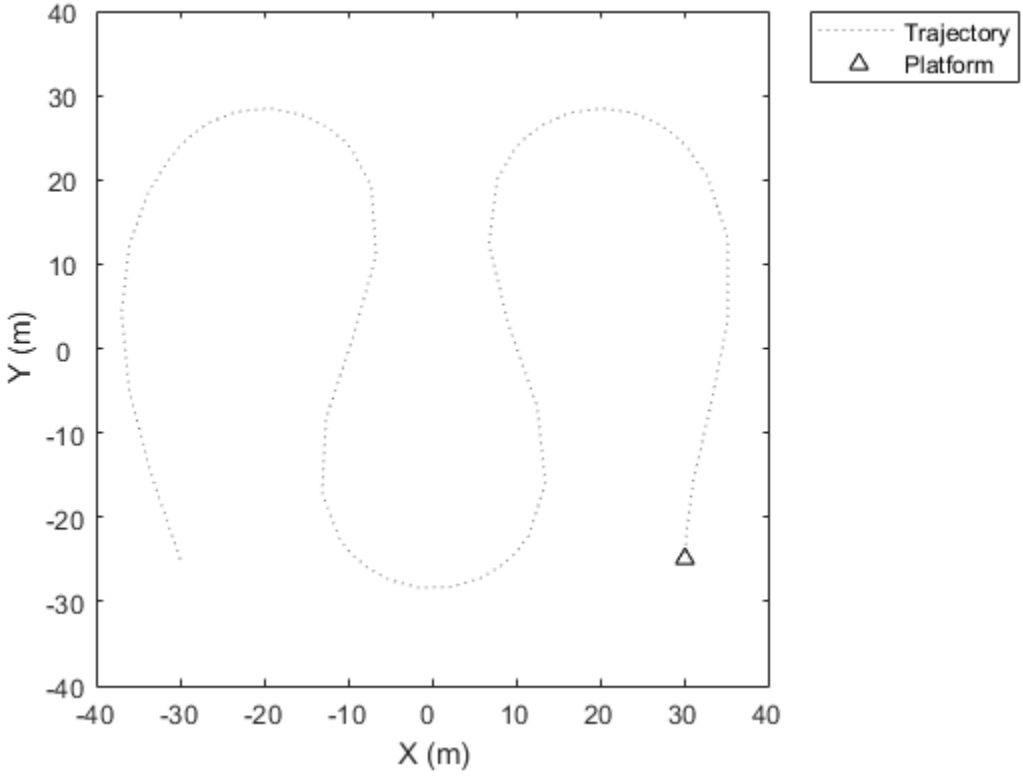
Create a theater plot to display the recorded trajectory.

```
tp = theaterPlot('XLim',[-40 40],'YLim',[-40 40]);
trajPlotter = trajectoryPlotter(tp,'DisplayName','Trajectory');
plotTrajectory(trajPlotter,{pposition})
```

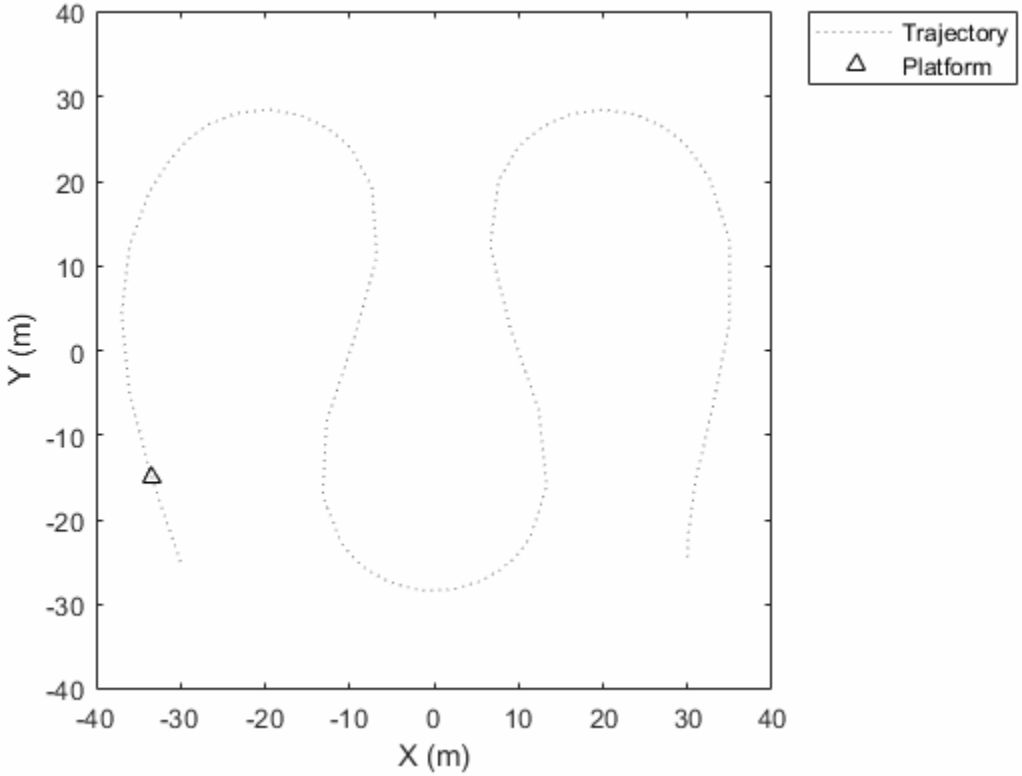


Animate using the `platformPlotter`.

```
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`.

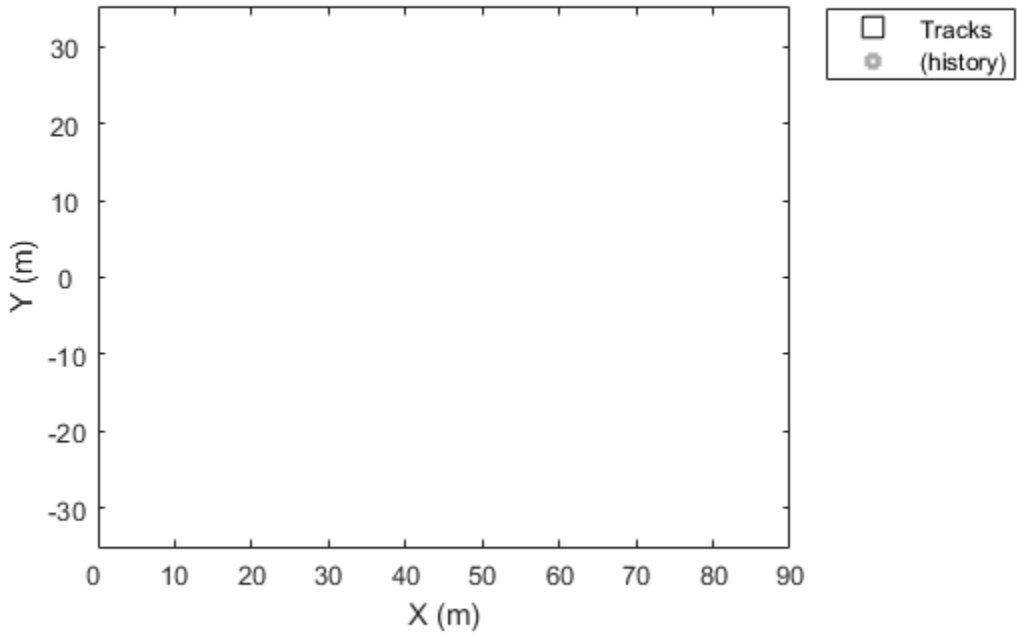
`tPlotter = trackPlotter(tp,Name,Value)` creates a track plotter with additional options specified by one or more `Name,Value` pair arguments.

Examples

Plot Tracks in Theater Plot

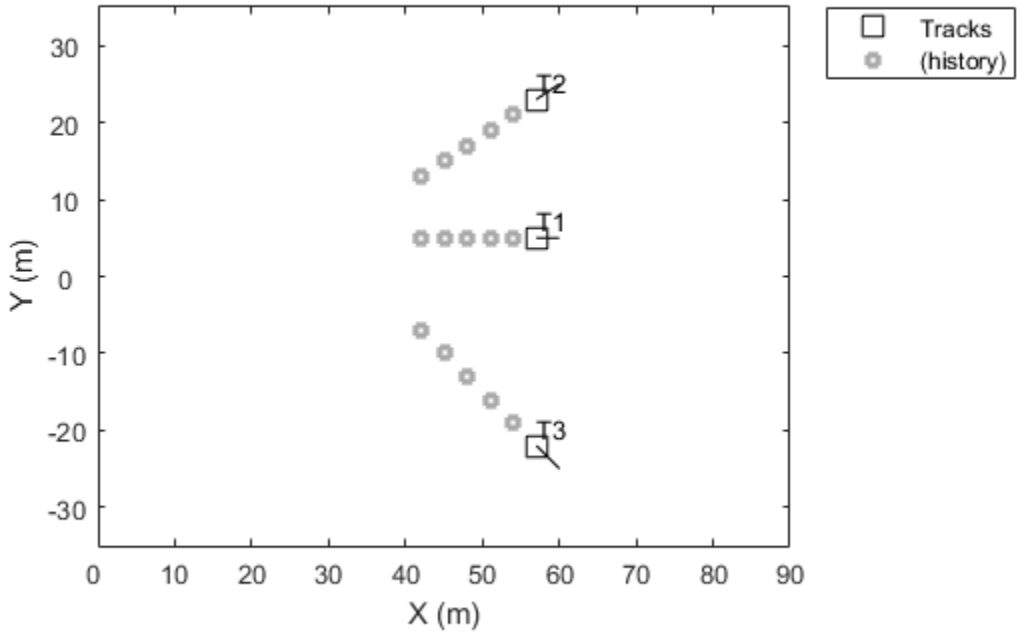
Create a theater plot. Create a track plotter with `DisplayName` set to `'Tracks'` and with `HistoryDepth` set to `5`.

```
tp = theaterPlot('XLim',[0,90],'YLim',[-35,35]);
tPlotter = trackPlotter(tp,'DisplayName','Tracks','HistoryDepth',5);
```

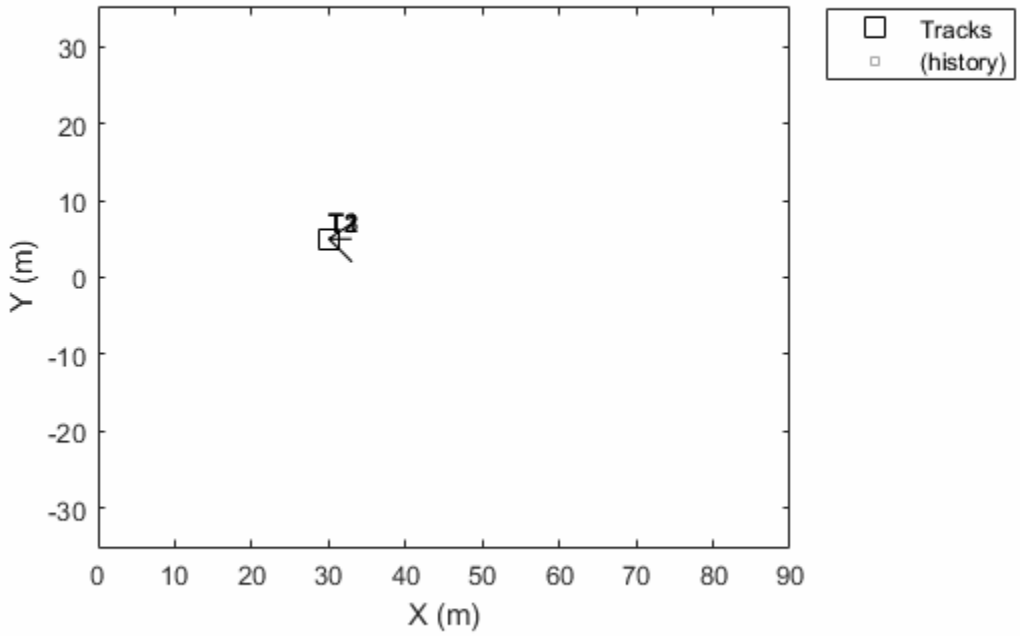


Update the track plotter with three tracks labeled `'T1'`, `'T2'`, and `'T3'` with start positions in units of meters all starting at (30, 5, 1) with corresponding velocities (in m/s) of (3, 0, 1), (3, 2, 2) and (3, -3, 5), respectively. Update the tracks with the velocities for ten iterations.

```
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
`'+'`	Plus sign
`'o'`	Circle
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'s'` or `'square'`	Square (default)
`'d'` or `'diamond'`	Diamond
`'v'`	Downward-pointing triangle
`'^'`	Upward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'p'` or `'pentagram'`	Five-pointed star (pentagram)
`'h'` or `'hexagram'`	Six-pointed star (hexagram)
`'none'`	No marker symbol

`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.

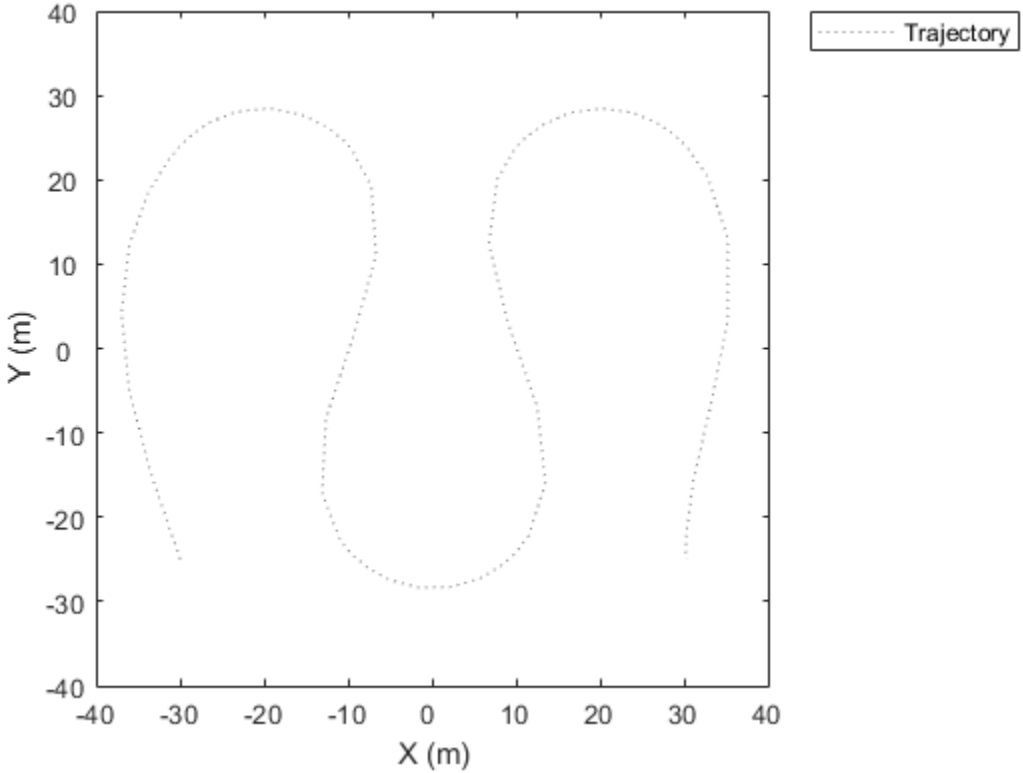
```
target = platform(ts);
traj = waypointTrajectory('Waypoints',wayPoints,'TimeOfArrival',elapsedTime);
target.Trajectory = traj;
```

Record the tracking scenario to retrieve the platform's trajectory.

```
r = record(ts);
pposes = [r(:).Poses];
pposition = vertcat(pposes.Position);
```

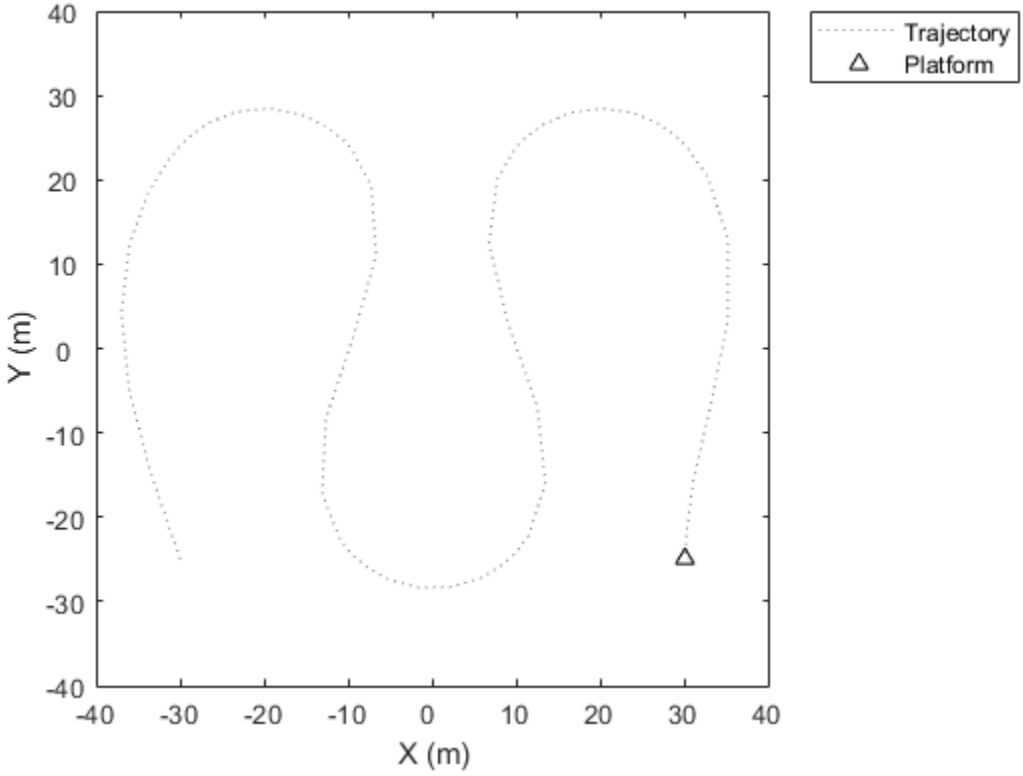
Create a theater plot to display the recorded trajectory.

```
tp = theaterPlot('XLim',[-40 40],'YLim',[-40 40]);
trajPlotter = trajectoryPlotter(tp,'DisplayName','Trajectory');
plotTrajectory(trajPlotter,{pposition})
```

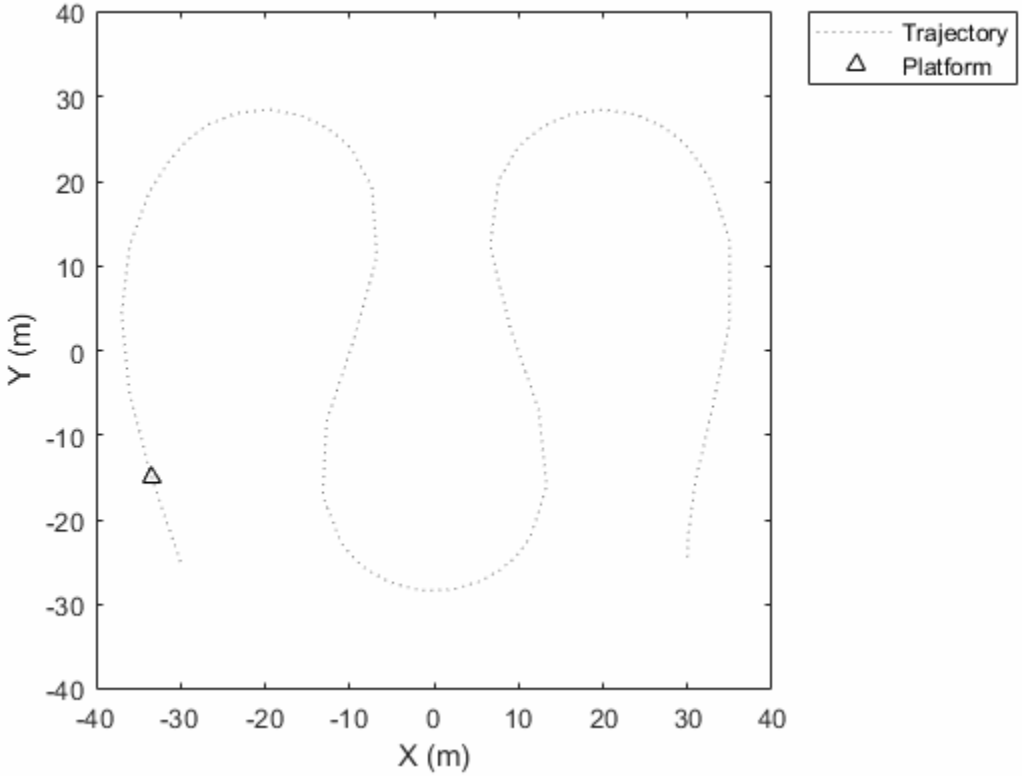


Animate using the `platformPlotter`.

```
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
`'Plotter`*`N`*`'` (default) | character vector | string scalar

Tag to associate with the plotter, specified as the comma-separated pair consisting of `'Tag'` and a character vector or string scalar. The default value is `'Plotter`*N*`'`, where *N* is an integer that corresponds to the *N*th plotter associated with the `theaterPlot`.

Tags provide a way to identify plotter objects, for example when searching using `findPlotter`.

See Also

`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 `'` and `''` 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

predict	Predict state and state estimation error covariance
correct	Correct state and state estimation error covariance
correctjpda	Correct state and state estimation error covariance using JPDA
distance	Distances between measurements and predicted measurements
residual	Measurement residual and residual noise
likelihood	Likelihood of measurement
clone	Copy filter for object tracking

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]);
```

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

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,measuremntFcn,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,measuremntFcn,state)` specifies the `StateTranstionFcn`, `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.

- If `HasAdditiveProcessNoise` is `true`, specify the function using one of these syntaxes:

  ```
  x(k) = transitionfcn(x(k-1))
  ```

  ```
  x(k) = transitionfcn(x(k-1),parameters)
  ```

  where `x(k)` is the state at time `k`. The `parameters` term stands for all additional arguments required by the state transition function.
- 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),parameters)
```

where `x(k)` is the state at time `k`, and `w(k)` is a value for the process noise at time `k`. The `parameters` argument stands for all additional arguments required by the state transition function.

Example: `@constacc`

Dependencies

This parameter depends on the `HasAdditiveNoise` 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` method. 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`

Dependencies

This parameter depends on the `HasAdditiveNoise` property.

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` method. 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`

Dependencies

This parameter depends on the `HasAdditiveNoise` property.

`HasAdditiveMeasurmentNoise` — 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

predict	Predict state and state estimation error covariance
correct	Correct state and state estimation error covariance
correctjpda	Correct state and state estimation error covariance using JPDA
distance	Distances between measurements and predicted measurements
residual	Measurement residual and residual noise
likelihood	Likelihood of measurement
clone	Copy filter for object tracking

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
```

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

Guassian-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 `ModelProbabilties` 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` method. 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

predict	Predict state and state estimation error covariance
correct	Correct state and state estimation error covariance
correctjpda	Correct state and state estimation error covariance using JPDA
distance	Distances between measurements and predicted measurements
residual	Measurement residual and residual noise
clone	Copy filter for object tracking
initialize	Initialize state and covariance of 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.

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

`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

`{trackingEKF,trackingEKF,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

`@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.
- 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 and column must sum to 1.

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` method.

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.

Data Types: `single` | `double`

Object Functions

predict	Predict state and state estimation error covariance
correct	Correct state and state estimation error covariance
correctjpda	Correct state and state estimation error covariance using JPDA
distance	Distances between measurements and predicted measurements
residual	Measurement residual and residual noise
clone	Copy filter for object tracking
initialize	Initialize state and covariance of 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
```

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`
- `MeasurementJacobianFcn` - `@cvmeasmscjac`
- `HasAdditiveProcessNoise` - `false`
- `HasAdditiveMeasurementNoise` - `true`

Creation

Syntax

```
mscekf = trackingMSCEKF
mscekf = trackingMSCEKF(Name,Value)
```

Description

`mscekf = 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.

`mscekf = 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. *M* is either 4 for 2-D tracking or 6 for 3-D tracking.

Example: `[az;azRate;1/r;rDot/r]` for 2-D tracking and `[az;omega;el;elRate;1/r;rDot/r]` for 3-D tracking

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` method. 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
`@cvmeasmsc` (default)

This property is read-only.

Measurement model function, specified as a function handle, `@cvmeasmsc`. 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 `@cvmeasmsc`.

Data Types: `function_handle`

`MeasurementJacobianFcn` — Jacobian of measurement function
`@cvmeasmscjac`

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 `@cvmeasmscjac`.

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` method.

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
correct	Correct state and state estimation error covariance
correctjpda	Correct state and state estimation error covariance using JPDA

distance	Distances between measurements and predicted measurements
residual	Measurement residual and residual noise
clone	Copy filter for object tracking
initialize	Initialize state and covariance of 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;
azRate = 0;
r = 1000;
rDot = 10;
el = 0.3;
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]; %degrees
correct(mscekf,meas);
```

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,measuremntFcn,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,measuremntFcn,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)
```

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` method. 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,varar
```

```
function [likelihood,distance] = measurementLikelihoodFcn(pf,predictedParticles,measure
```

`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

predict	Predict state and state estimation error covariance
correct	Correct state and state estimation error covariance
correctjpda	Correct state and state estimation error covariance using JPDA
distance	Distances between measurements and predicted measurements
residual	Measurement residual and residual noise
clone	Copy filter for object tracking

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`.

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 `ConfirmationThreshold` and `DeletionThreshold` 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 `ConfirmationThreshold(2)` and `DeletionThreshold(2)`. 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 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  0]. Confirmation Flag: 0. Deletion Flag: 0
History: [1  0  1  0  0  0  0]. Confirmation Flag: 0. Deletion Flag: 0
History: [0  1  0  1  0  0  0]. Confirmation Flag: 0. Deletion Flag: 0
History: [1  0  1  0  1  0  0]. Confirmation Flag: 1. Deletion Flag: 0
```

Update the logic with a miss six times. The deletion flag is `true` by the end of the fifth update, because six misses (*Md*) are counted in the most recent seven updates (*Nd*).

```
for i = 1:6
    miss(historyLogic);

    history = historyLogic.History;
    confFlag = checkConfirmation(historyLogic);
    delFlag = checkDeletion(historyLogic);
    disp(['History: [',num2str(history),']. Confirmation Flag: ',num2str(confFlag), ..
        '. Deletion Flag: ',num2str(delFlag)']);
end
```

```
History: [0  1  0  1  0  1  0]. Confirmation Flag: 0. Deletion Flag: 0
History: [0  0  1  0  1  0  1]. Confirmation Flag: 0. Deletion Flag: 0
History: [0  0  0  1  0  1  0]. Confirmation Flag: 0. Deletion Flag: 0
History: [0  0  0  0  1  0  1]. Confirmation Flag: 0. Deletion Flag: 0
History: [0  0  0  0  0  1  0]. Confirmation Flag: 0. Deletion Flag: 1
History: [0  0  0  0  0  0  1]. Confirmation Flag: 0. Deletion Flag: 1
```

References

[1] Blackman, S., and R. Popoli. *Design and Analysis of Modern Tracking Systems*. Boston, MA: Artech House, 1999.

Extended Capabilities

C/C++ Code Generation

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

See Also

`trackScoreLogic` | `trackerGNN`

Topics

"Introduction to Track Logic"

Introduced in R2018b

checkConfirmation

Check if track should be confirmed

Syntax

```
tf = checkConfimation(historyLogic)
tf = checkConfimation(scoreLogic)
```

Description

`tf = checkConfimation(historyLogic)` returns a flag that is `true` when at least *Mc* out of *Nc* recent updates of the track history logic object `historyLogic` are `true`.

`tf = checkConfimation(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 *Mc* and *Nc* as the vector [2 3]. Specify deletion threshold values *Md* and *Nd* 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 (*Mc*).

```
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 (*Mc*) are counted in the most recent three updates (*Nc*).

```
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

`checkDeletion` | `hit` | `miss` | `output`

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 *Md* out of *Nd* recent updates of the track history logic object `historyLogic` are `false`.

`tf = checkDeletion(historyLogic,tentativeTrack,age)` returns a flag that is `true` when the track is tentative and there are not enough detections to allow it to confirm. Use the logical flag `tentativeTrack` to indicate if the track is tentative and provide `age` as a numerical 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 *Mc* and *Nc* as the vector [2 3]. Specify deletion threshold values *Md* and *Nd* 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 (*Mc*).

```
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 (*Mc*) are counted in the most recent three updates (*Nc*).

```
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 *Mc* and *Nc* as the vector [2 3]. Specify deletion threshold values *Md* and *Nd* 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 (*Nc*) is less than 2 (*Mc*).

```
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 (*Nm*) is less than 4 (*Mc*).

```
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
`checkConfirmation` | `hit` | `miss` | `output`

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

`mergeScores` | `sync`

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

`checkConfirmation` | `checkDeletion` | `init` | `miss`

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 (*azimuth resolution in radians*) * (*range*) * (*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

`checkConfirmation` | `checkDeletion` | `hit` | `miss`

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

`checkConfirmation` | `checkDeletion` | `hit` | `init`

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 = 1×2

     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 = 1×2

   11.6699   11.6699
```

Update the logic with a miss, then get the updated scores.

```
miss(scoreLogic)
s = output(scoreLogic)
```

```
s = 1×2

    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
`checkConfirmation` | `checkDeletion` | `hit` | `miss`

Introduced in R2018b

reset

Reset state of track logic

Syntax

```
reset(logic)
```

Description

`reset(logic)` resets the track logic object, `logic`.

Examples

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 = 1×2

     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 = 1×2
```

```
   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 = 1×2

     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

`init`

Introduced in R2018b

3

System Objects in Sensor Fusion and Tracking Toolbox

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? (MATLAB).

Creation

Syntax

```
altimeter = altimeterSensor
altimeter = altimeterSensor(Name,Value)
```

Description

`altimeter = altimeterSensor` returns a System object, `altimeter`, that simulates altimeter readings.

`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 (MATLAB).

`UpdateRate` — 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/√Hz)

`0` (default) | nonnegative scalar

Power spectral density of sensor noise in m/√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

Syntax

```
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 NED coordinate system (m)

N-by-3 matrix

Position of sensor in the local NED 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 NED coordinate system (m)

N-element column vector

Altitude of sensor relative to the local NED 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`.

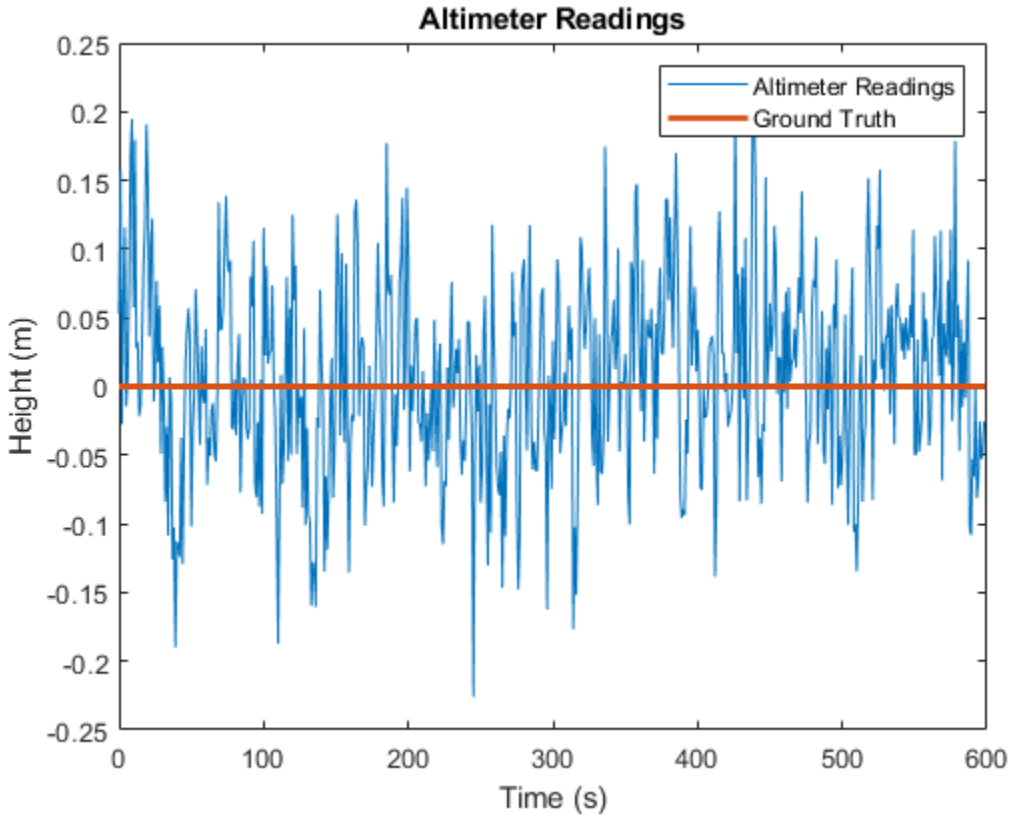
```
Fs = 1;
duration = 60*10;
numSamples = duration*Fs;


altimeter = altimeterSensor('UpdateRate',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? (MATLAB).

Creation

Syntax

```
FUSE = ahrsfilter
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(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 (MATLAB).

`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 scalar

Decimation factor by which to reduce the input sensor data rate as part of the fusion algorithm, specified as a positive integer scalar.

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 ($(m/s^2)^2$)
`0.00019247` (default) | positive real scalar

Variance of accelerometer signal noise in $(m/s^2)^2$, specified as a positive real scalar.

Tunable: Yes

Data Types: `single` | `double`

`MagnetometerNoise` — Variance of magnetometer signal noise (μT^2)
`0.1` (default) | positive real scalar

Variance of magnetometer signal noise in μT^2, specified as a positive real scalar.

Tunable: Yes

Data Types: `single` | `double`

`GyroscopeNoise` — Variance of gyroscope signal noise ((rad/s)²)

`9.1385e-5` (default) | positive real scalar

Variance of gyroscope signal noise in $(rad/s)^2$, specified as a positive real scalar.

Tunable: Yes

Data Types: `single` | `double`

`GyroscopeDriftNoise` — Variance of gyroscope offset drift ((rad/s)²)

`3.0462e-13` (default) | positive real scalar

Variance of gyroscope offset drift in $(rad/s)^2$, specified as a positive real scalar.

Tunable: Yes

Data Types: `single` | `double`

`LinearAccelerationNoise` — Variance of linear acceleration noise (m/s²)²

`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 `LinearAcclerationDecayFactor` to a lower value. If linear acceleration changes slowly, set `LinearAcclerationDecayFactor` 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 (μT^2)
`0.5` (default) | real finite positive scalar

Variance of magnetic disturbance noise in μT^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   0.000006092348396                   0                   0
                   0                   0   0.000006092348396                   0
                   0                   0                   0   0.000076154354947
                   0                   0                   0                   0   0.00
                   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
   0.009623610000000                  0                   0                   0
                  0   0.009623610000000                   0                   0
                  0                   0   0.009623610000000                   0
                  0                   0                   0   0.600000000000000
                  0                   0                   0                   0   0.60
                  0                   0                   0                   0
```

The initial process covariance matrix accounts for the error in the process model.

Data Types: `single` | `double`

`ExpectedMagneticFieldStrength` — Expected estimate of magnetic field strength (μT)
`50` (default) | real positive scalar

Expected estimate of magnetic field strength in μT, 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^2)

N-by-3 matrix

Accelerometer readings in the sensor body coordinate system in m/s^2, 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 NED 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 NED coordinate system to a body coordinate system, returned as quaternions or an array. The size and type of `orientation` depends on whether the OrienationFormat 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 NED 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)
```

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 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);
```

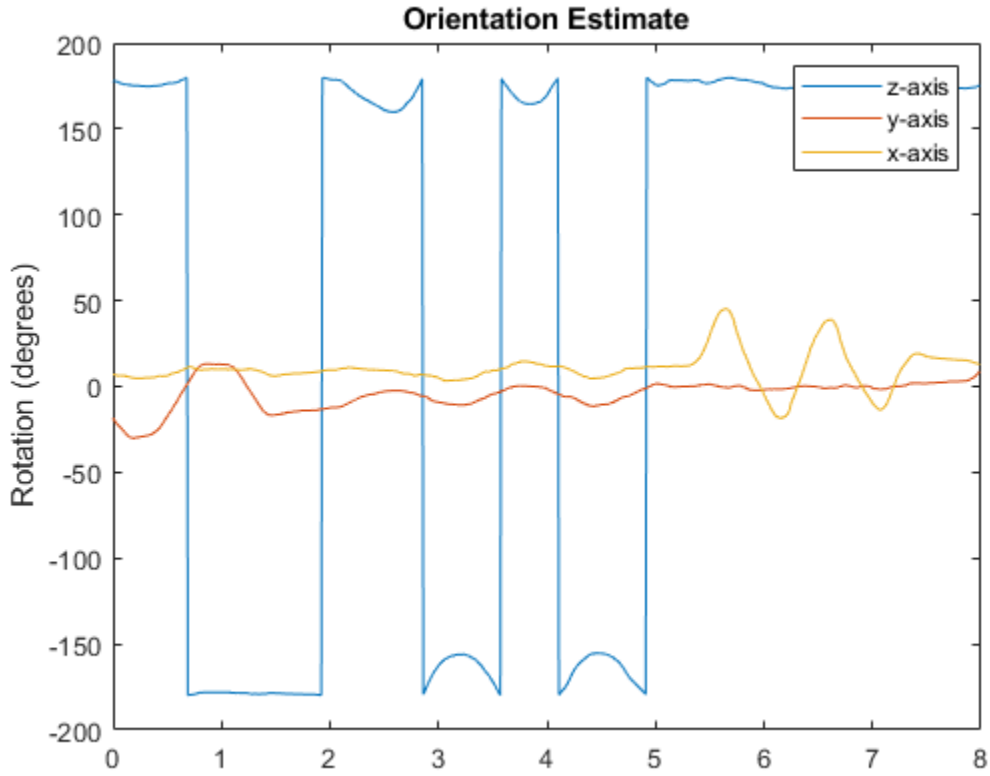
Pass the accelerometer readings, gyroscope readings, and magnetometer readings to the `ahrsfilter` object, `fuse`, to output an estimate of the sensor body orientation over time. By default, the orientation is output as a vector of quaternions.

```
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);
```

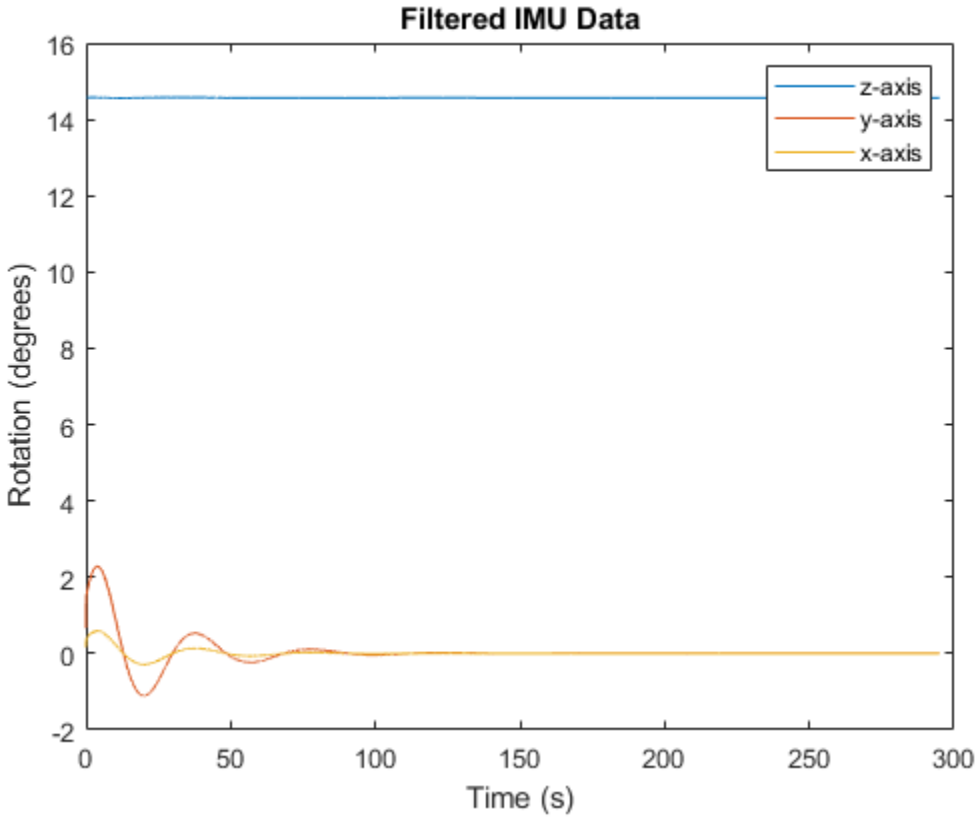
The `ahrsfilter` uses magnetic field strength to stabilize its orientation against the assumed constant magnetic field of the Earth. However, there are many natural and man-made objects which output magnetic fields and can confuse the algorithm. To account for the presence of transient magnetic fields, 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.

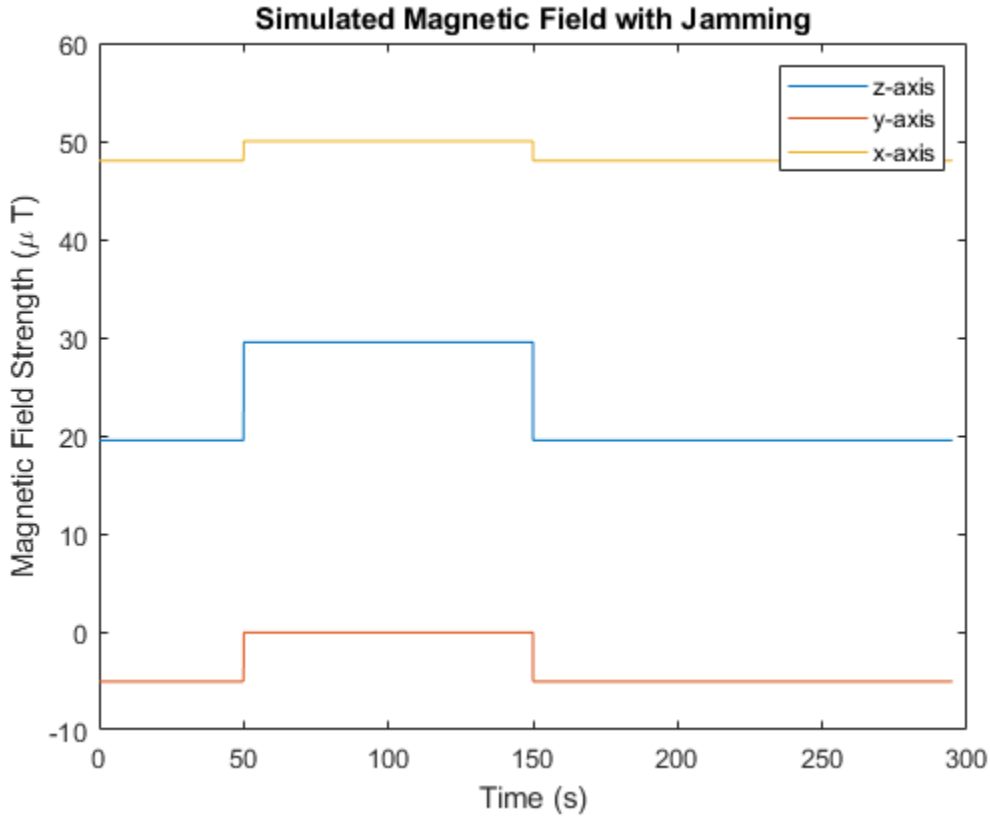
```
orientation = FUSE(accelReadings,gyroReadings,magReadings);

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)')
```

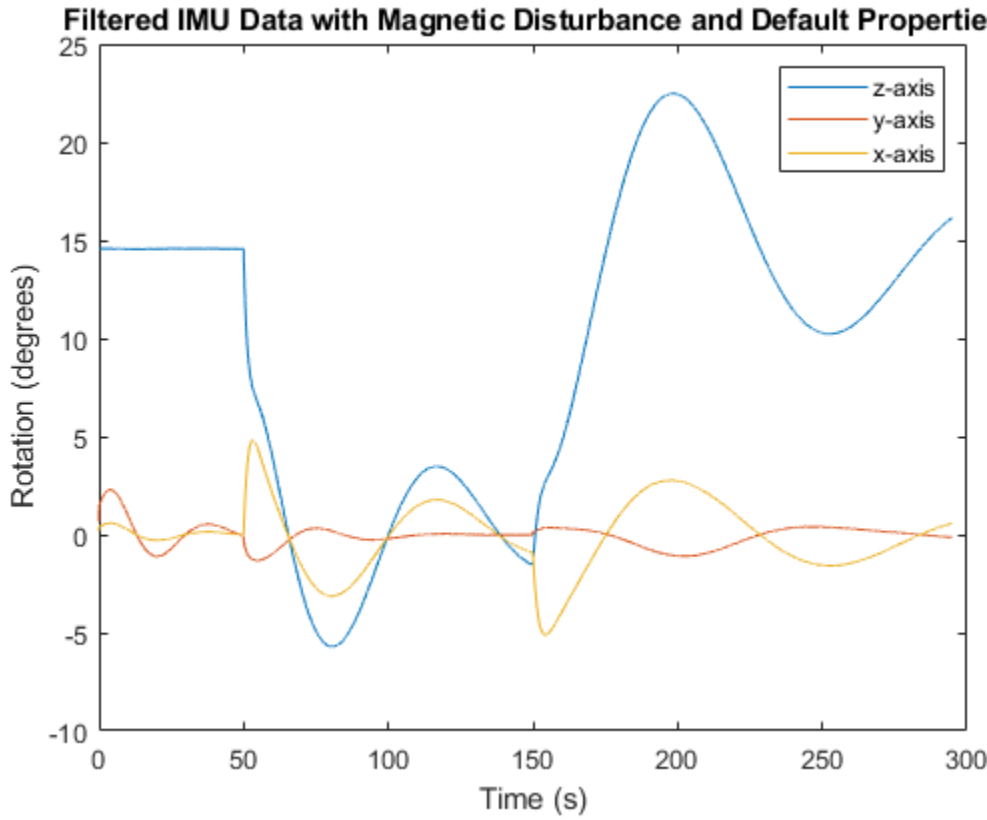
```
title('Simulated Magnetic Field with Jamming')
legend('z-axis','y-axis','x-axis')
```



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')
```

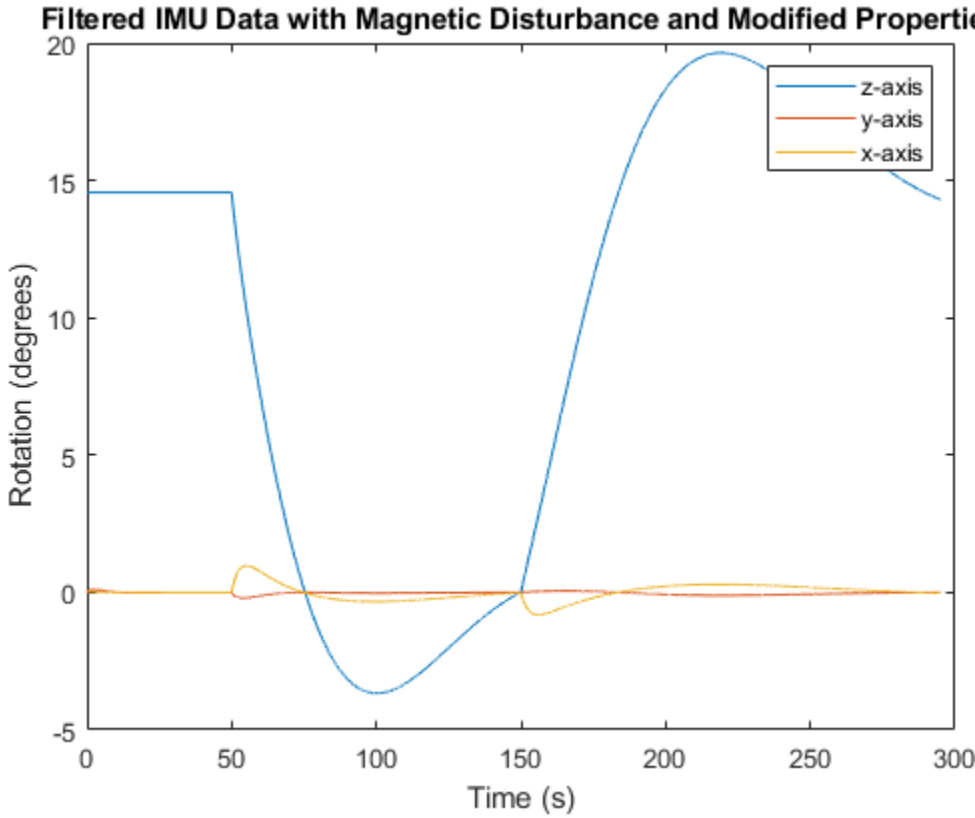


The magnetic jamming was misinterpreted by the AHRS filter, and the sensor body orientation was incorrectly estimated. You can compensate for jamming by increasing the `MagneticDisturbanceNoise` property. Increasing the `MagneticDisturbanceNoise` property increases the assumed noise range for magnetic 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')
```

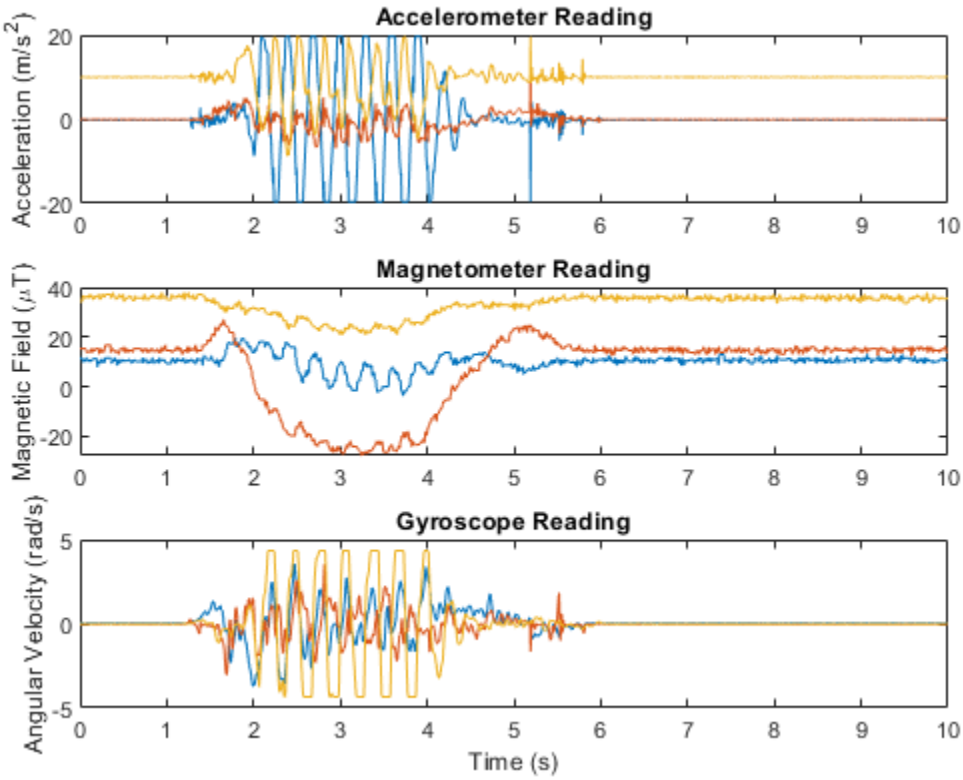


Track Shaking 9-Axis IMU

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.

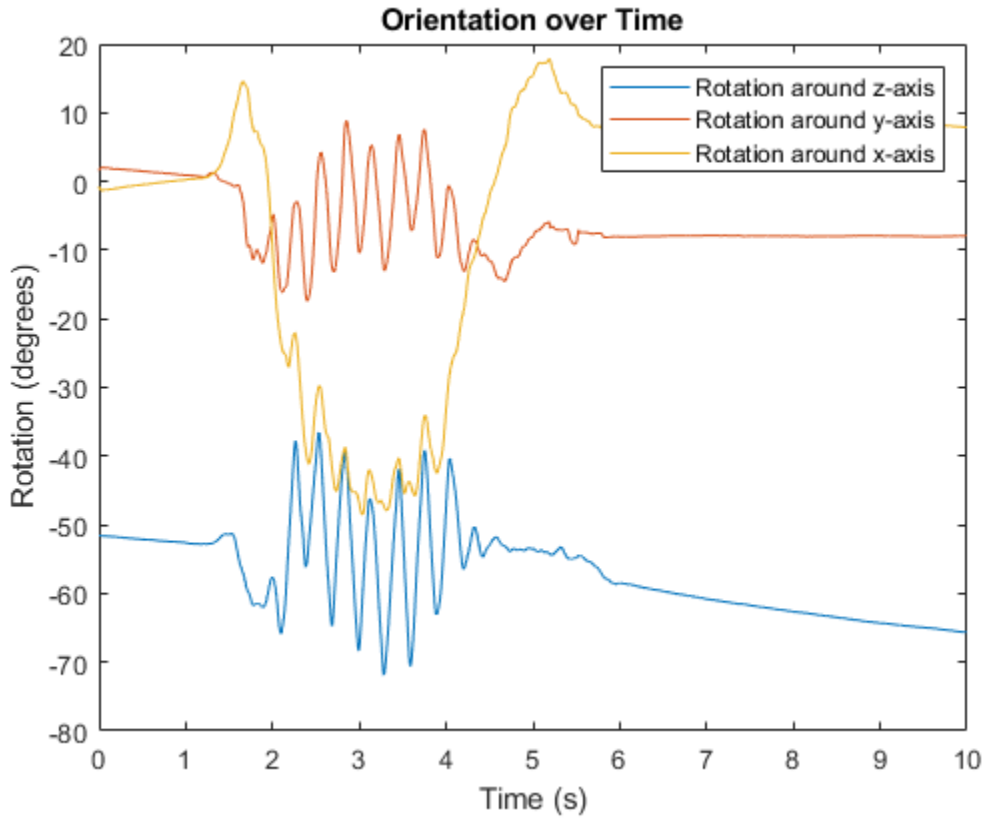
```
load 'IMUReadingsShaken' accelReadings gyroReadings magReadings SampleRate
numSamples = size(accelReadings,1);
time = (0:(numSamples-1))'/SampleRate;

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')
```

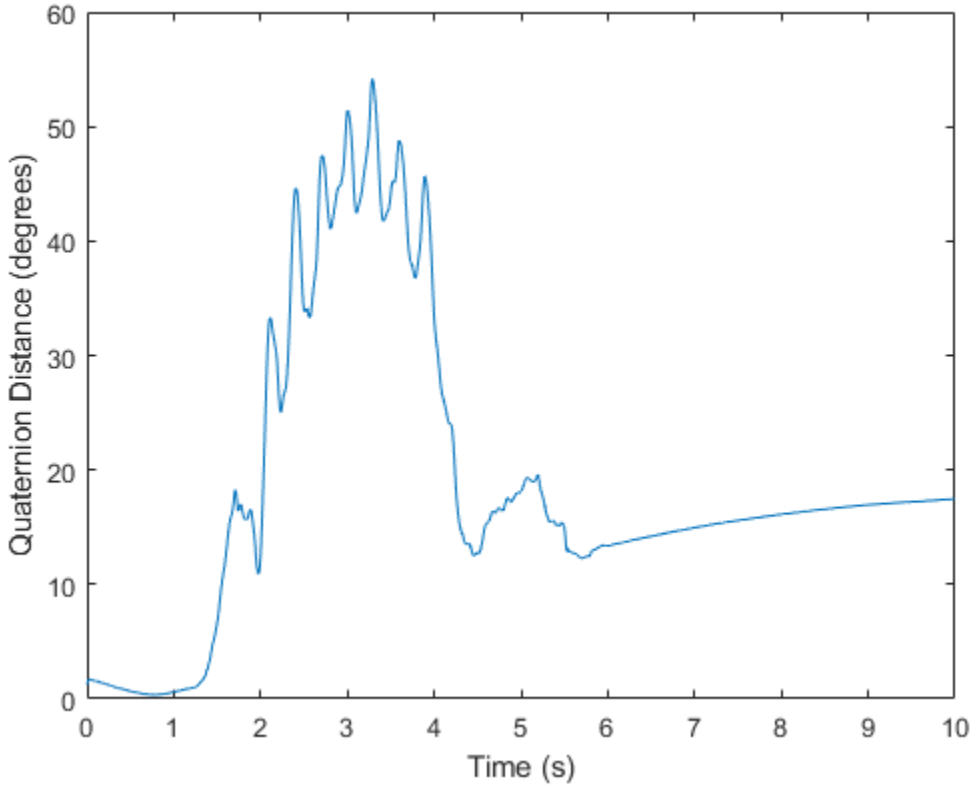


In the IMU recording, the shaking stops after approximately six seconds. Determine the resting orientation so that you can characterize how fast the `ahrsfilter` converges.

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)')
```

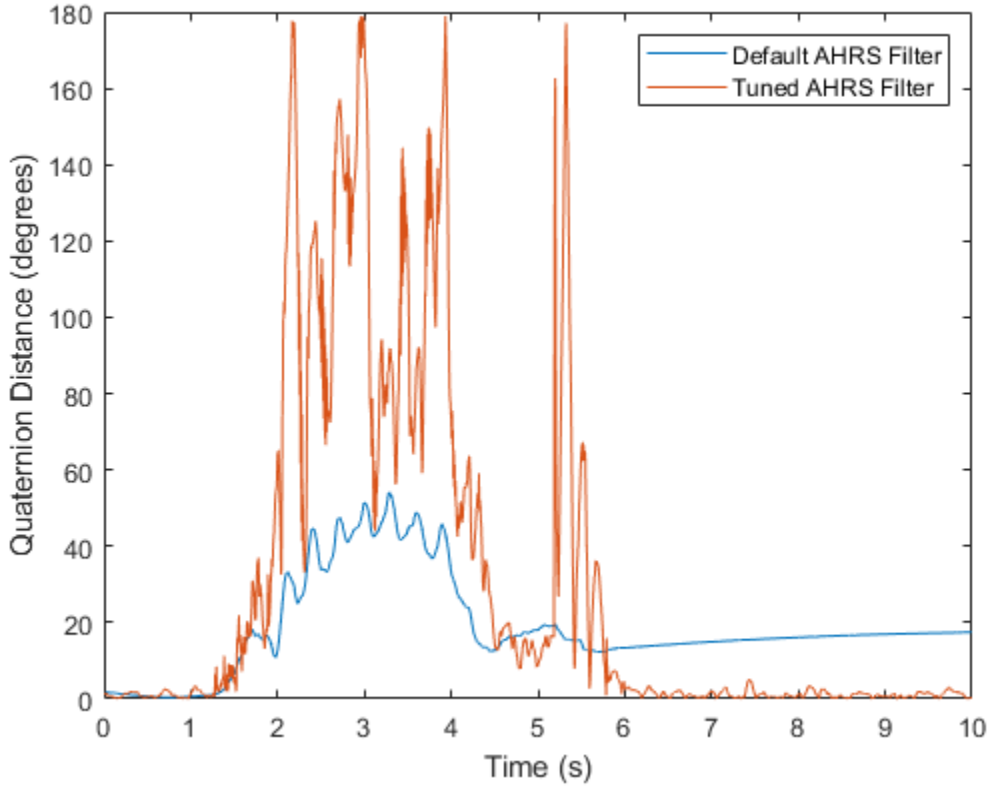


Modify the default `ahrsfilter` properties so that the filter converges to gravity more quickly. Increase the `GyroscopeDriftNoise` to `1e-2` and decrease the `LinearAccelerationNoise` to `1e-4`. This instructs the `ahrsfilter` algorithm to

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

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:

- θ_k –– 3-by-1 orientation error vector, in degrees, at time k
- b_k –– 3-by-1 gyroscope zero rate offset 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 µT, 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:

$$x_k^- = F_k x_{k-1}^+$$
$$P_k^- = F_k P_{k-1}^+ F_k^T + Q_k$$

$$y_k = z_k - H_k x_k^-$$
$$S_k = R_k + H_k P_k^- H_k{}^T$$
$$K_k = P_k^- H_k^T (S_k)^{-1}$$
$$x_k^+ = x_k^- + K_k y_k$$
$$P_k^+ = P_k{}^- - K_k H_k P_k^-$$

Kalman equations used in this algorithm:

$$x_k^- = 0$$

$$P_k^- = Q_k$$

$$y_k = z_k$$

$$S_k = R_k + H_k P_k^- H_k T$$

$$K_k = P_k^- H_k^T (S_k)^{-1}$$

$$x_k^+ = K_k y_k$$

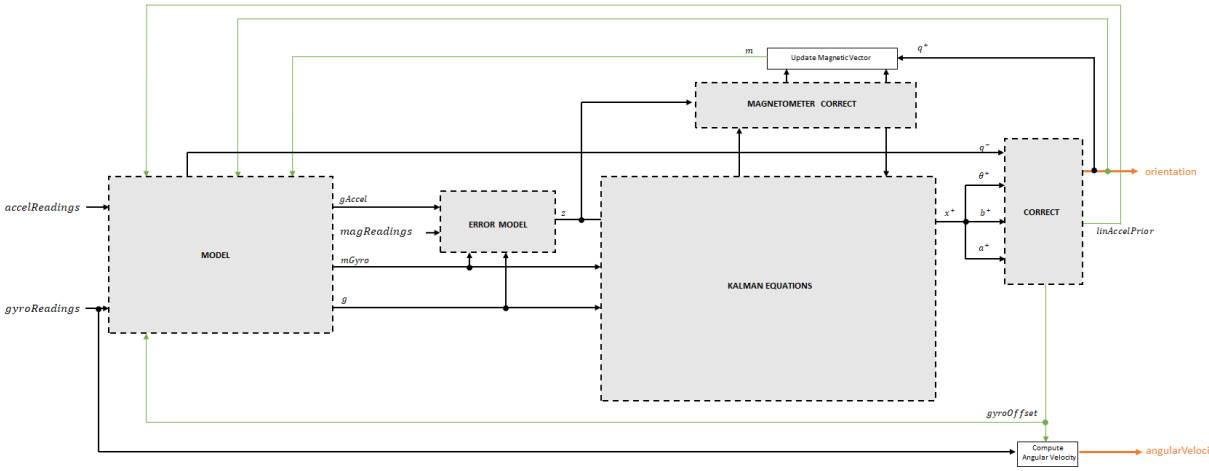
$$P_k^+ = P_k - - K_k H_k P_k^-$$

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.

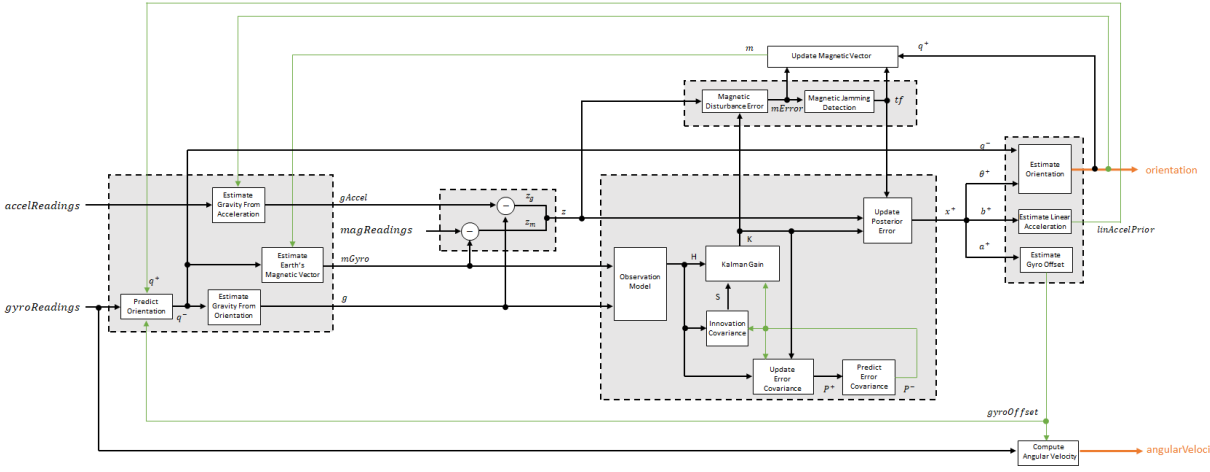
The graphic and following steps describe a single frame-based iteration through the algorithm.



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.

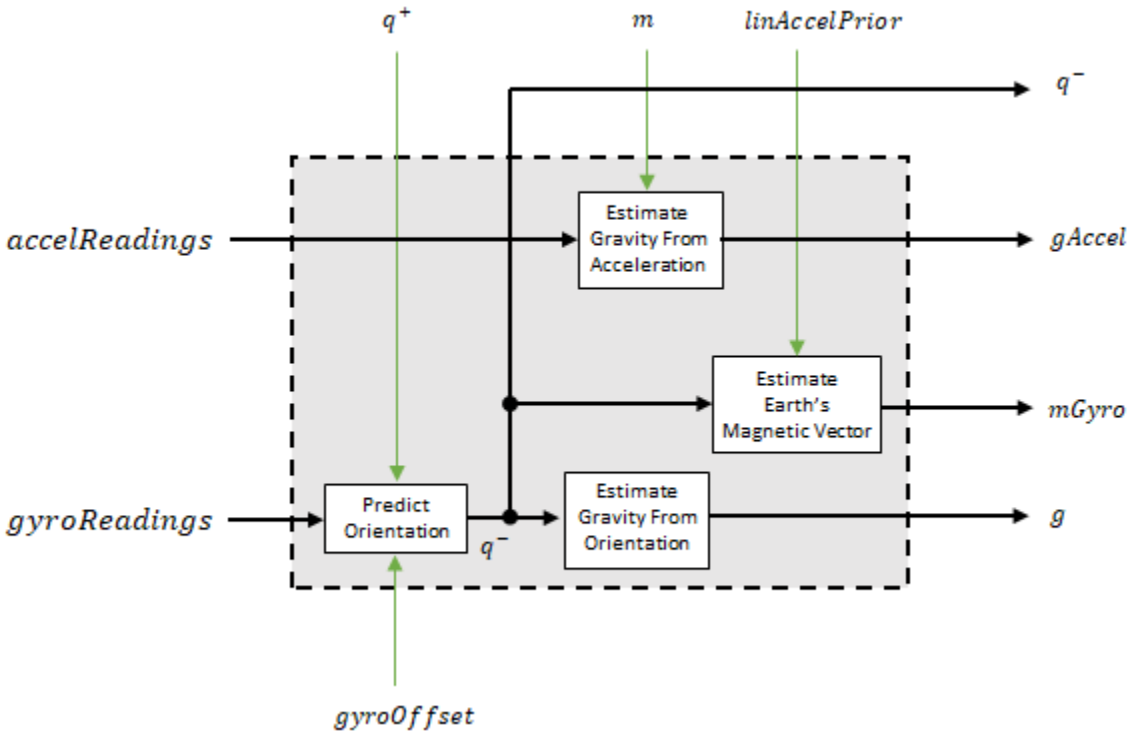
Detailed Overview

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 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 ΔQ:

$$q_{1\times 1}^{-} = \left(q_{1\times 1}^{+}\right)\left(\prod_{n=1}^{N}\Delta Q_n\right)$$

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:

$$g_{1\times 3} = (rPrior(:,3))^{T}$$

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:

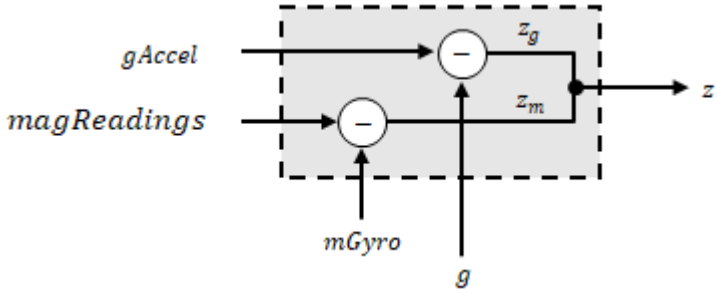
$$gAccel_{1\times 3} = accelReadings_{1\times 3} - linAccelprior_{1\times 3}$$

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:

$$mGyro_{1\times 3} = \left((rPrior)\left(m^{T}\right)\right)^{T}$$

Error Model

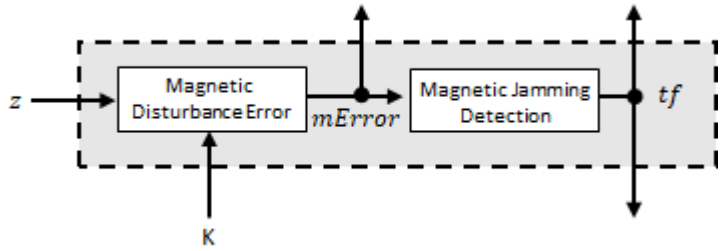


The error model combines two differences:

- 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

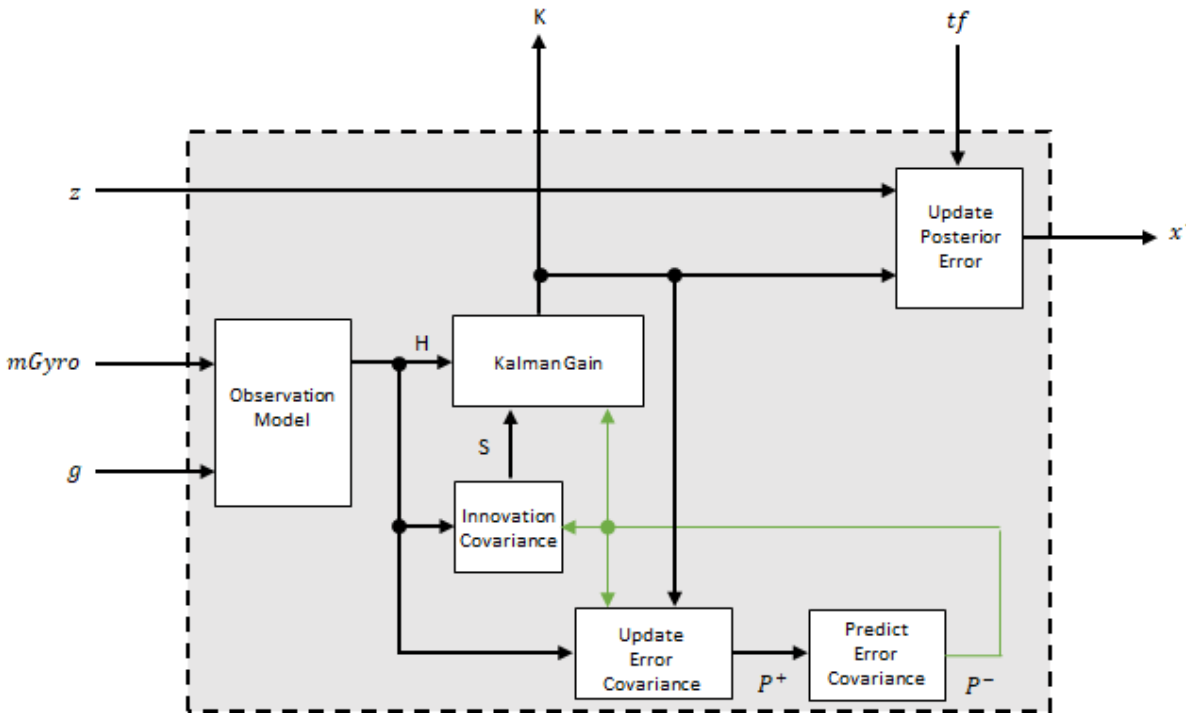
Magnetic jamming is determined by verifying that the power of the detected magnetic disturbance is less than or equal to four times the power of the expected magnetic field strength:

$$tf = \begin{cases} \text{true} & \text{if } \sum |mError|^2 > (4)(\text{ExpectedMagneticFieldStrength})^2 \\ \text{false} & \text{else} \end{cases}$$

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. κ is a constant determined by the SampleRate and DecimationFactor properties: κ = `DecimationFactor`/`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:

$$S_{6x6} = R_{6x6} + (H_{6x12})\left(P^{-}_{12x12}\right)(H_{6x12})^{T}$$

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:

$$R_{6\times 6} = \begin{bmatrix} accel_{\text{noise}} & 0 & 0 & 0 & 0 & 0 \\ 0 & accel_{\text{noise}} & 0 & 0 & 0 & 0 \\ 0 & 0 & accel_{\text{noise}} & 0 & 0 & 0 \\ 0 & 0 & 0 & mag_{\text{noise}} & 0 & 0 \\ 0 & 0 & 0 & 0 & mag_{\text{noise}} & 0 \\ 0 & 0 & 0 & 0 & 0 & mag_{\text{noise}} \end{bmatrix}$$

where

$accel_{\text{noise}}$ = AccelerometerNoise + LinearAccelerationNoise + κ^2 (GyroscopeDriftNoise + GyroscopeNoise)

and

mag_{noise} = MagnetometerNoise + MagneticDisturbanceNoise + κ^2 (GyroscopeDriftNoise + GyroscopeNoise)

The following properties define the observation model noise variance:

- κ –– 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:

$$P^{+}_{12\times 12} = P^{-}_{12\times 12} - (K_{12\times 6})(H_{6\times 12})\left(P^{-}_{12\times 12}\right)$$

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:

$$Q =$$

$$\begin{bmatrix} P^+(1) + \kappa^2 P^+(40) + \beta + \eta & 0 & 0 & -\kappa\left(P^+(40) + \beta\right) \\ 0 & P^+(14) + \kappa^2 P^+(53) + \beta + \eta & 0 & 0 \\ 0 & 0 & P^+(27) + \kappa^2 P^+(66) + \beta + \eta & 0 \\ -\kappa\left(P^+(40) + \beta\right) & 0 & 0 & P^+(40) + \beta \\ 0 & -\kappa\left(P^+(53) + \beta\right) & 0 & 0 \\ 0 & 0 & -\kappa\left(P^+(66) + \beta\right) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

where

- P^+ –– is the updated (*a posteriori*) error estimate covariance
- κ –– DecimationFactor/SampleRate
- β –– GyroscopeDriftNoise
- η –– GyroscopeNoise
- ν –– LinearAcclerationDecayFactor
- ξ –– LinearAccelerationNoise
- σ –– MagneticDisturbanceDecayFactor
- γ –– 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:

$$K_{12\times 6} = \left(P^-_{12\times 12}\right)(H_{6\times 12})^T\left((S_{6\times 6})^T\right)^{-1}$$

where

- P^- –– predicted error covariance
- H –– observation model
- S –– innovation covariance

Update Posterior Error

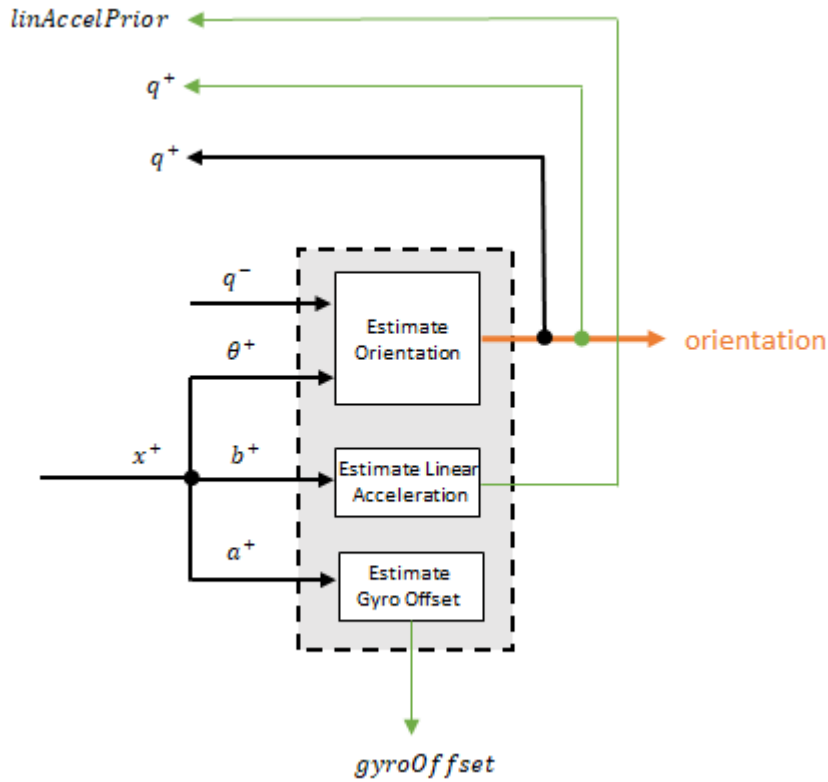
The *a posterior* error estimate is determined by combining the Kalman gain matrix with the error in the gravity vector and magnetic vector estimations:

$$x_{12\times 1} = (K_{12\times 6})(z_{1\times 6})^T$$

If magnetic jamming is detected in the current iteration, the magnetic vector error signal is ignored, and the *a posterior* 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:

$$q^+ = (q^-)\left(\theta^+\right)$$

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

- ν –- LinearAcclerationDecayFactor

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:

$$angularVelocity_{1\times 3} = \frac{\sum gyroReadings_{N\times 3}}{N} - gyroOffset_{1\times 3}$$

where N is the decimation factor specified by the `DecimationFactor` property.

The gyroscope offset estimation is initialized to zeros for the first iteration.

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 NED frame:

$$mErrorNED_{1\times 3} = \left((rPost_{3\times 3})^{T}(mError_{1\times 3})^{T}\right)^{T}$$

The magnetic disturbance error in the NED 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:

$m(1)$ = (ExpectedMagneticFieldStrength)(cos(*inclination*))
$m(2)$ = 0
$m(3)$ = (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

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? (MATLAB).

Creation

Syntax

```
FUSE = imufilter
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(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 (MATLAB).

`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 ($(m/s^2)^2$)

`0.00019247` (default) | positive real scalar

Variance of accelerometer signal noise in $(m/s^2)^2$, 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 ($(rad/s)^2$)

`9.1385e-5` (default) | positive real scalar

Variance of gyroscope signal noise in $(rad/s)^2$, 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 ($(rad/s)^2$)

`3.0462e-13` (default) | positive real scalar

Variance of gyroscope offset drift in $(rad/s)^2$, 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 ($(m/s^2)^2$)

`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` | `uint8` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64`

`LinearAcclerationDecayFactor` — 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   0.000006092348396                   0                   0                   0                   0
                   0                   0   0.000006092348396                   0                   0                   0
                   0                   0                   0   0.000076154354947                   0                   0
                   0                   0                   0                   0   0.000076154354947                   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

  Columns 7 through 9

                   0                   0                   0
                   0                   0                   0
                   0                   0                   0
                   0                   0                   0
                   0                   0                   0
                   0                   0                   0
   0.009623610000000                   0                   0
                   0   0.009623610000000                   0
                   0                   0   0.009623610000000
```

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^2)

N-by-3 matrix

Accelerometer readings in the sensor body coordinate system in m/s^2, 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 OrienationFormat 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)
```

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.

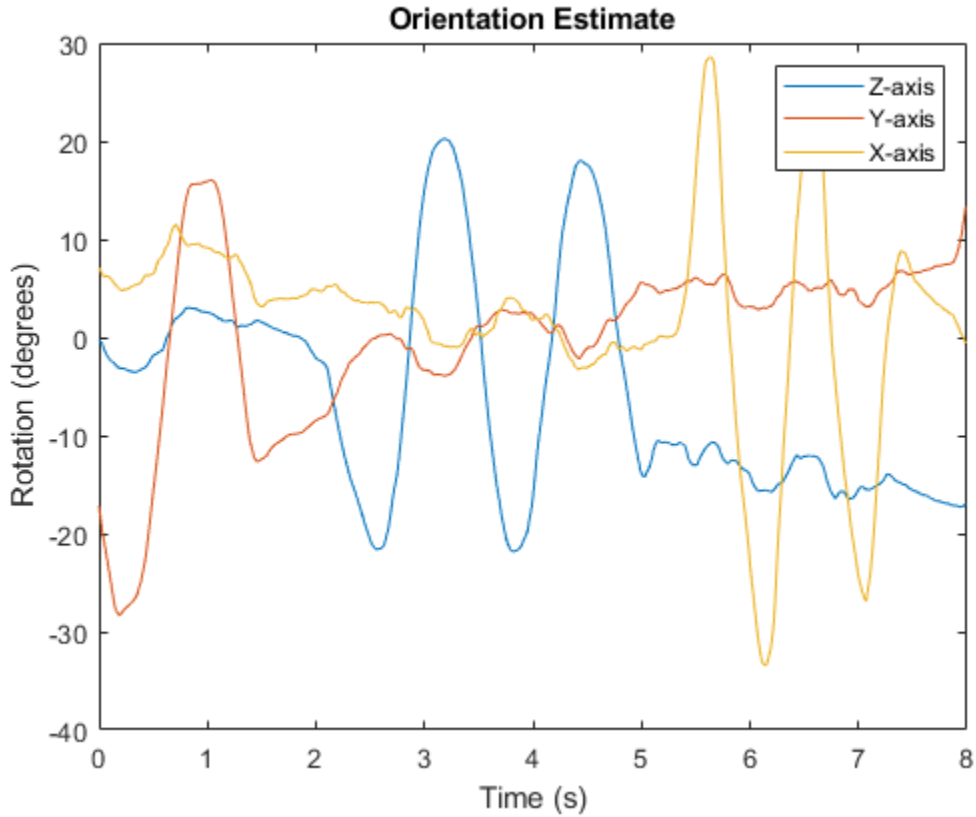
```
q = fuse(accelerometerReadings,gyroscopeReadings);
```

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:

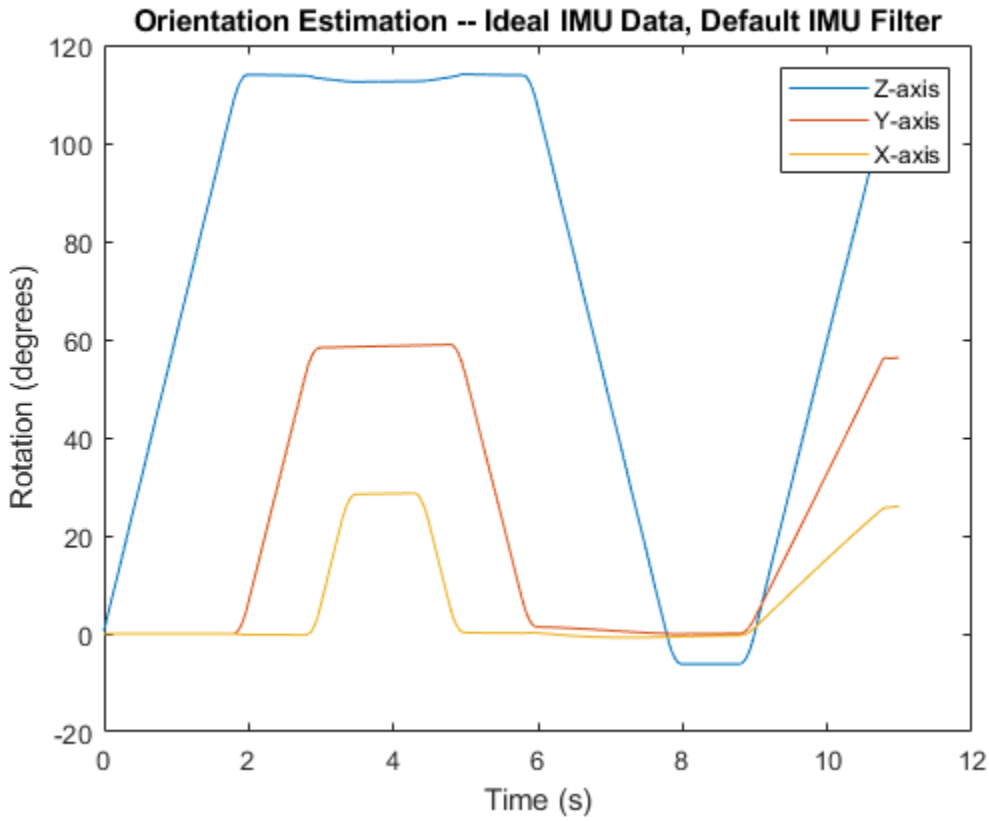
1. Simulate IMU output by feeding the ground-truth motion to the IMU sensor object.
2. Filter the IMU output using the default IMU filter object.

```
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.

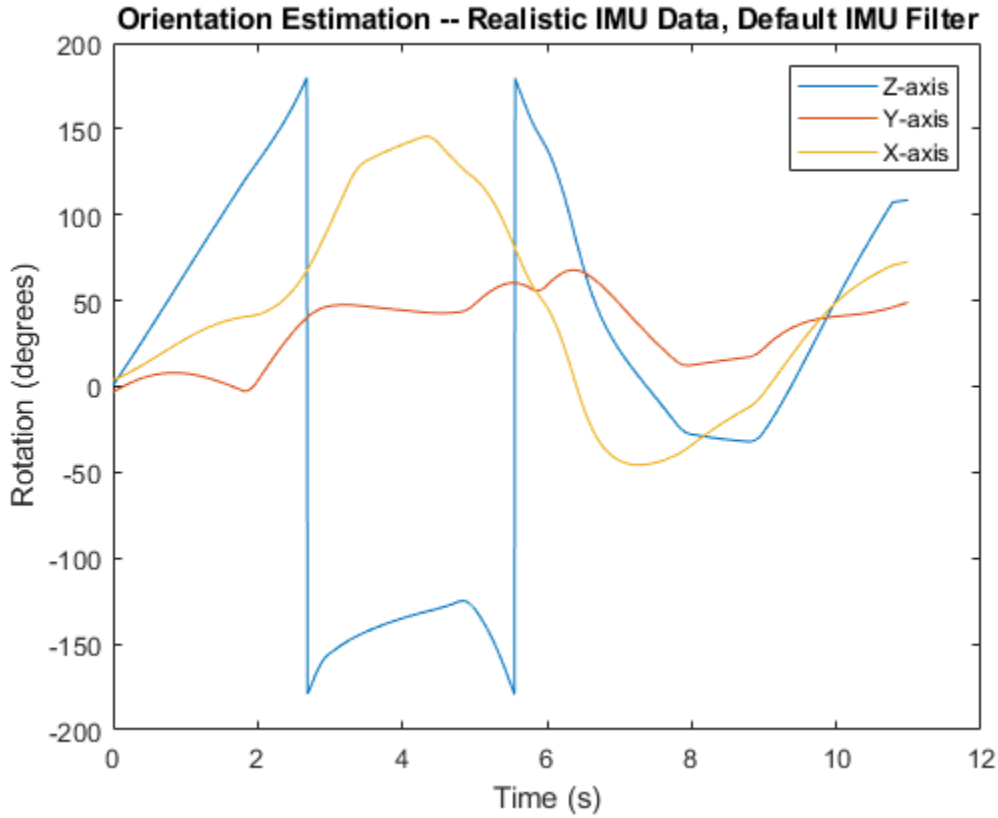
```
figure(1)
plot(t,eulerd(orientation,'ZYX','frame'))
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Ideal IMU Data, Default IMU Filter')
legend('Z-axis','Y-axis','X-axis')
```



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, ...
```

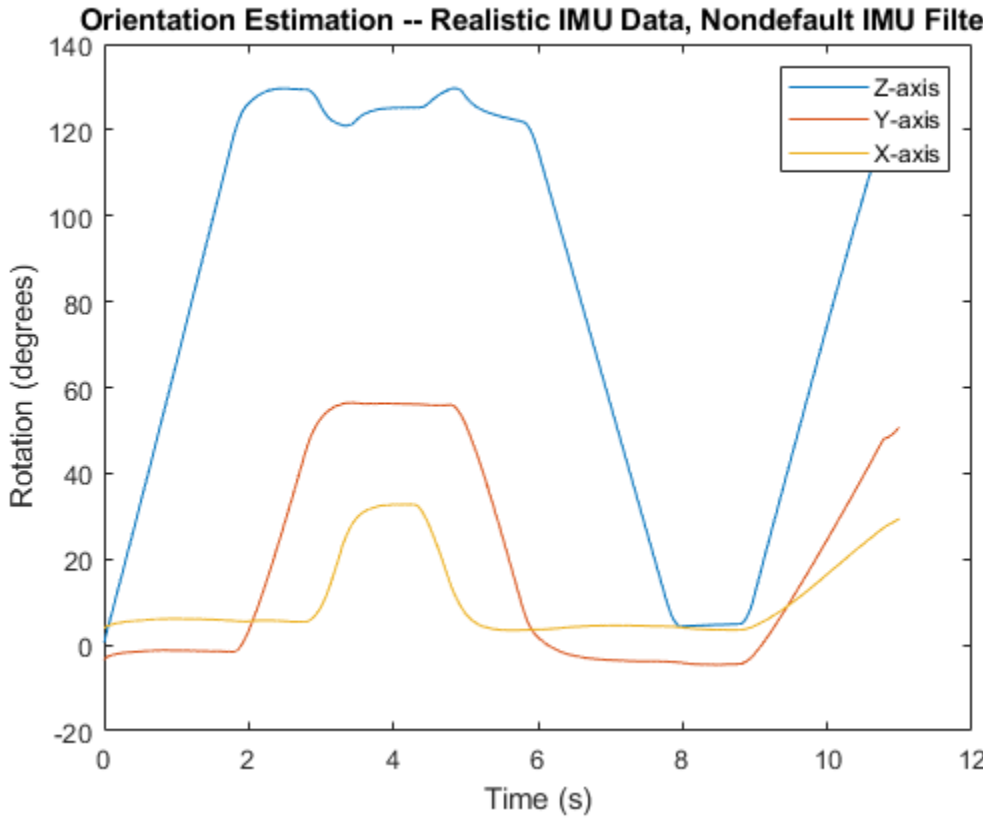
```
    'Resolution',0.00013323, ...
    'AxesMisalignment',2, ...
    'NoiseDensity',8.7266e-05, ...
    'TemperatureBias',0.34907, ...
    'TemperatureScaleFactor',0.02, ...
    'AccelerationBias',0.00017809, ...
    'ConstantBias',[0.3491,0.5,0]);

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.

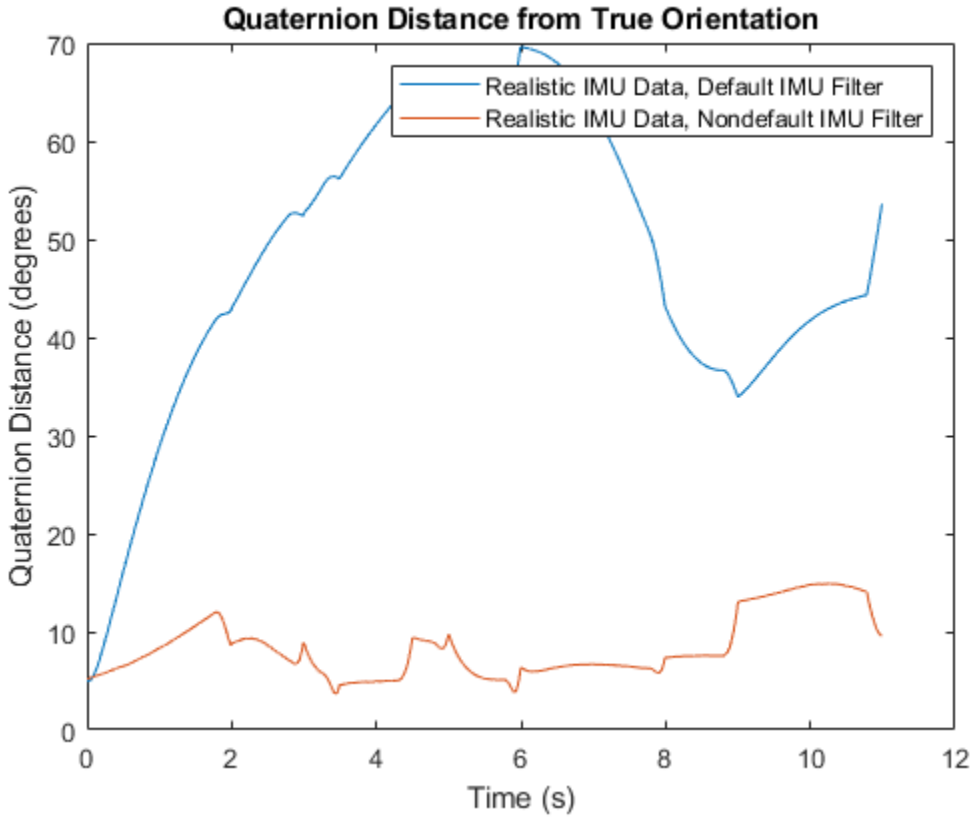
```
aFilter.GyroscopeNoise          = 7.6154e-7;
aFilter.AccelerometerNoise      = 0.0015398;
aFilter.GyroscopeDriftNoise     = 3.0462e-12;
aFilter.LinearAccelerationNoise = 0.00096236;
aFilter.InitialProcessNoise     = aFilter.InitialProcessNoise*10;

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 a constant angular velocity about the *y*- and *z*-axes.

```
duration = 60*5;
fs = 20;
numSamples = duration * fs;

angVelBody = repmat([0,0.5,0.25],numSamples,1);
```

```
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('BiasInstability',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.

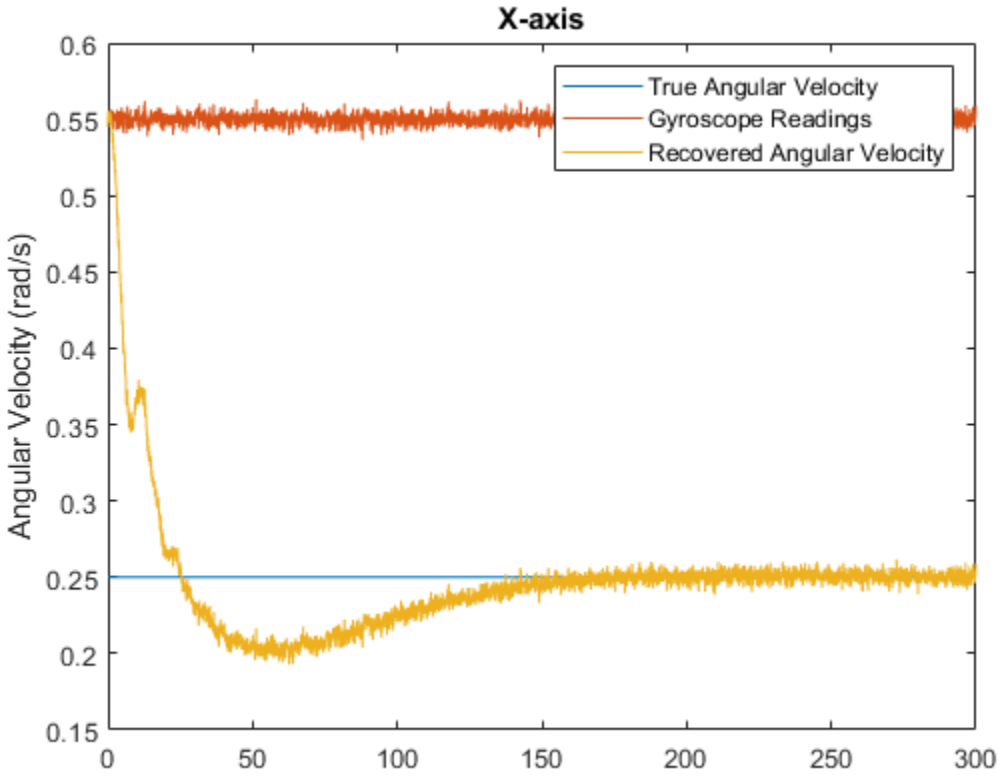
```
fuse = imufilter('SampleRate',fs);

[~,angVelBodyRecovered] = fuse(accelReadings,gyroReadingsBody);
```

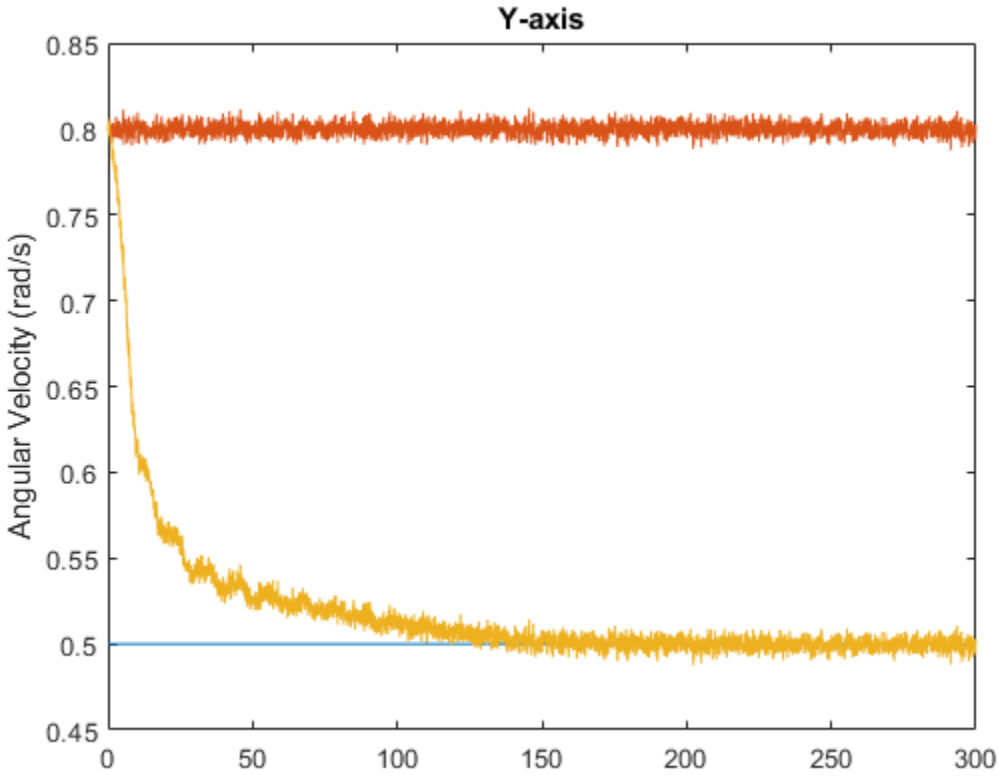
Plot the ground-truth angular velocity, the gyroscope readings, and the recovered angular velocity for each axis.

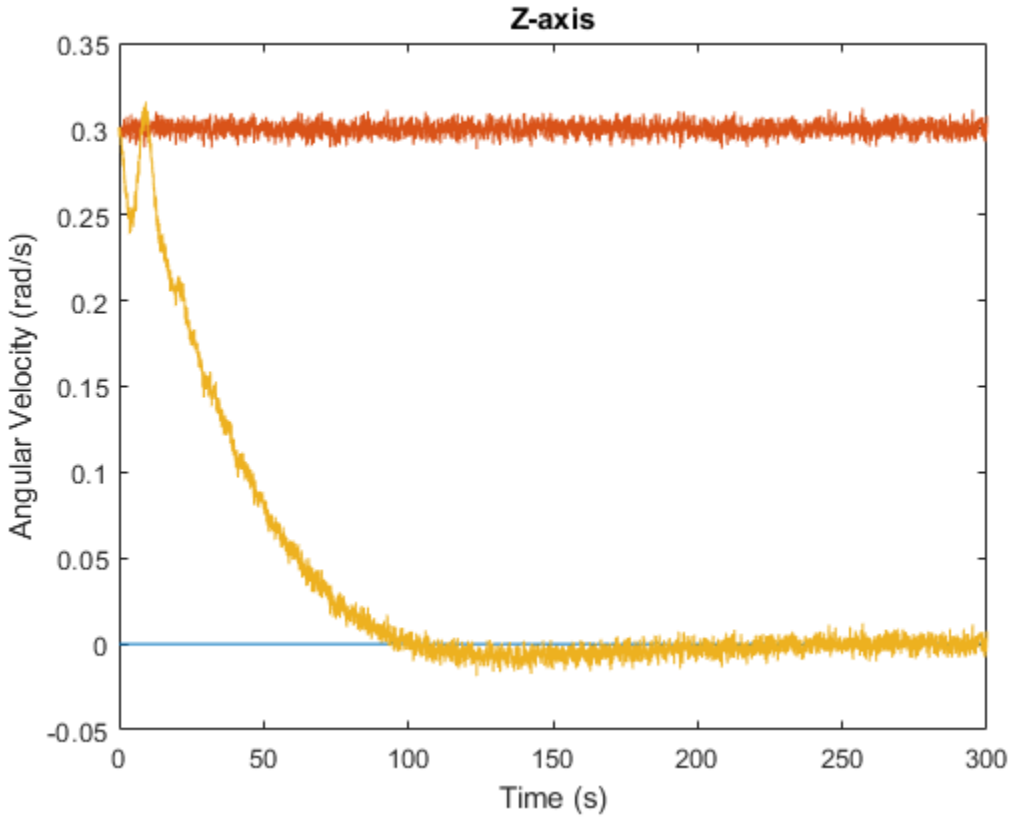
The angular velocity returned from the `imufilter` compensates for the effect of the gyroscope bias over time and converges to the true angular velocity.

```
time = (0:numSamples-1)'/fs;

figure(1)
plot(time,angVelBody(:,3), ...
     time,gyroReadingsBody(:,3), ...
     time,angVelBodyRecovered(:,3))
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(:,1), ...
     time,gyroReadingsBody(:,1), ...
     time,angVelBodyRecovered(:,1))
title('Z-axis')
ylabel('Angular Velocity (rad/s)')
xlabel('Time (s)')
```




Y-axis
Angular Velocity (rad/s)
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0
50
100
150
200
250
300



Algorithms

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:

- θ_k –- 3-by-1 orientation error vector, in degrees, at time k
- b_k –- 3-by-1 gyroscope zero rate offset 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:

$$x_k^- = F_k x_{k-1}^+$$
$$P_k^- = F_k P_{k-1}^+ F_k^T + Q_k$$

$$y_k = z_k - H_k x_k^-$$
$$S_k = R_k + H_k P_k^- H_k^T$$
$$K_k = P_k^- H_k^T (S_k)^{-1}$$
$$x_k^+ = x_k^- + K_k y_k$$
$$P_k^+ = P_k^- - K_k H_k P_k^-$$

Kalman equations used in this algorithm:

$$x_k^- = 0$$
$$P_k^- = Q_k$$

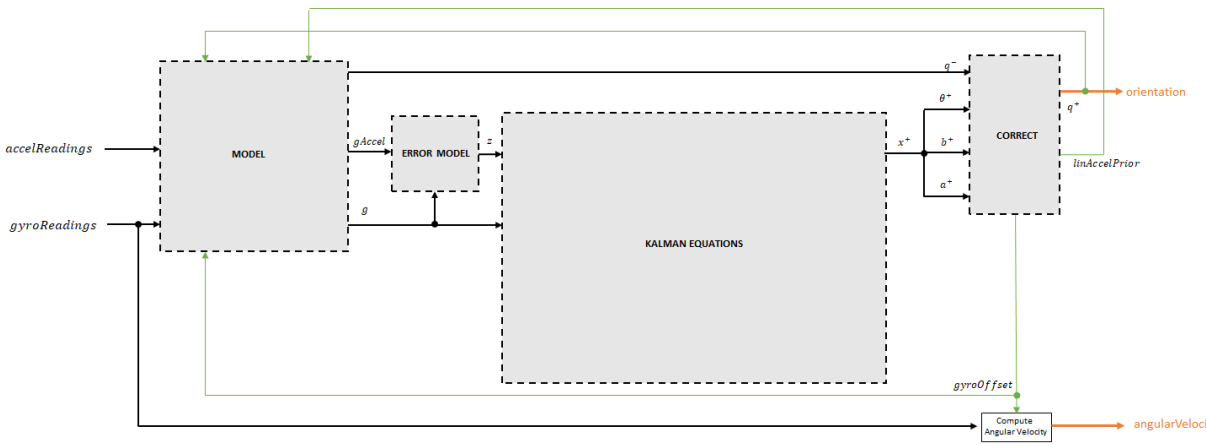
$$y_k = z_k$$
$$S_k = R_k + H_k P_k^- H_k^T$$
$$K_k = P_k^- H_k^T (S_k)^{-1}$$
$$x_k^+ = K_k y_k$$
$$P_k^+ = P_k^- - K_k H_k P_k^-$$

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.

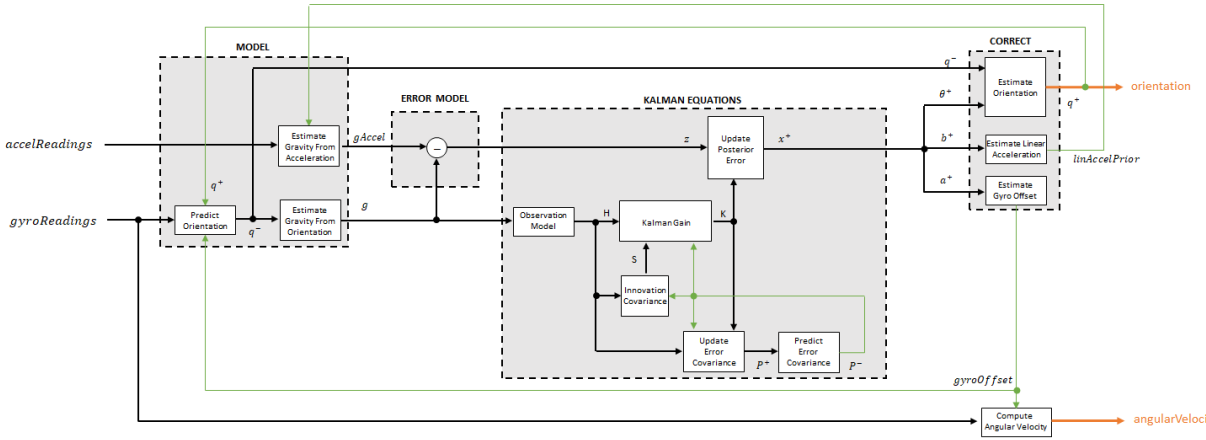
The graphic and following steps describe a single frame-based iteration through the algorithm.



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.

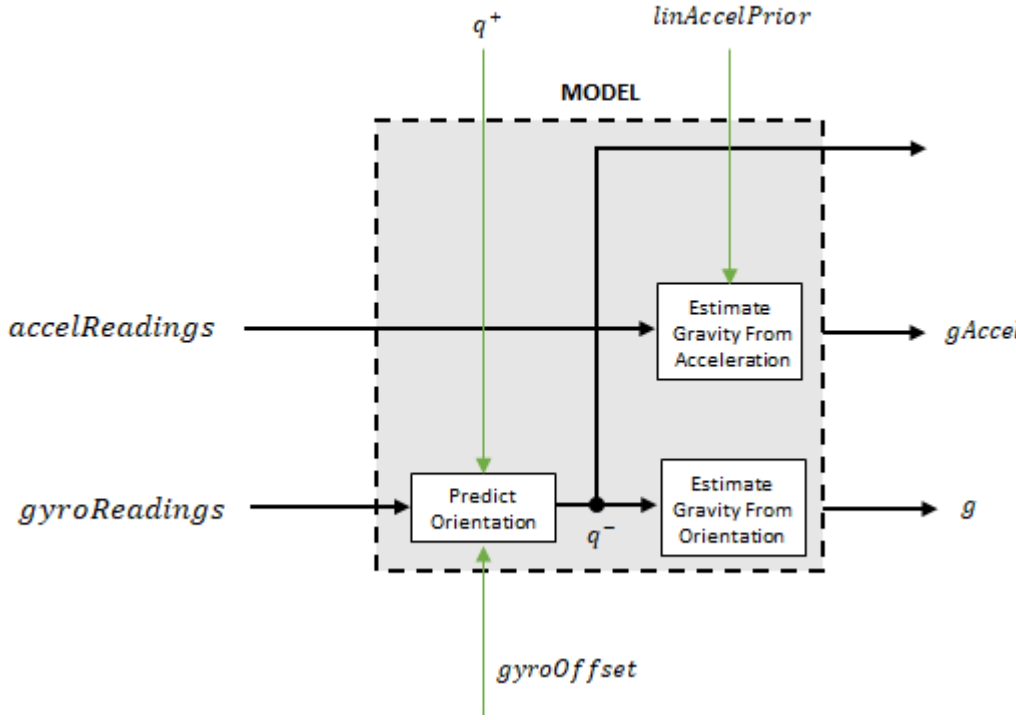
Detailed Overview

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 ΔQ:

$$q_{1\times 1}^{-} = \left(q_{1\times 1}^{+}\right)\left(\prod_{n=1}^{N} \Delta Q_n\right)$$

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.

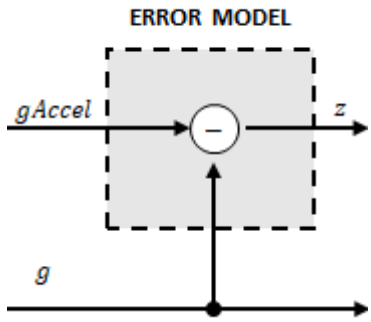
Estimate Gravity from Acceleration

A second gravity vector estimation is made by subtracting the decayed linear acceleration estimate of the previous iteration from the accelerometer readings:

$$gAccel_{1\times 3} = accelReadings_{1\times 3} - linAccelprior_{1\times 3}$$

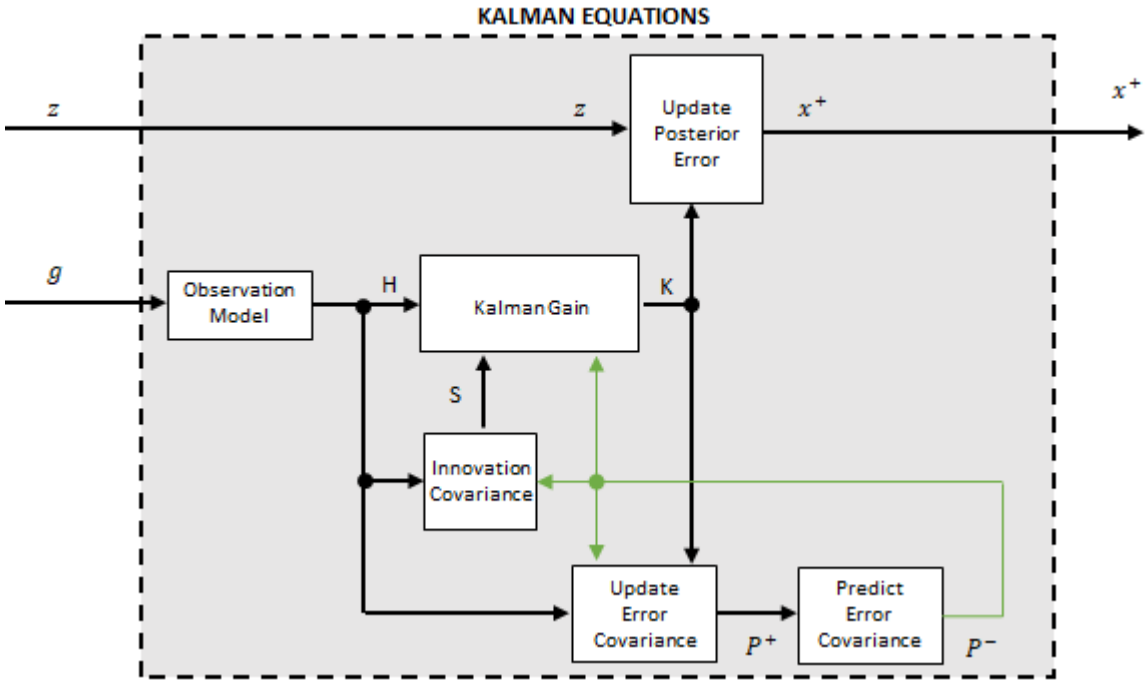
Error Model

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. κ is a constant determined by the SampleRate and DecimationFactor properties: κ = `DecimationFactor`/`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:

$$S_{3x3} = R_{3x3} + (H_{3x9})\left(P^{-}_{9x9}\right)(H_{3x9})^{T}$$

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:

  - κ –- (DecimationFactor/SampleRate)^2
  - β –- GyroscopeDriftNoise
  - η –- GyroscopeNoise
  - λ –- AccelerometerNoise
  - ξ –- 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:

$$P^{+}_{9\times 9} = P^{-}_{9\times 9} - (K_{9\times 3})(H_{3\times 9})\left(P^{-}_{9\times 9}\right)$$

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:

$Q =$

$$\begin{bmatrix} P^+(1) + \kappa^2 P^+(31) + \beta + \eta & 0 & 0 & -\kappa\left(P^+(31) + \beta\right) \\ 0 & P^+(11) + \kappa^2 P^+(41) + \beta + \eta & 0 & 0 \\ 0 & 0 & P^+(21) + \kappa^2 P^+(51) + \beta + \eta & 0 \\ -\kappa\left(P^+(31) + \beta\right) & 0 & 0 & P^+(31) + \beta \\ 0 & P^+(41) + \beta & 0 & 0 \\ 0 & 0 & P^+(51) + \beta & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

where

- P^+ –– is the updated (*a posteriori*) error estimate covariance
- κ –– DecimationFactor/SampleRate
- β –– GyroscopeDriftNoise
- η –– GyroscopeNoise
- ν –– LinearAcclerationDecayFactor
- ξ –– 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:

$$K_{9\times 3} = \left(P^-_{9\times 9}\right)\left(H_{3\times 9}\right)^T\left(\left(S_{3\times 3}\right)^T\right)^{-1}$$

where

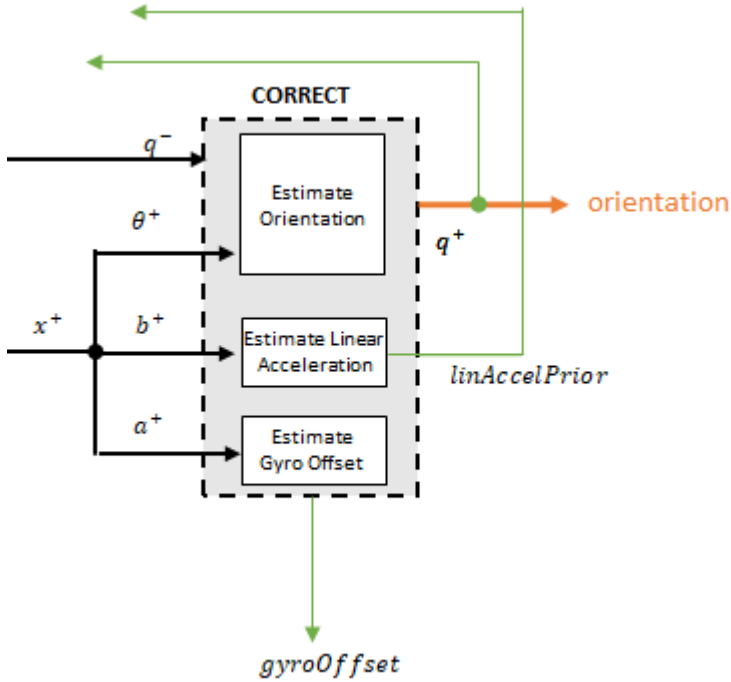
- P^- –– predicted error covariance
- H –– observation model
- S –– innovation covariance

Update Posterior Error

The posterior error estimate is determined by combining the Kalman gain matrix with the error in the gravity vector estimations:

$$x_{9\times 1} = \left(K_{9\times 3}\right)\left(z_{1\times 3}\right)^T$$

Correct



Estimate Orientation

The orientation estimate is updated by multiplying the previous estimation by the error:

$$q^+ = (q^-)\left(\theta^+\right)$$

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

- ν –– LinearAcclerationDecayFactor

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:

$$angularVelocity_{1\times 3} = \frac{\sum gyroReadings_{N\times 3}}{N} - gyroOffset_{1\times 3}$$

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

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? (MATLAB).

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 (MATLAB).

`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

`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)
```

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 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');
```

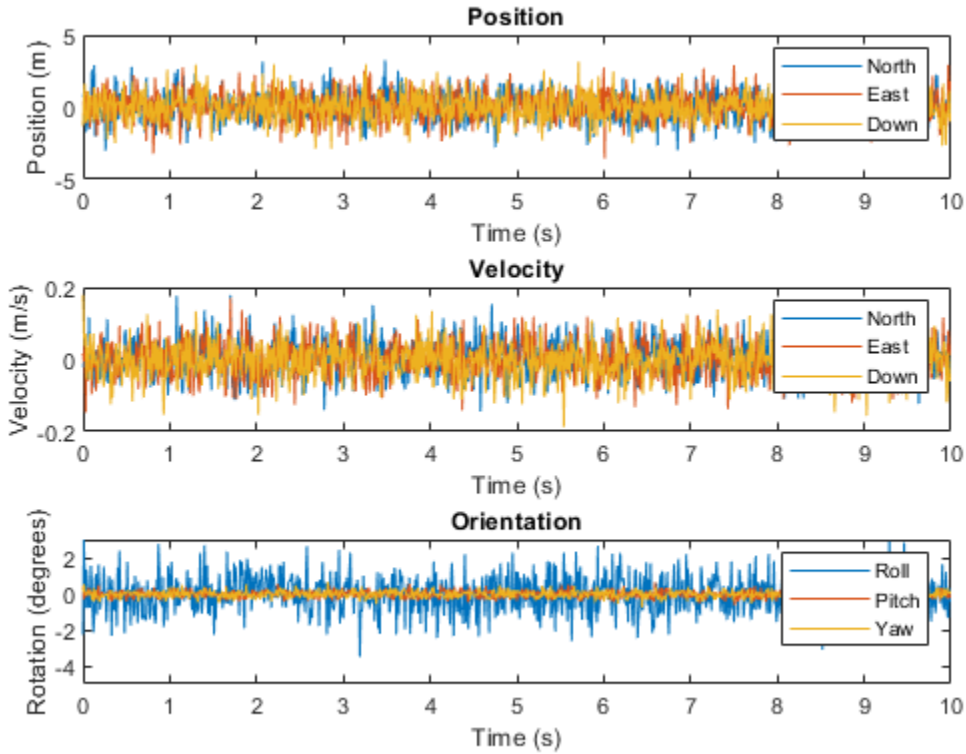
In a loop, call `INS` with the stationary motion struct to return the position, velocity, and orientation measurements in the local NED coordinate system. Log the position, velocity, and orientation measurements.

```
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`.

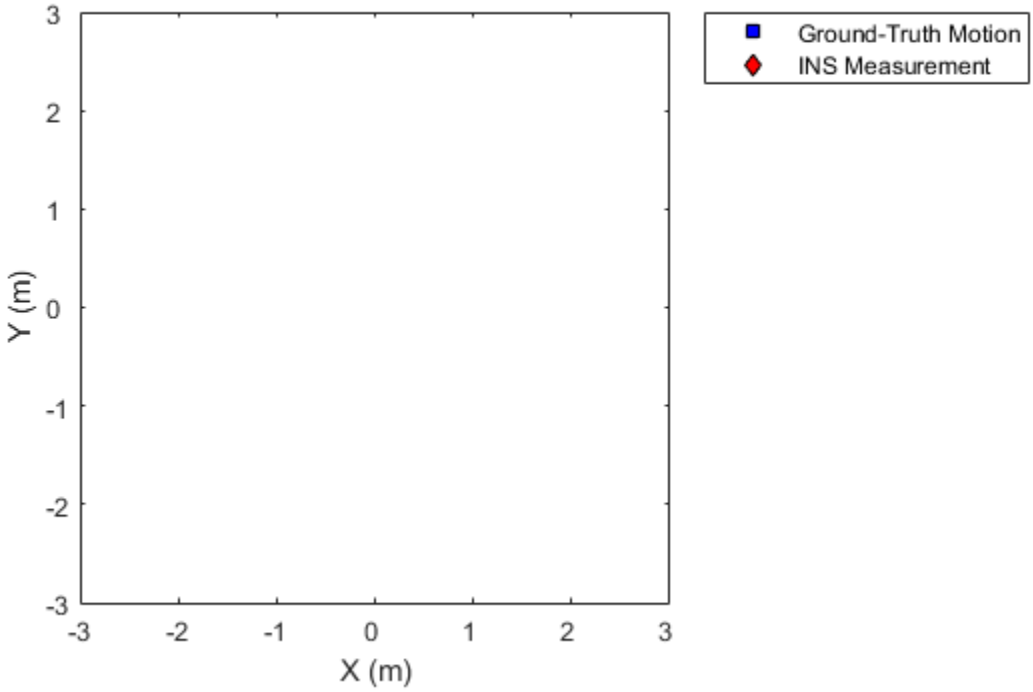
```
ins = insSensor('PositionAccuracy',0.1);
```

Create a tracking scenario with a single platform whose motion is defined by `path`.

```
scenario = trackingScenario('UpdateRate',Fs);
quadcopter = platform(scenario);
quadcopter.Trajectory = path;
```

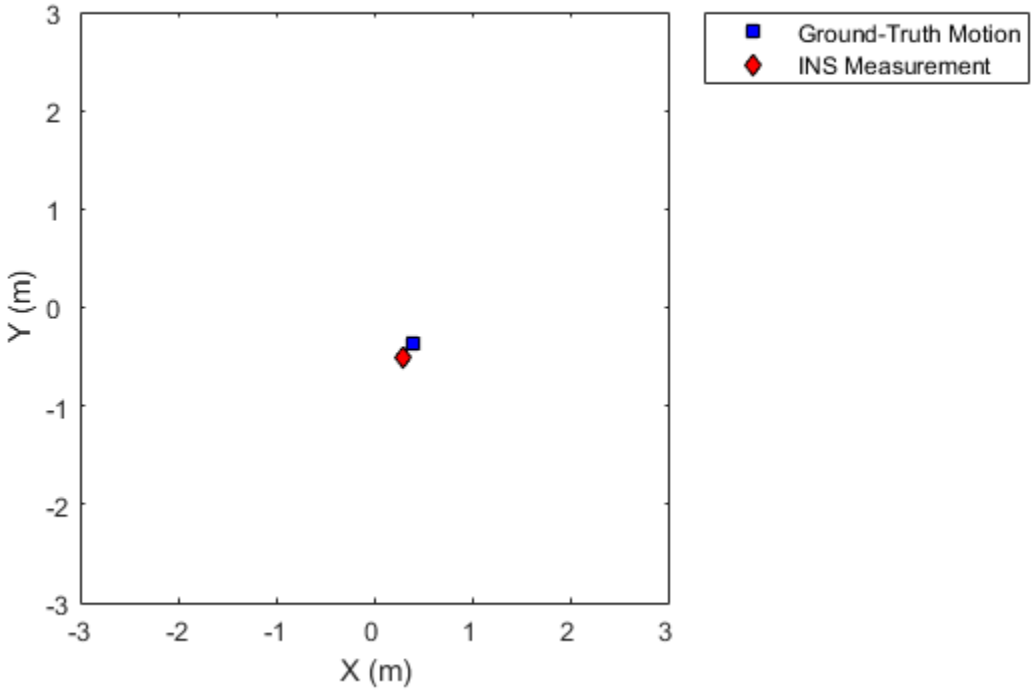
Create a theater plot to visualize the ground-truth `quadcopter` motion and the quadcopter motion measurements modeled by `insSensor`.

```
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);
```

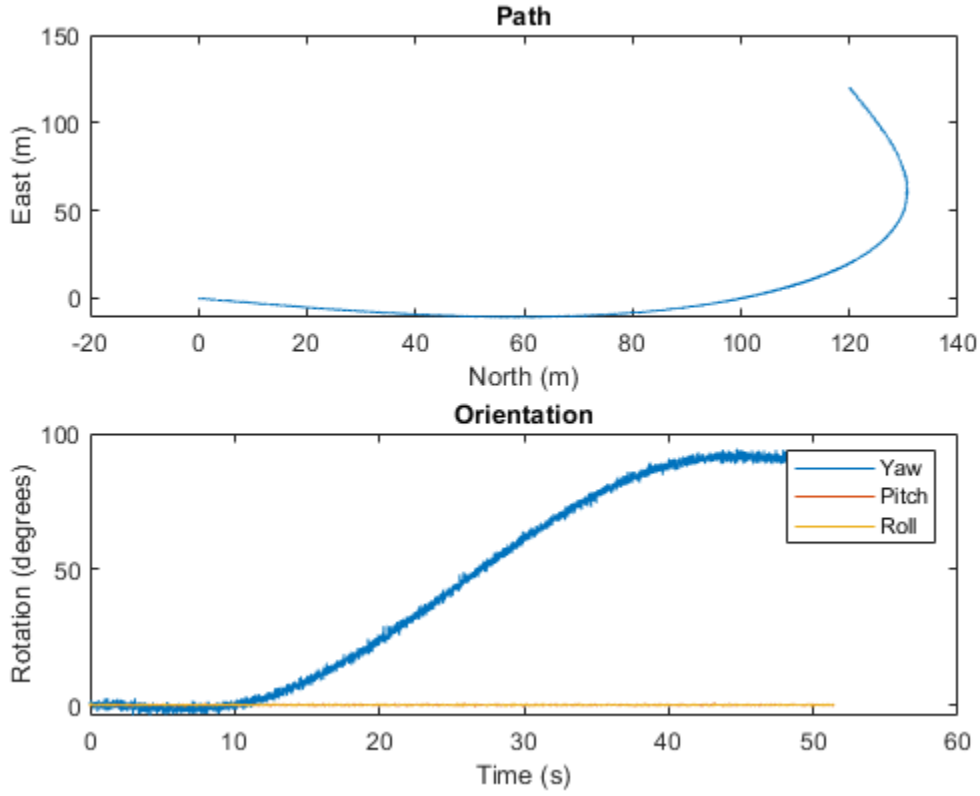
Call the waypoint trajectory object, `path`, to generate the ground-truth motion. Call the INS simulator, `ins`, with the ground-truth motion to generate INS measurements.

```
[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

System 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? (MATLAB).

Creation

Syntax

```
GPS = gpsSensor
GPS = gpsSensor(Name,Value)
```

Description

`GPS = gpsSensor` returns a System object, `GPS`, 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° N
- longitude: 0° E
- altitude: 0 m

`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 (MATLAB).

`UpdateRate` — 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 NED 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 NED coordinate system (m)
N-by-3 matrix

Position of the GPS receiver in the local NED 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 NED coordinate system (m/s)
N-by-3 matrix

Velocity of GPS receiver in the local NED 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 NED coordinate system (m/s)
N-by-3 matrix

Velocity of the GPS receiver in the local NED 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 NED coordinate system (m/s)
N-by-1 column vector

Magnitude of the horizontal velocity of the GPS receiver in the local NED 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 NED coordinate system (°)
N-by-1 column vector

Direction of the horizontal velocity of the GPS receiver in the local NED 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];
```

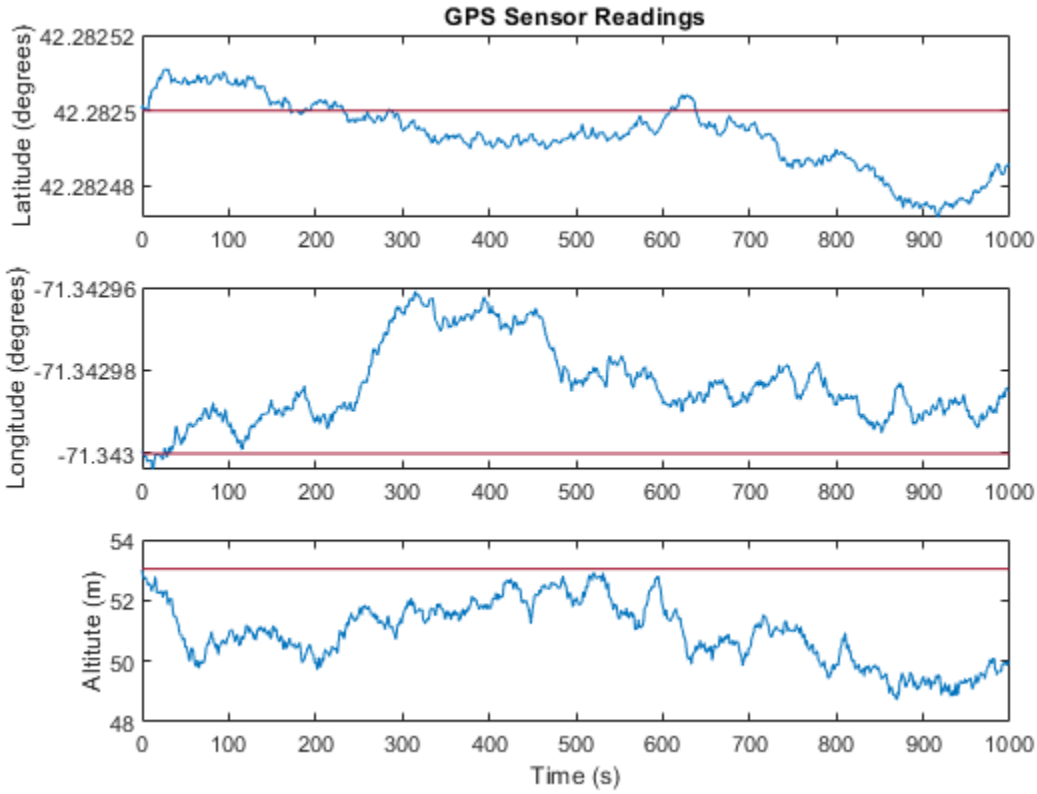
```
truePosition = zeros(numSamples,3);
trueVelocity = zeros(numSamples,3);

gps = gpsSensor('UpdateRate',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('Altitute (m)')
xlabel('Time (s)')
```

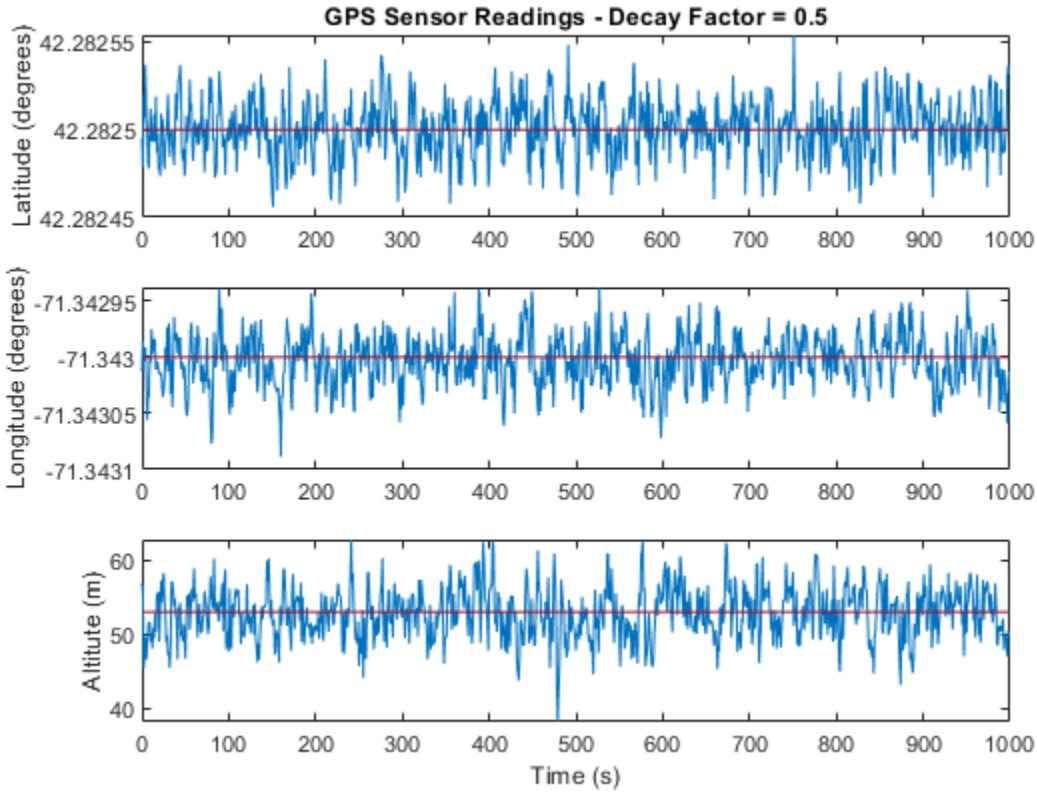


The position readings have noise controlled by `HorizontalPositionAccuracy`, `VerticalPositionAccuracy`, `VelocityAccuracy`, and `DecayFactor`. The `DecayFactor` property controls the drift in the noise model. By default, `DecayFactor` is set to `0.999`, which approaches a random walk process. To observe 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('Altitute (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:

                  UpdateRate: 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.UpdateRate;

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];
```

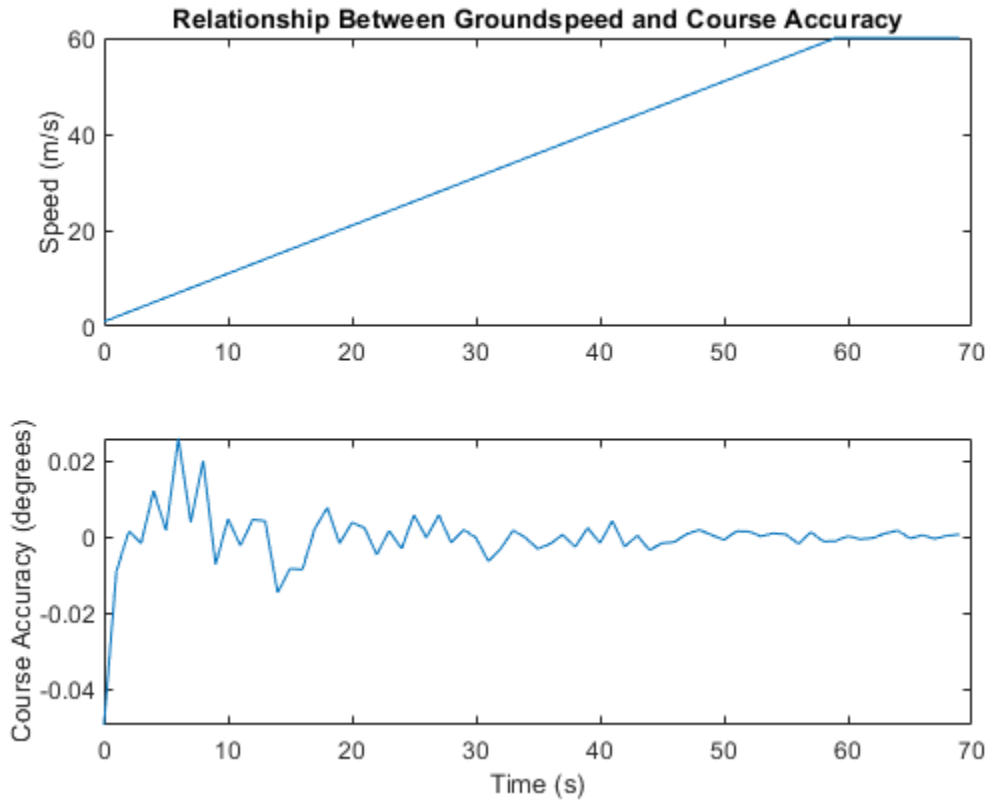
Model GPS measurement data by calling the GPS object with your velocity and position matrices.

```
[~,~,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.UpdateRate;

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);
```

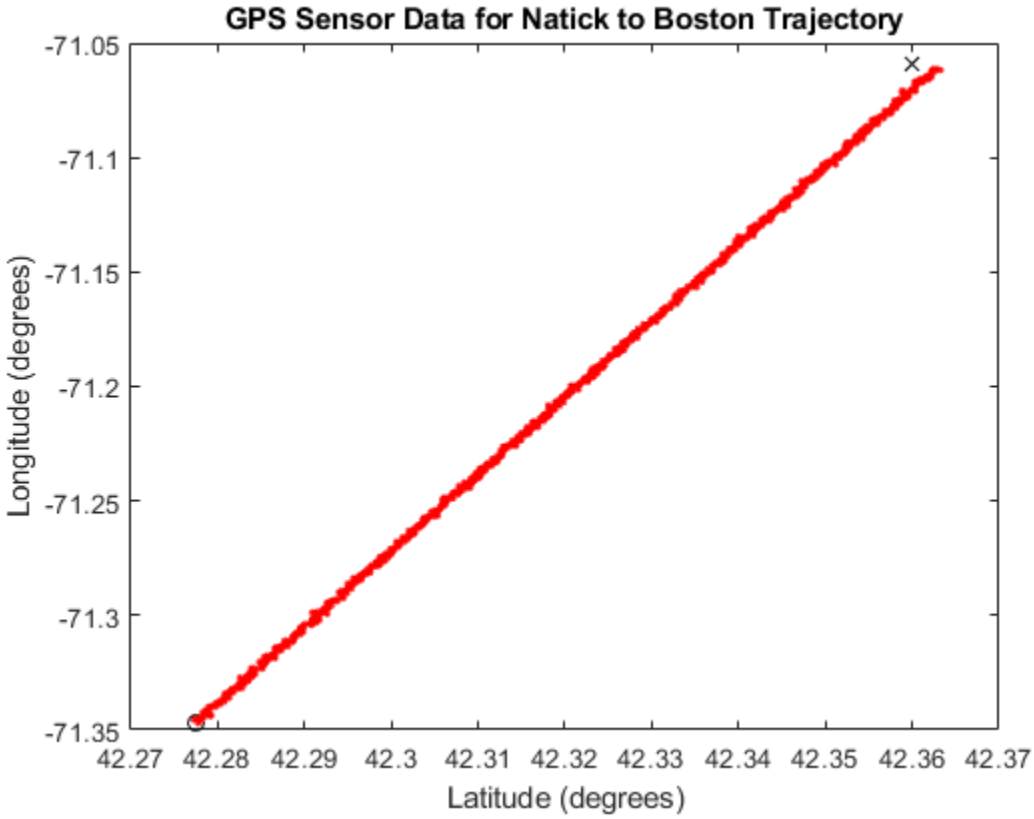
Create a `gpsSensor` object to model receiving GPS data for the platform. Set the `HorizontalPositionalAccuracy` to `25` and the `DecayFactor` to `0.25` to emphasize the noise. Set the `ReferenceLocation` to the Natick coordinates in LLA.

```
GPS = gpsSensor( ...
    'HorizontalPositionAccuracy',25, ...
    'DecayFactor',0.25, ...
    'UpdateRate',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

System 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.

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? (MATLAB).

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 3-140 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 3-140 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 3-140 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 (MATLAB).

`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 3-135.

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 dBmi.

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

`[1;5]` | 2-by-1 vector of positive real values

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 radar are not detected. Units are in degrees.

Example: `[14;7]`

Dependencies

To enable this property, set the `DetectionMode` property to `'ESM'` or `'bistatic'`.

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 `MechanicalScanLimits` 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 `ElectronicScanLimits` 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 `mountingAngles` 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° 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'`, 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'	MechnicalAngle
'Electronic'	ElectronicAngle
'Mechanical and Electronic'	MechnicalAngle + 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 vector

Types 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 ith waveform as the jth 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-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 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,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,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
`EmitterIndex`	Unique emitter index
`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 `UpdateInterval`.
`IsScanDone`	`IsScanDone` is `true` when the emitter has completed a scan.
`FieldOfView`	Field of view of emitter.
`MeasurementParameters`	`MeasurementParameters` is an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.

For more details on `MeasurementParameters`, see “Measurement Parameters” on page 3-162.

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 which has these fields:

Field	Definition

`Position`	Position of the GPS receiver in the local NED coordinate system specified as a real-valued 1-by-3 vector. Units are in meters.
`Velocity`	Velocity of the GPS receiver in the local NED coordinate system specified as a real-valued 1-by-3 vector. Units are in meters per second.
`Orientation`	Orientation of the INS with respect to the local NED coordinate system specified as a scalar quaternion or a 3-by-3 real-valued orthonormal frame rotation matrix. Defines the frame rotation from the local NED coordinate system to the current INS body coordinate system. 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
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters
`ObjectAttributes`	Additional information passed to tracker

`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 3-136.

`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
`SensorIndex`	Unique sensor index
`IsValidTime`	Valid detection time, returned as `0` or `1`. `IsValidTime` is `0` when detection updates are requested at times that are between update intervals specified by `UpdateInterval`.
`IsScanDone`	`IsScanDone` is `true` when the sensor has completed a scan.
`FieldOfView`	Field of view of sensor determines which objects fall within the sensor beam during object execution. The field of view is defined as a 2-by-1 vector of positive real values, [azfov;elfov].
`MeasurementParameters`	`MeasurementParameters` is an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current 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)
```

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)
```

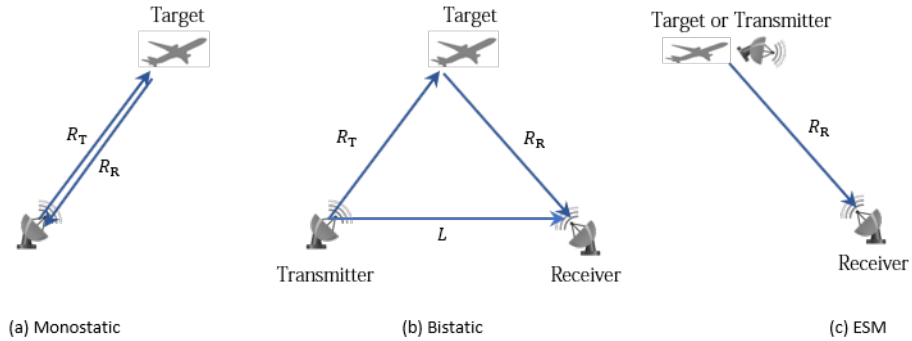
```
dets =

  1×1 cell array

    {1×1 objectDetection}


numDets =

     1


config =

  struct with fields:

              SensorIndex: 1
              IsValidTime: 1
               IsScanDone: 0
              FieldOfView: [1 5]
```

```
    MeasurementParameters: [1×1 struct]
```

Definitions

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 Δt 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 3-135).

Measurement Coordinates

<table>
<tr><th>DetectionCoordinates</th><th>Measurement and Measurement Noise Coordinates</th></tr>
<tr><td>'Scenario'
'Body'
'Sensor rectangular'</td><td>Coordinate Dependence on HasRangeRate<table><tr><th>HasRangeRate</th><th>Coordinates</th></tr><tr><td>true</td><td>[x; y; z; vx; vy; vz]</td></tr><tr><td>false</td><td>[x; y; z]</td></tr></table></td></tr>
<tr><td>'Sensor spherical'</td><td>Coordinate for 'monostatic' or 'bistatic' Detection Mode (Dependence on HasRangeRate and HasElevation)<table><tr><th>HasRangeRate</th><th>HasElevation</th><th>Coordinates</th></tr><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></table>Coordinate for 'ESM' Detection Mode (Dependence on HasElevation)<table><tr><th>HasElevation</th><th>Coordinates</th></tr><tr><td>true</td><td>[az; el]</td></tr><tr><td>false</td><td>[az]</td></tr></table></td></tr>
</table>

where `az`, `el`, `rng` and `rr` represent azimuth angle, elevation angle, range and range rate, respectively.

Measurement Parameters

The `MeasurementParameters` field 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”). The longest possible sequence of transformations is Sensor → Platform → Scenario. For example, if the detections are

reported in sensor spherical coordinates and `HasINS` is set to `false`, then the sequence consists of one transformation from sensor to platform. If `HasINS` is `true`, the sequence of transformations consists of two transformations – first to platform coordinates then to scenario coordinates. Trivially, if the detections are reported in platform rectangular coordinates and `HasINS` is set to `false`, the transformation consists only of the identity.

The structure fields 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 `'rectangular'`. When detections are reported in spherical coordinates, `Frame` is set `'spherical'` for the first `struct`.
`OriginPosition`	Position offset of the origin of the frame relative to the parent frame, represented as a 3-by-1 vector.
`OriginVelocity`	Velocity offset of the origin of the frame relative to the parent frame, represented as a 3-by-1 vector.
`Orientation`	3-by-3 real-valued orthonormal frame rotation matrix.
`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 indication if azimuth is included in the measurement.
`HasRange`	A logical scalar indication 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`	• Measured center frequency of the detected radar signal. Units are in Hz. • This attribute is present only when the `DetectionMode` property is set to `'ESM'` or `'Bistatic'`.
`Bandwidth`	• Measured bandwidth of the detected radar signal, Units are in Hz. • This attribute is present only when the `DetectionMode` property is set to `'ESM'` or `'Bistatic'`.

`WaveformType`	• Identifier of the waveform type that was classified by the ESM sensor for the detected signal. • This attribute is present only when the `DetectionMode` property is set to `'ESM'` or `'Bistatic'`.

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

Classes

`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? (MATLAB).

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 3-164 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 3-164 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 3-164 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 (MATLAB).

`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

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 sonar 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 sonar 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 sonar
`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 sonar 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 sonar 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 sonar 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 `MaxAllowedOccusion` 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.
`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]`.

Field	Description
`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 which has these fields:

Field	Definition
`Position`	Position of the GPS receiver in the local NED coordinate system specified as a real-valued 1-by-3 vector. Units are in meters.
`Velocity`	Velocity of the GPS receiver in the local NED coordinate system specified as a real-valued 1-by-3 vector. Units are in meters per second.
`Orientation`	Orientation of the INS with respect to the local NED coordinate system specified as a scalar quaternion or a 3-by-3 real-valued orthonormal frame rotation matrix. Defines the frame rotation from the local NED coordinate system to the current INS body coordinate system. 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
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters
`ObjectAttributes`	Additional information passed to tracker

`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 sensor beam during object execution.

Field	Description

`SensorIndex`	Unique sensor index
`IsValidTime`	Valid detection time, returned as `0` or `1`. `IsValidTime` is `0` when detection updates are requested at times that are between update intervals specified by `UpdateInterval`.
`IsScanDone`	`IsScanDone` is `true` when the sensor has completed a scan.
`FieldOfView`	Field of view of sensor determines which objects fall within the sensor beam during object execution. The field of view is defined as a 2-by-1 vector of positive real values, [azfov;elfov].
`MeasurementParameters`	`MeasurementParameters` is an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current 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)
```

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 =

  1×1 cell array

    {1×1 objectDetection}


numDets =

     1


config =

  struct with fields:

              SensorIndex: 1
              IsValidTime: 1
               IsScanDone: 0
              FieldOfView: [64.0108 64.0108]
```

```
    MeasurementParameters: [1×1 struct]
```

Definitions

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

<table>
<tr><th>DetectionCoordinates</th><th>Measurement and Measurement Noise Coordinates</th></tr>
<tr><td>'Scenario'
'Body'
'Sensor rectangular'</td><td>Coordinate Dependence on HasRangeRate<table><tr><th>HasRangeRate</th><th>Coordinates</th></tr><tr><td>true</td><td>[x; y; z; vx; vy; vz]</td></tr><tr><td>false</td><td>[x; y; z]</td></tr></table></td></tr>
<tr><td>'Sensor spherical'</td><td>Coordinate Dependence on HasRangeRate and HasElevation<table><tr><th>HasRangeRate</th><th>HasElevation</th><th>Coordinates</th></tr><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></table></td></tr>
</table>

Measurement Parameters

The `MeasurementParameters` field 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”). The longest possible sequence of transformations is Sensor → Platform → Scenario. For example, if the detections are reported in sensor spherical coordinates and `HasINS` is set to `false`, then the sequence consists of one transformation from sensor to platform. If `HasINS` is `true`, the sequence of transformations consists of two transformations – first to platform coordinates then to scenario coordinates. Trivially, if the detections are reported in platform rectangular coordinates and `HasINS` is set to `false`, the transformation consists only of the identity.

The structure fields 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 `'rectangular'`. When detections are reported in spherical coordinates, `Frame` is set `'spherical'` for the first `struct`.
`OriginPosition`	Position offset of the origin of the frame relative to the parent frame, represented as a 3-by-1 vector.
`OriginVelocity`	Velocity offset of the origin of the frame relative to the parent frame, represented as a 3-by-1 vector.
`Orientation`	3-by-3 real-valued orthonormal frame rotation matrix.
`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 indication if azimuth is included in the measurement.
`HasRange`	A logical scalar indication 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.
`SNR`	Detection signal-to-noise ratio in dB.

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]`

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
`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 infrared 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 infrared 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? (MATLAB).

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 3-190 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 3-190 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 3-190 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 (MATLAB).

`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

`[1;5]` | real-valued 2-by-1 vector of positive real-values

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 sonar are not detected. Units are in degrees.

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 `ElectronicScanLimits` 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 `mountingAngles` 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° 90°]. 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 vector

Types 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^{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-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 µPa 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 AzimuthResolutionproperty 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 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,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,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 `[-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`.

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
`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 `UpdateInterval`.

`IsScanDone`	`IsScanDone` is `true` when the emitter has completed a scan.
`FieldOfView`	Field of view of emitter.
`MeasurementParameters`	`MeasurementParameters` is an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.

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 which has these fields:

Field	**Definition**
`Position`	Position of the GPS receiver in the local NED coordinate system specified as a real-valued 1-by-3 vector. Units are in meters.
`Velocity`	Velocity of the GPS receiver in the local NED coordinate system specified as a real-valued 1-by-3 vector. Units are in meters per second.
`Orientation`	Orientation of the INS with respect to the local NED coordinate system specified as a scalar quaternion or a 3-by-3 real-valued orthonormal frame rotation matrix. Defines the frame rotation from the local NED coordinate system to the current INS body coordinate system. 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
`Time`	Measurement time
`Measurement`	Object measurements
`MeasurementNoise`	Measurement noise covariance matrix
`SensorIndex`	Unique ID of the sensor
`ObjectClassID`	Object classification
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters
`ObjectAttributes`	Additional information passed to tracker

`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
`SensorIndex`	Unique sensor index
`IsValidTime`	Valid detection time, returned as `0` or `1`. `IsValidTime` is `0` when detection updates are requested at times that are between update intervals specified by `UpdateInterval`.
`IsScanDone`	`IsScanDone` is `true` when the sensor has completed a scan.
`FieldOfView`	Field of view of sensor determines which objects fall within the sensor beam during object execution. The field of view is defined as a 2-by-1 vector of positive real values, [azfov;elfov].

`MeasurementParameters`	`MeasurementParameters` is an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current 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)
```

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 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 =

  1×1 cell array

    {1×1 objectDetection}


numDets =

     1


config =

  struct with fields:

              SensorIndex: 1
              IsValidTime: 1
               IsScanDone: 1
              FieldOfView: [1 5]
    MeasurementParameters: [1×1 struct]
```

Definitions

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

<table>
<tr><th>DetectionCoordinates</th><th>Measurement and Measurement Noise Coordinates</th></tr>
<tr><td>'Scenario'
'Body'
'Sensor rectangular'</td><td>Coordinate Dependence on HasRangeRate<table><tr><th>HasRangeRate</th><th>Coordinates</th></tr><tr><td>true</td><td>[x; y; z; vx; vy; vz]</td></tr><tr><td>false</td><td>[x; y; z]</td></tr></table></td></tr>
<tr><td>'Sensor spherical'</td><td>Coordinate Dependence on HasRangeRate and HasElevation<table><tr><th>HasRangeRate</th><th>HasElevation</th><th>Coordinates</th></tr><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></table></td></tr>
</table>

Measurement Parameters

The `MeasurementParameters` field 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”). The longest possible sequence of

transformations is Sensor → Platform → Scenario. For example, if the detections are reported in sensor spherical coordinates and `HasINS` is set to `false`, then the sequence consists of one transformation from sensor to platform. If `HasINS` is `true`, the sequence of transformations consists of two transformations – first to platform coordinates then to scenario coordinates. Trivially, if the detections are reported in platform rectangular coordinates and `HasINS` is set to `false`, the transformation consists only of the identity.

The structure fields 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 `'rectangular'`. When detections are reported in spherical coordinates, `Frame` is set `'spherical'` for the first `struct`.
`OriginPosition`	Position offset of the origin of the frame relative to the parent frame, represented as a 3-by-1 vector.
`OriginVelocity`	Velocity offset of the origin of the frame relative to the parent frame, represented as a 3-by-1 vector.
`Orientation`	3-by-3 real-valued orthonormal frame rotation matrix.
`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 indication if azimuth is included in the measurement.
`HasRange`	A logical scalar indication 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`	• Measured center frequency of the detected sonar signal. Units are in Hz. • This attribute is present only when the `DetectionMode` property is set to `'passive'`.

`Bandwidth`	• Measured bandwidth of the detected sonar signal, Units are in Hz. • This attribute is present only when the `DetectionMode` property is set to `'passive'`.
`WaveformType`	• Identifier of the waveform type that was classified by a passive sensor for the detected signal. • This attribute is present only when the `DetectionMode` property is set to `'passive'`.

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**
`ScanMode`	`'Electronic'`
`HasElevation`	`true`
`ElectronicScanLimits`	`[-45 45; -10 0]`

Rotator Scanning

This syntax sets these properties:

Property	**Value**
`ScanMode`	`'Electronic'`
`FieldOfView`	`[1:10]`

`HasElevation`	`false` or `true`
`ElevationResolution`	`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
`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? (MATLAB).

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° to +45° and in elevation from the horizon to 10° above the horizon. See “Raster Scanning” on page 3-312 for the properties set by this syntax.

`emitter = radarEmitter(EmitterIndex,'Rotator')` is a convenience syntax that creates a `radarEmitter` 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 “Rotator Scanning” on page 3-312 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° 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 “Sector Scanning” on page 3-312 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 (MATLAB).

`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 emitter
`[1;5]` | real-valued 2-by-1 vector of positive real-values

Fields 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 spanned by the emitter. Each component must lie in the interval *(0,180]*. Units are in degrees.

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 `MechanicalScanLimits` 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 `ElectronicScanLimits` 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 `mountingAngles` 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'`	`MechnicalAngle`
`'Electronic'`	`ElectronicAngle`
`'Mechanical and Electronic'`	`MechnicalAngle` + `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
`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.

Field	Description
`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`}

`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
`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 `UpdateInterval`.
`IsScanDone`	`IsScanDone` is `true` when the emitter has completed a scan.
`FieldOfView`	Field of view of emitter.

`MeasurementParameters`	`MeasurementParameters` is an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.

Data Types: `struct`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Model 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
```

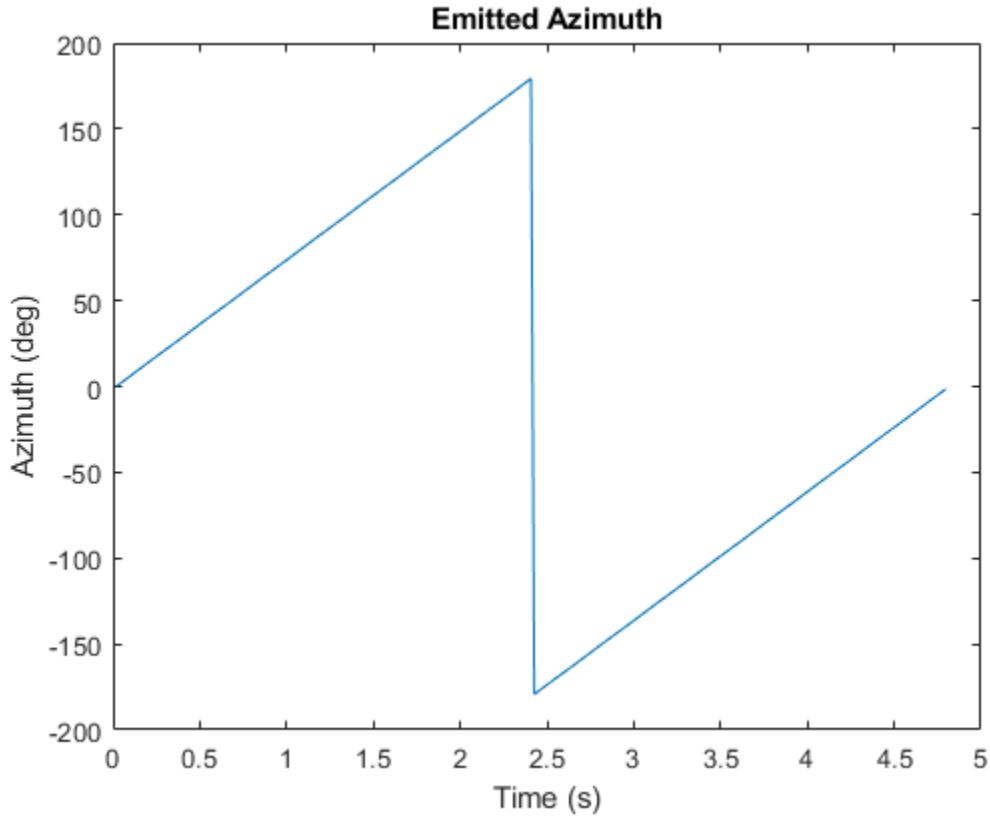
```
tower =

  Platform with properties:

       PlatformID: 1
          ClassID: 0
       Dimensions: [1x1 struct]
       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)')
```



Definitions

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

coustic 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? (MATLAB).

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° in azimuth from –45° to +45° and in elevation from the horizon to 10° above the horizon. See “Raster Scanning” on page 3-312 for the properties set by this syntax.

`emitter = sonarEmitter(EmitterIndex,'Rotator')` is a convenience syntax that creates a `sonarEmitter` object that mechanically scans 360° 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 3-312 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° azimuth sector from –45° to +45°. 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 3-312 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 (MATLAB).

`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 emitter
`[1;5]` | real-valued 2-by-1 vector of positive real-values

Fields 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 spanned by the emitter. Each component must lie in the interval *(0,180]*. Units are in degrees.

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 `ElectronicScanLimits` 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 `mountingAngles` 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° 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 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'`	`MechnicalAngle`
`'Electronic'`	`ElectronicAngle`
`'Mechanical and Electronic'`	`MechnicalAngle` + `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 µPa. Units are in dB//1 µPa.

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.

Field	Description
`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]`.

Field	Description
`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
`EmitterIndex`	Unique emitter index

`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 `UpdateInterval`.
`IsScanDone`	`IsScanDone` is `true` when the emitter has completed a scan.
`FieldOfView`	Field of view of emitter.
`MeasurementParameters`	`MeasurementParameters` is an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current emitter frame.

Data Types: `struct`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step Run System object algorithm
release Release resources and allow changes to System object property values and input characteristics
reset Reset internal states of System object

Examples

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}
```

Definitions

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
`ScanMode`	`'Electronic'`
`HasElevation`	`true`
`ElectronicScanLimits`	`[-45 45; -10 0]`

Rotator Scanning

This syntax sets these properties:

Property	Value
`ScanMode`	`'Electronic'`
`FieldOfView`	`[1:10]`
`HasElevation`	`false` or `true`
`ElevationResolution`	`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

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? (MATLAB).

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 NED coordinate system (m)
`[0 0 0]` (default) | 3-element row vector

Position state in the local NED coordinate system in meters, specified as a three-element row vector.

Tunable: Yes

Data Types: `single` | `double`

`Velocity` — Velocity state in local NED coordinate system (m/s)
`[0 0 0]` (default) | 3-element row vector

Velocity state in the local NED coordinate system in m/s, specified as a three-element row vector.

Tunable: Yes

Data Types: `single` | `double`

`Orientation` — Orientation state in local NED coordinate system
`quaternion(1,0,0,0)` (default) | scalar quaternion | 3-by-3 real matrix

Orientation state in the local NED coordinate system, specified as a scalar quaternion or 3-by-3 real matrix. The orientation is a frame rotation from the local NED 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^2)

`[0 0 0]` (default) | three-element row vector

Acceleration state in m/s^2, 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/s²)

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 NED coordinate system (m)

N-by-3 matrix

Position in the local NED 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 NED coordinate system

N-element quaternion column vector | 3-by-3-by-*N* real array

Orientation in the local NED 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 NED 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 NED coordinate system (m/s)

N-by-3 matrix

Velocity in the local NED 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 NED coordinate system (m/s^2)

N-by-3 matrix

Acceleration in the local NED 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 NED coordinate system (rad/s)
N-by-3 matrix

Angular velocity in the local NED 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

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(bodyAccelerat
```

```
position =

     0     0     0


orientation =

  quaternion


     1 + 0i + 0j + 0k


velocity =

     0     0     0


acceleration =

     5     5     0


angularVelocity =

     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 NED 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)
rotatedAngularVelocity = rotatepoint(orientation,angularVelocity)
```

```
rotatedAcceleration =

     5     5     0


rotatedAngularVelocity =

     0     0     1
```

The `kinematicTrajectory` System object™ enables you to modify the trajectory state through the properties. Set the position to [0,0,0] and then call the object with a specified acceleration and angular velocity in the body coordinate system. For illustrative purposes, clone the `trajectory` object before modifying the `Position` property. Call both objects and observe that the positions diverge.

```
trajectoryClone = clone(trajectory);
trajectory.Position = [0,0,0];

position = trajectory(bodyAcceleration,bodyAngularVelocity)
clonePosition = trajectoryClone(bodyAcceleration,bodyAngularVelocity)
```

```
position =

     0     0     0


clonePosition =

   1.0e-03 *

    0.2500    0.2500         0
```

Create Oscillating Trajectory

This example shows how to create a trajectory oscillating along the North axis of a local NED coordinate system using the `kinematicTrajectory` System object™.

Create a default `kinematicTrajectory` object. The default initial orientation is aligned with the local NED coordinate system.

```
traj = kinematicTrajectory
```

```
traj =

  kinematicTrajectory with properties:

               SampleRate: 100
                 Position: [0 0 0]
              Orientation: [1x1 quaternion]
                 Velocity: [0 0 0]
       AccelerationSource: 'Input'
    AngularVelocitySource: 'Input'
```

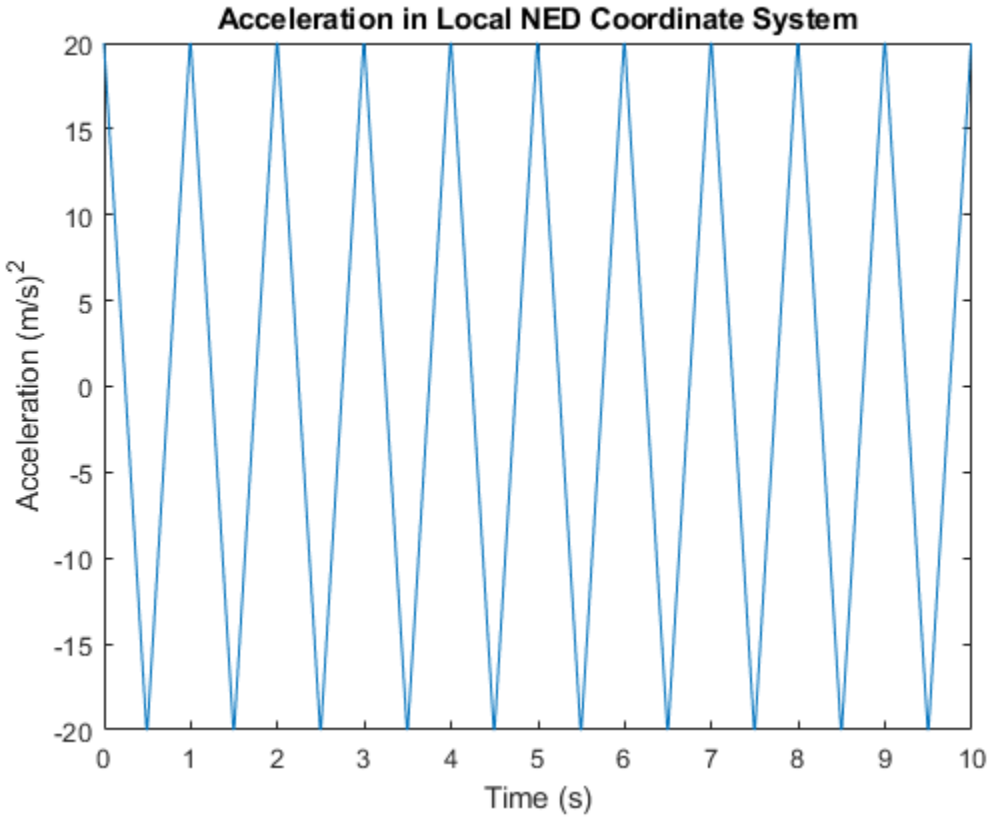
Define a trajectory for a duration of 10 seconds consisting of rotation around the East axis (pitch) and an oscillation along North axis of the local NED coordinate system. Use the default `kinematicTrajectory` sample rate.

```
fs = traj.SampleRate;
duration = 10;
```

```
numSamples = duration*fs;

cyclesPerSecond = 1;
samplesPerCycle = fs/cyclesPerSecond;
numCycles = ceil(numSamples/samplesPerCycle);
maxAccel = 20;

triangle = [linspace(maxAccel,1/fs-maxAccel,samplesPerCycle/2), ...
    linspace(-maxAccel,maxAccel-(1/fs),samplesPerCycle/2)]';
oscillation = repmat(triangle,numCycles,1);
oscillation = oscillation(1:numSamples);

accNED = [zeros(numSamples,2),oscillation];

angVelNED = zeros(numSamples,3);
angVelNED(:,2) = 2*pi;
```

Plot the acceleration control signal.

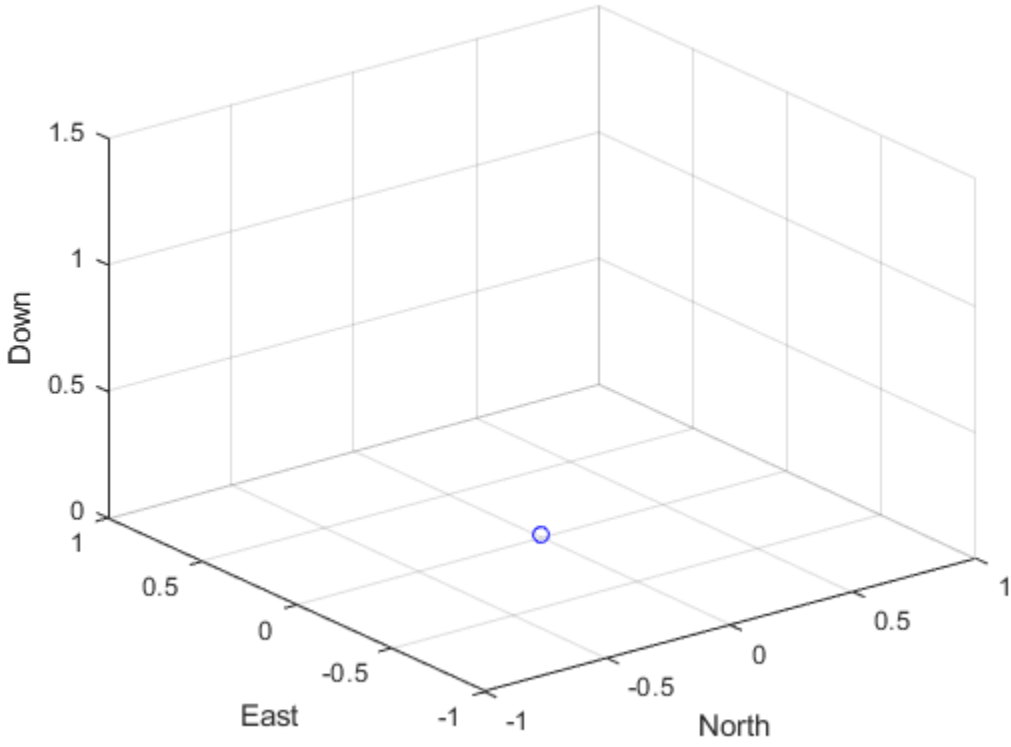
```
timeVector = 0:1/fs:(duration-1/fs);

figure(1)
plot(timeVector,oscillation)
xlabel('Time (s)')
ylabel('Acceleration (m/s)^2')
title('Acceleration in Local NED Coordinate System')
```

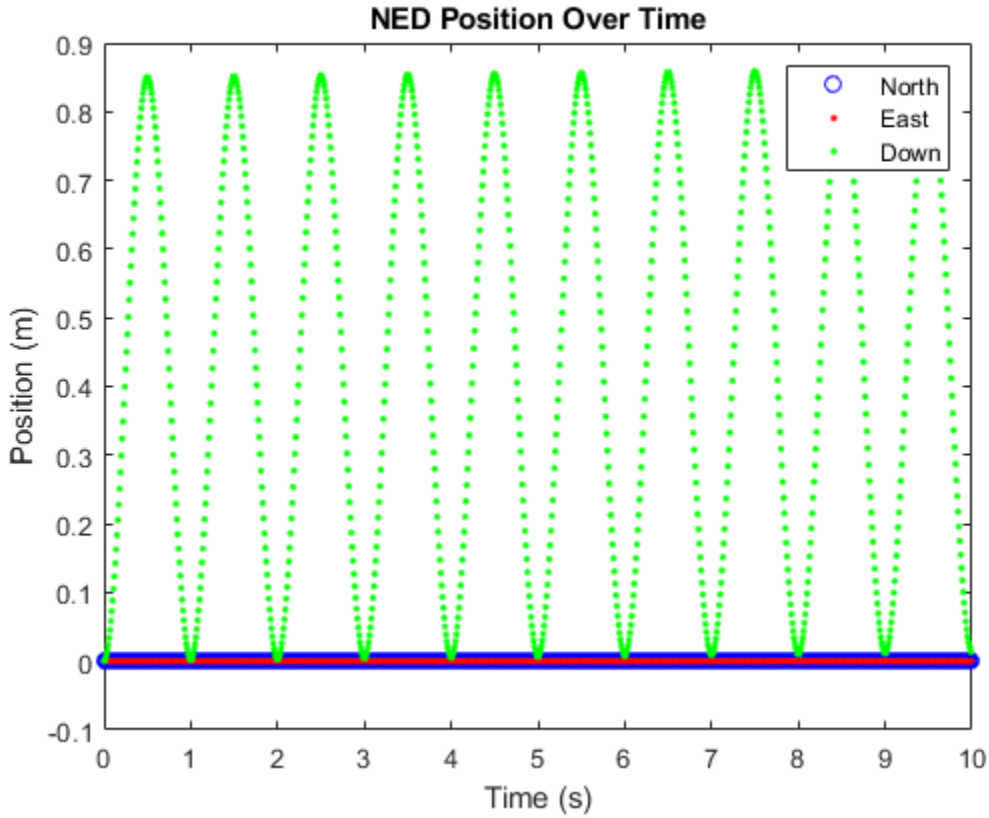


Generate the trajectory sample-by-sample in a loop. The `kinematicTrajectory` System object assumes the acceleration and angular velocity inputs are in the local sensor body coordinate system. Rotate the acceleration and angular velocity control signals from the NED coordinate system to the sensor body coordinate system using `rotateframe` and the `Orientation` state. Update a 3-D plot of the position at each time. Add `pause` to mimic real-time processing. Once the loop is complete, plot the position over time. Rotating the `accNED` and `angVelNED` control signals to the local body coordinate system assures the motion stays along the Down axis.

```
figure(2)
plotHandle = plot3(traj.Position(1),traj.Position(2),traj.Position(3),'bo');
grid on
xlabel('North')
```

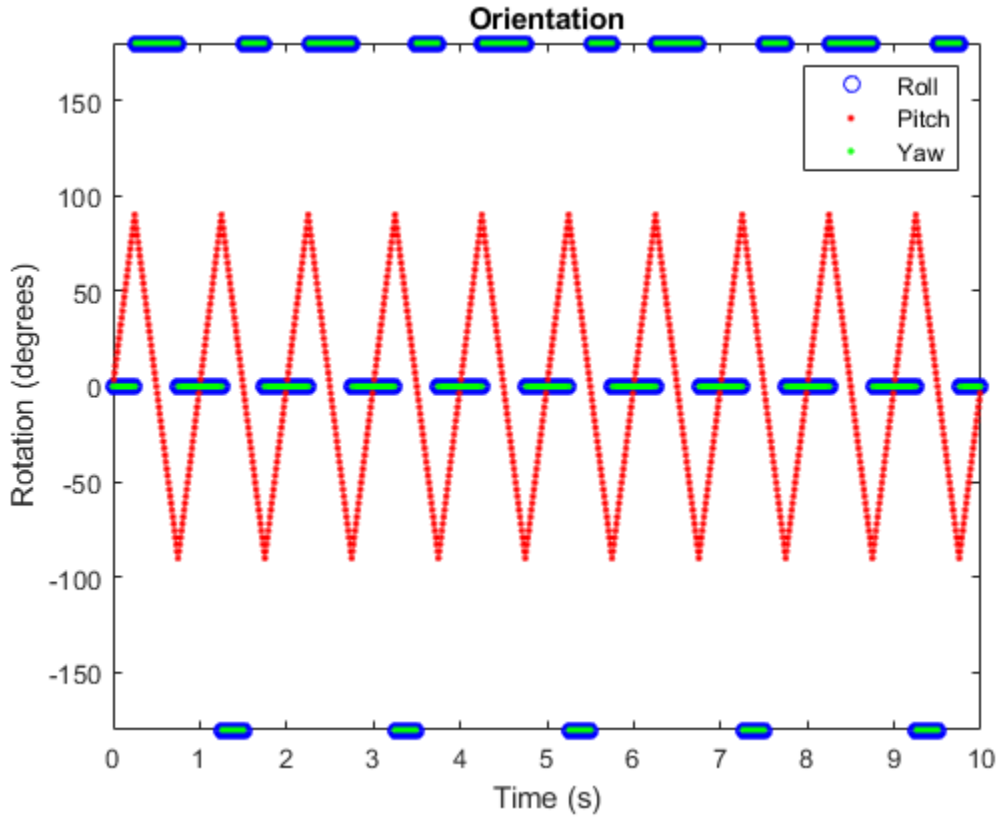
```
ylabel('East')
zlabel('Down')
axis([-1 1 -1 1 0 1.5])
hold on

q = ones(numSamples,1,'quaternion');
for ii = 1:numSamples
     accBody = rotateframe(traj.Orientation,accNED(ii,:));
     angVelBody = rotateframe(traj.Orientation,angVelNED(ii,:));

    [pos(ii,:),q(ii),vel,ac] = traj(accBody,angVelBody);

    set(plotHandle,'XData',pos(ii,1),'YData',pos(ii,2),'ZData',pos(ii,3))

    pause(1/fs)
end

figure(3)
plot(timeVector,pos(:,1),'bo',...
     timeVector,pos(:,2),'r.',...
     timeVector,pos(:,3),'g.')
xlabel('Time (s)')
ylabel('Position (m)')
title('NED Position Over Time')
legend('North','East','Down')
```


1.5
1
0.5
0
Down
1
0.5
0
-0.5
-1
East
-1
-0.5
0
0.5
1
North



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],'eulerd','zyx','frame');

trajectory = kinematicTrajectory('SampleRate',fs, ...
    'Velocity',initVel, ...
    'Position',initPos, ...
    'Orientation',initOrientation);
```

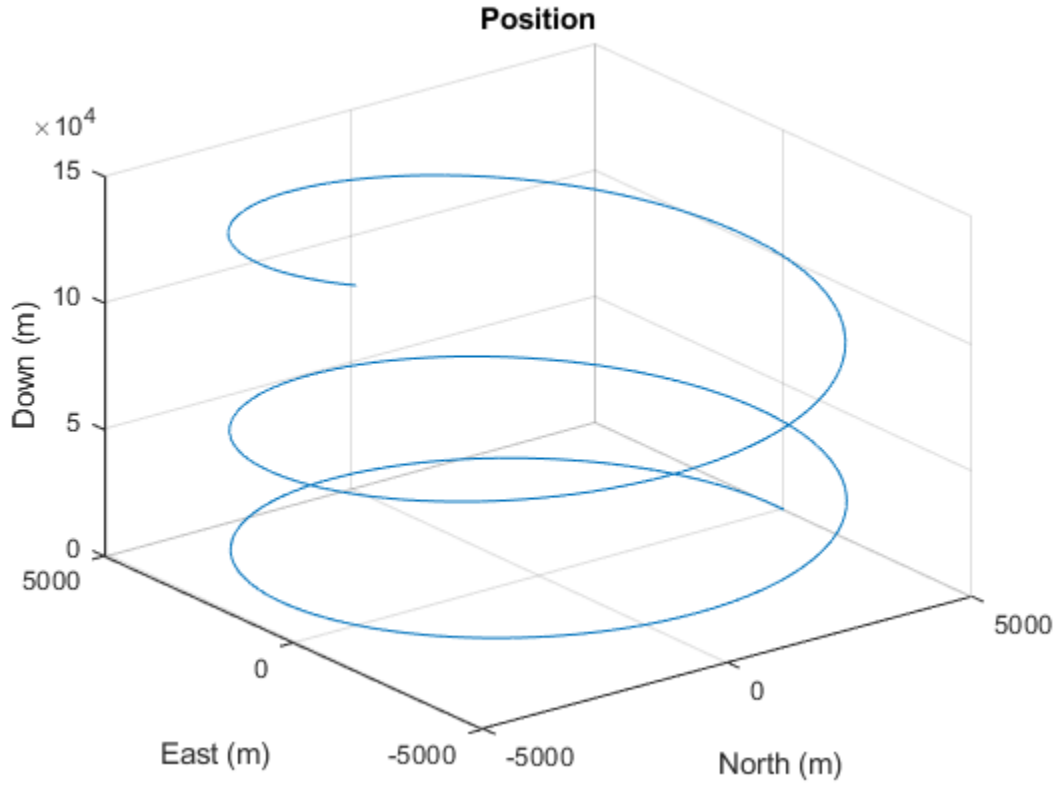
Specify a constant acceleration and angular velocity in the body coordinate system. Rotate the body frame to account for the pitch.

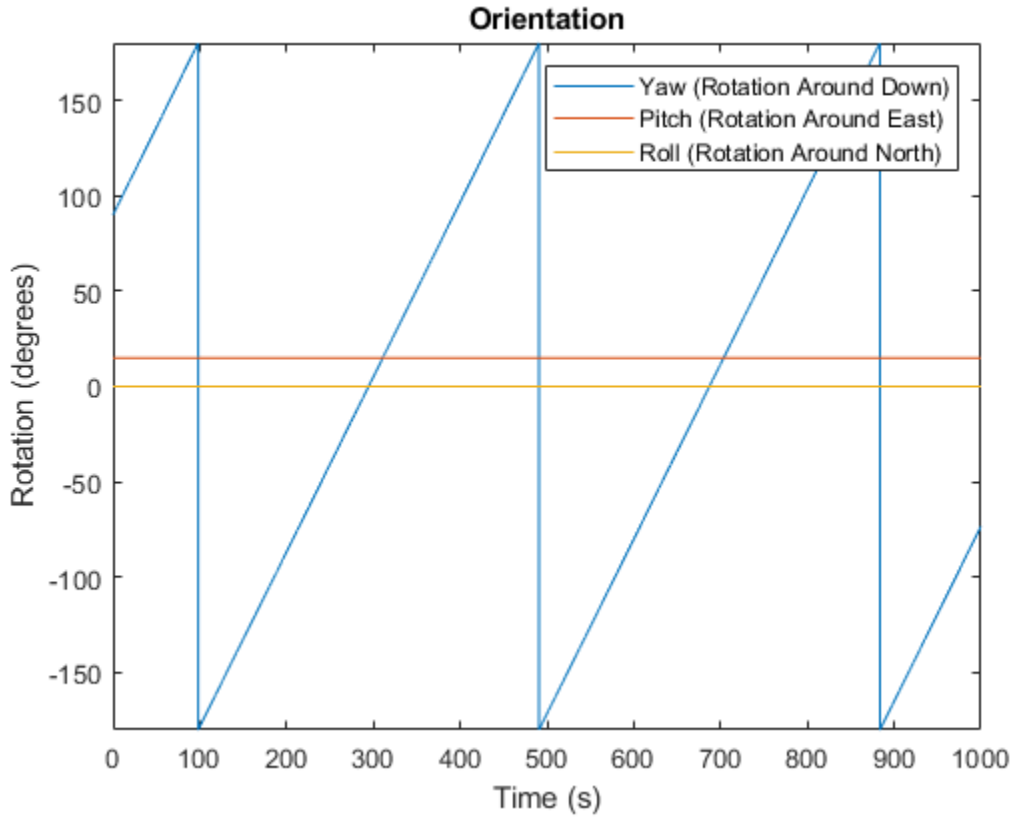
```
accBody = zeros(N,3);
accBody(:,2) = speed^2/radius;
accBody(:,3) = 0.2;

angVelBody = zeros(N,3);
angVelBody(:,3) = speed/radius;

pitchRotation = quaternion([0,pitch,0],'eulerd','zyx','frame');
angVelBody = rotateframe(pitchRotation,angVelBody);
accBody = rotateframe(pitchRotation,accBody);
```

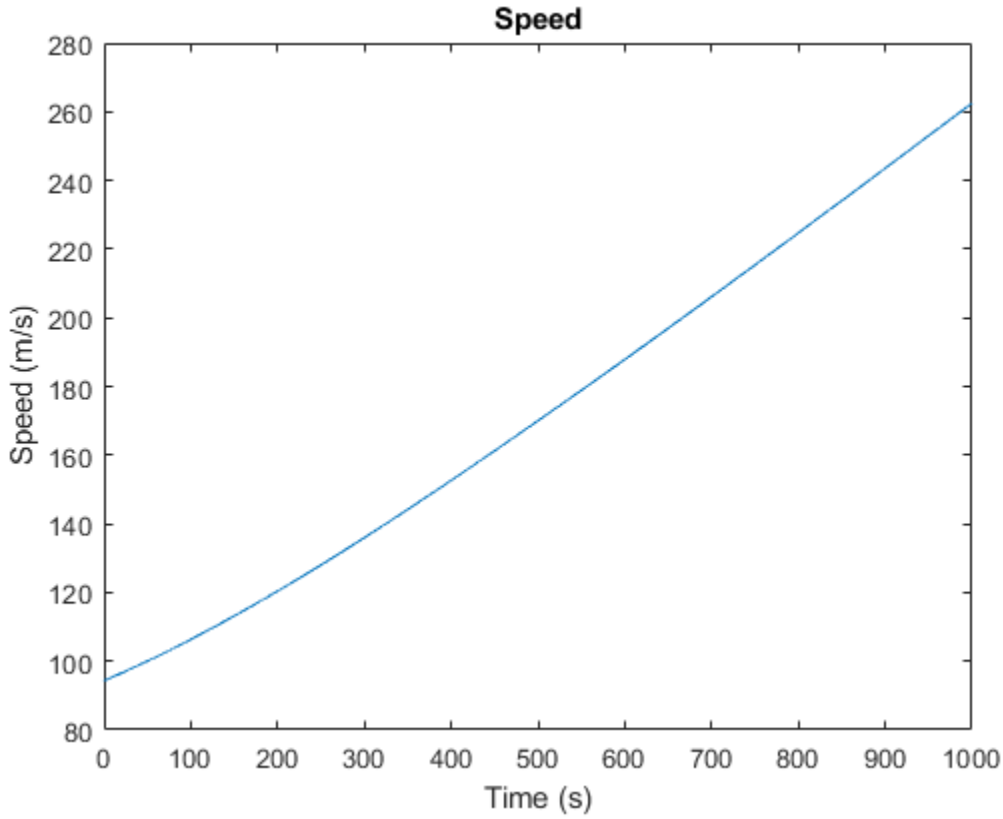
Call `trajectory` with the specified acceleration and angular velocity in the body coordinate system. Plot the position, orientation, and speed over time.

```
[position, orientation, velocity] = trajectory(accBody,angVelBody);

eulerAngles = eulerd(orientation,'ZYX','frame');
speed = sqrt(sum(velocity.^2,2));

timeVector = (0:(N-1))/fs;

figure(1)
plot3(position(:,1),position(:,2),position(:,3))
xlabel('North (m)')
ylabel('East (m)')
```

```
zlabel('Down (m)')
title('Position')
grid on

figure(2)
plot(timeVector,eulerAngles(:,1),...
     timeVector,eulerAngles(:,2),...
     timeVector,eulerAngles(:,3))
axis([0,duration,-180,180])
legend('Yaw (Rotation Around Down)','Pitch (Rotation Around East)','Roll (Rotation Arou
xlabel('Time (s)')
ylabel('Rotation (degrees)')
title('Orientation')

figure(3)
plot(timeVector,speed)
xlabel('Time (s)')
ylabel('Speed (m/s)')
title('Speed')
```


Position
×10^4
15
10
5
0
Down (m)
5000
0
East (m)
-5000
-5000
0
5000
North (m)


Orientation
Yaw (Rotation Around Down)
Pitch (Rotation Around East)
Roll (Rotation Around North)
Rotation (degrees)
150
100
50
0
-50
-100
-150
0
100
200
300
400
500
600
700
800
900
1000
Time (s)



Generate Spiraling Circular Trajectory with No Inputs

Define a constant angular velocity and constant acceleration that describe a spiraling circular trajectory.

```
Fs = 100;
r = 10;
speed = 2.5;
initialYaw = 90;

initPos = [r 0 0];
initVel = [0 speed 0];
```

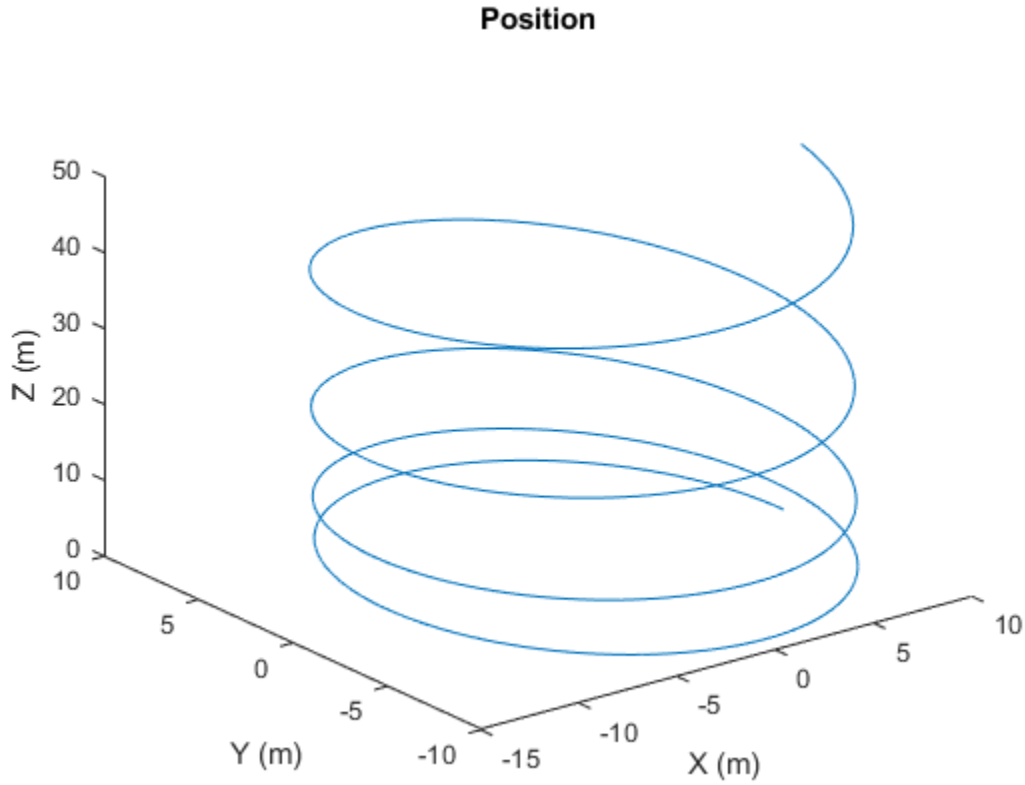
```
initOrient = quaternion([initialYaw 0 0], 'eulerd', 'ZYX', 'frame');

accBody = [0 speed^2/r 0.01];
angVelBody = [0 0 speed/r];
```

Create a kinematic trajectory object.

```
traj = kinematicTrajectory('SampleRate',Fs, ...
    'Position',initPos, ...
    'Velocity',initVel, ...
    'Orientation',initOrient, ...
    'AccelerationSource','Property', ...
    'Acceleration',accBody, ...
    'AngularVelocitySource','Property', ...
    'AngularVelocity',angVelBody);
```

Call the kinematic trajectory object in a loop and log the position output. Plot the position over time.

```
N = 10000;
pos = zeros(N, 3);
for i = 1:N
    pos(i,:) = traj();
end

plot3(pos(:,1), pos(:,2), pos(:,3))
title('Position')
xlabel('X (m)')
ylabel('Y (m)')
zlabel('Z (m)')
```



Extended Capabilities

C/C++ Code Generation

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

Usage notes and limitations:

“System Objects in MATLAB Code Generation” (MATLAB Coder)

See Also

`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? (MATLAB).

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.

`Waypoints` — Positions in the NED coordinate system (m)

N-by-3 matrix

Positions in the NED coordinate system in meters, specified as an *N*-by-3 matrix. The columns of the matrix correspond to the North, East, and Down 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 NED coordinate system at each waypoint (m/s)

N-by-3 matrix

Velocity in the NED 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 North, East, and Down axes, respectively. 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 and TimeOfArrival properties.

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`

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 (MATLAB).

`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 in output
1 (default) | positive scalar integer

Number of samples in the output, specified as a positive scalar integer.

Tunable: No

Data Types: `double`

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 NED coordinate system (m)
M-by-3 matrix

Position in the local NED 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 NED coordinate system
M-element quaternion column vector | 3-by-3-by-*M* real array

Orientation in the local NED 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 NED coordinate system to the current body coordinate system.

M is specified by the SamplesPerFrame property.

Data Types: `double`

`velocity` — Velocity in local NED coordinate system (m/s)
M-by-3 matrix

Velocity in the local NED 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 NED coordinate system (m/s^2)
M-by-3 matrix

Acceleration in the local NED 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 NED coordinate system (rad/s)
M-by-3 matrix

Angular velocity in the local NED 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

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
isDone	End-of-data status

Examples

Create Default `waypointTrajectory`

```
trajectory = waypointTrajectory
```

```
trajectory =
  waypointTrajectory with properties:

         SampleRate: 100
    SamplesPerFrame: 1
```

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=2×2 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.

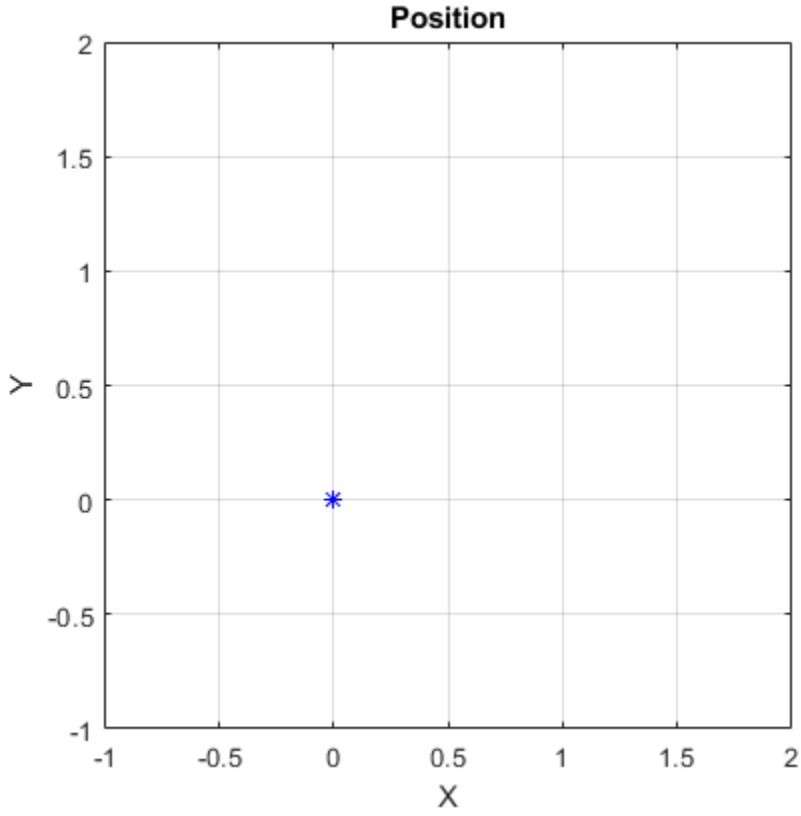
Create a square trajectory by defining the vertices of the square. Define the orientation at each waypoint as pointing in the direction of motion. Specify a 1 Hz sample rate and use the default `SamplesPerFrame` of 1.

```
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')
```

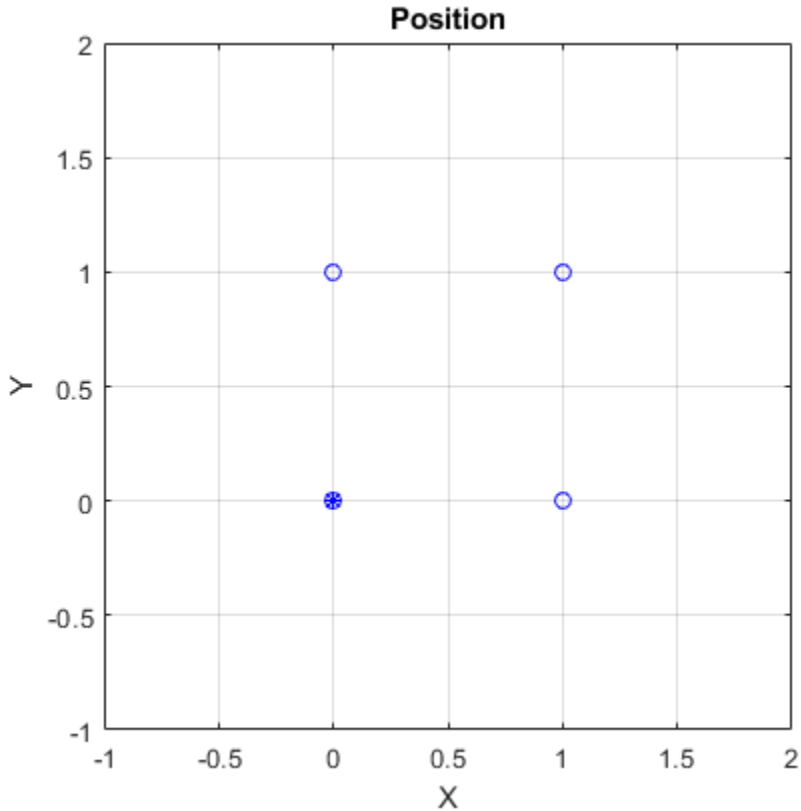
```
grid on
hold on
```



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')
```

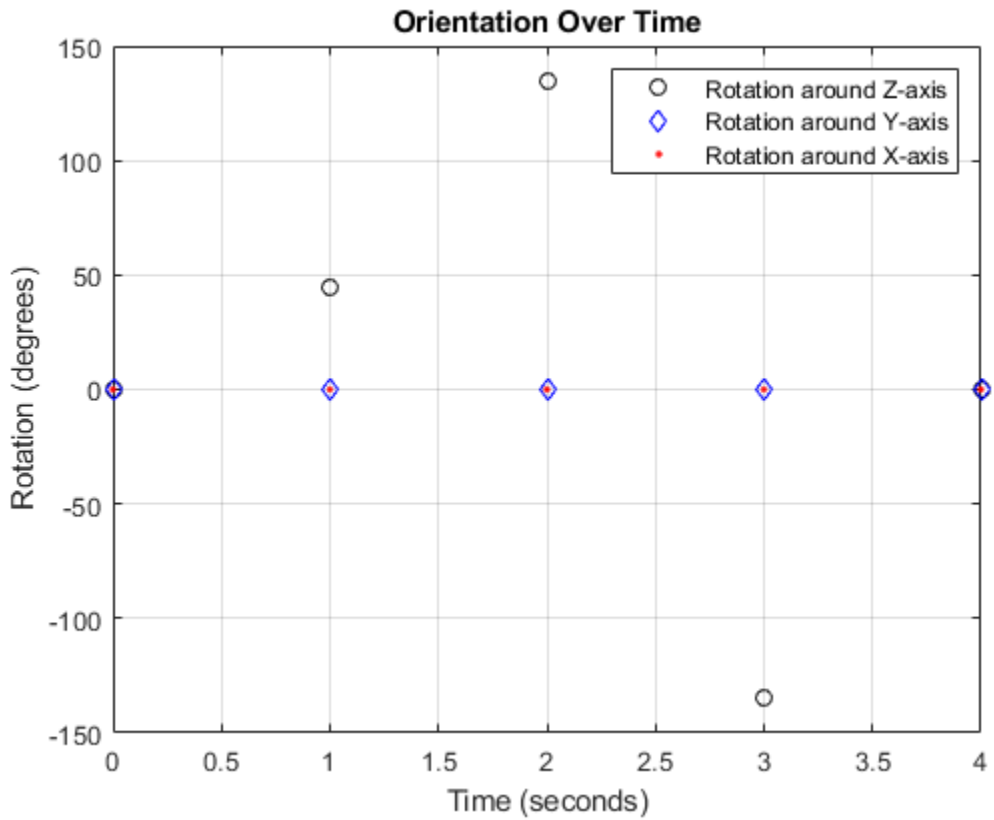
```
    pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
    count = count + 1;
end
hold off
```



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
```

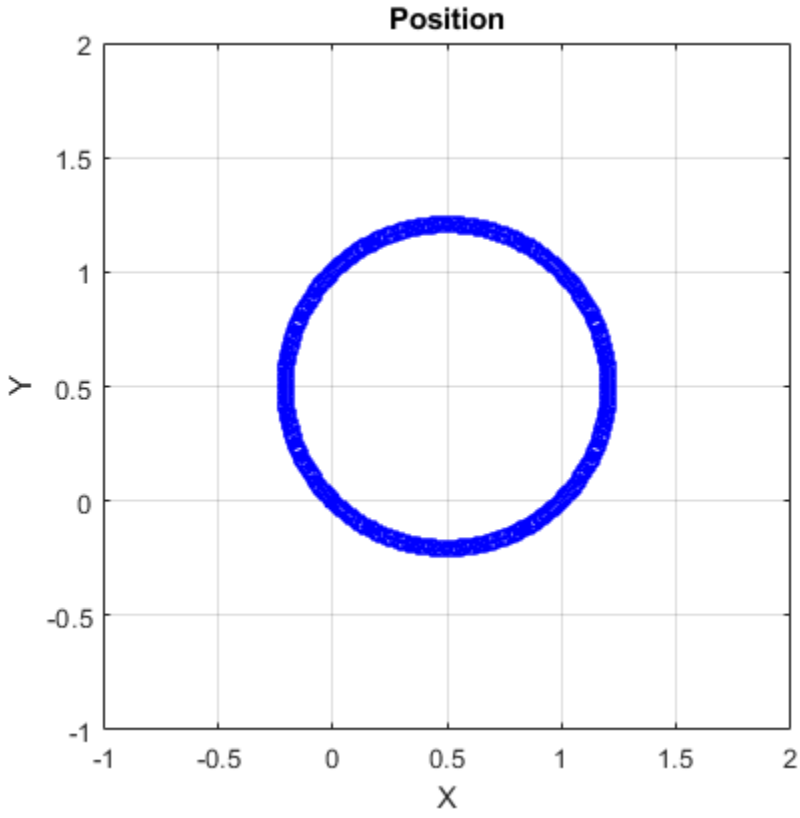


So far, the trajectory object has only output the waypoints that were specified during construction. To interpolate between waypoints, increase the sample rate to a rate faster than the time of arrivals of the waypoints. Set the `trajectory` sample rate to 100 Hz and call `reset`.

```
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(trajectory.SamplesPerFrame/trajectory.SampleRate)
   count = count + 1;
end
hold off
```

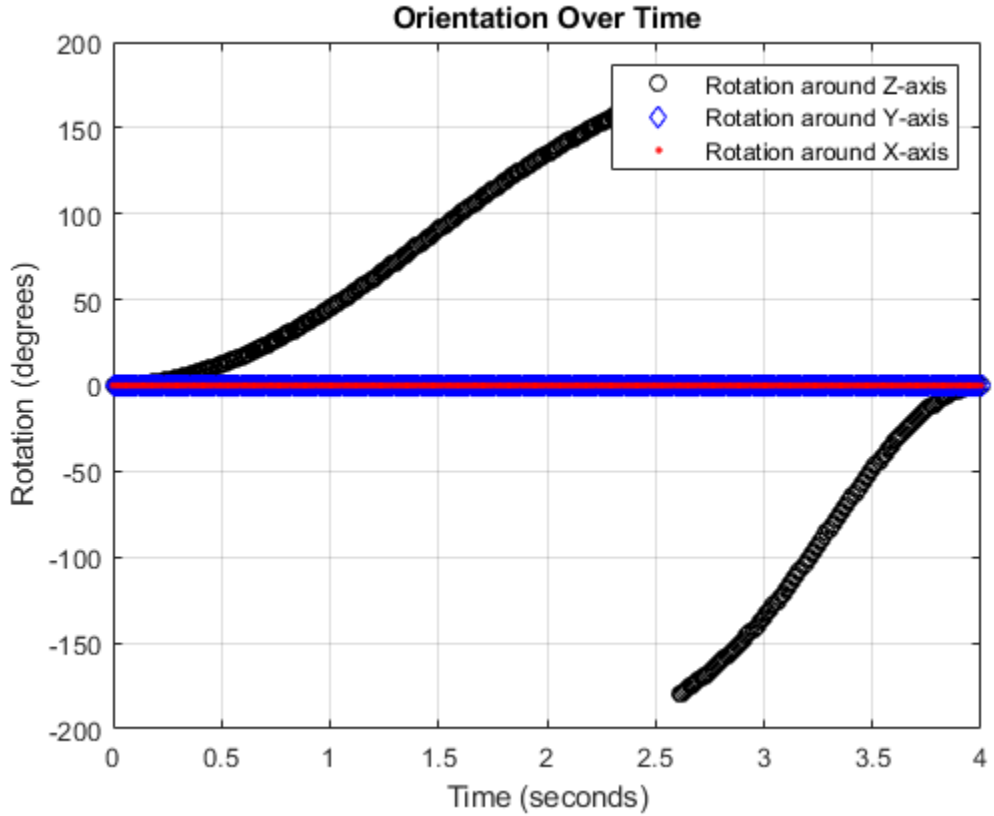


The trajectory output now appears circular. This is because the `waypointTrajectory` System object™ minimizes the acceleration and angular velocity when interpolating, which results in smoother, more realistic motions in most scenarios.

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/trajectory.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.

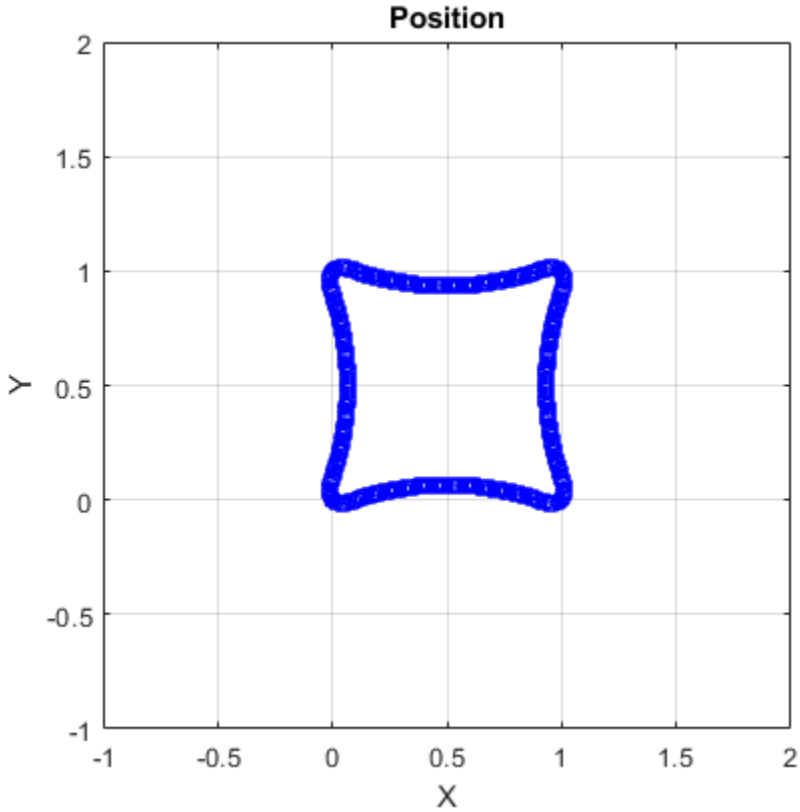
```
                % Time, Waypoint, Orientation
trajectoryInfo = [0,   0,0,0,    0,0,0; ... % Initial position
                  0.1, 0,0.1,0,  0,0,0; ...
```

```
                    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(trajectory.SamplesPerFrame/trajectory.SampleRate)
   count = count+1;
```

```
end
hold off
```

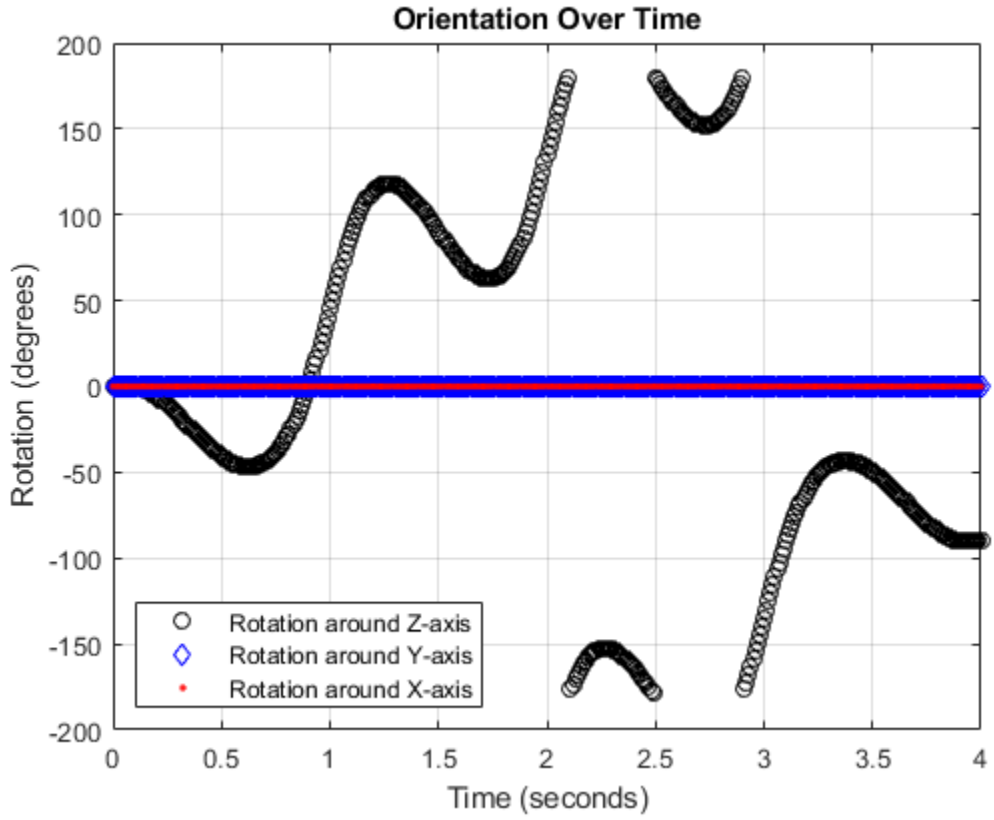


The trajectory output now appears more square-like, especially around the vertices with waypoints.

Convert the orientation quaternions to Euler angles for easy interpretation, and then plot orientation over time.

```
figure(2)
eulerAngles = eulerd([orientation(1);orientationLog],'ZYX','frame');
t = 0:1/trajectory.SampleRate:4;
eulerAngles = plot(t,eulerAngles(:,1),'ko', ...
                   t,eulerAngles(:,2),'bd', ...
```

```
                    t,eulerAngles(:,3),'r.');
title('Orientation Over Time')
legend('Rotation around Z-axis', ...
       'Rotation around Y-axis', ...
       'Rotation around X-axis', ...
       'Location', 'SouthWest')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on
```



Create Arc Trajectory

This example shows how to create an arc trajectory using the `waypointTrajectory` System object™. `waypointTrajectory` creates a path through specified waypoints that minimizes acceleration and angular velocity. After creating an arc trajectory, you restrict the trajectory to be within preset bounds.

Create an Arc Trajectory

Define a constraints matrix consisting of waypoints, times of arrival, and orientation for an arc trajectory. The generated trajectory passes through the waypoints at the specified times with the specified orientation. The `waypointTrajectory` System object requires orientation to be specified using quaternions or rotation matrices. Convert the Euler angles saved in the constrains matrix to quaternions when specifying the `Orientation` property.

```
          % Arrival, Waypoints, Orientation
constraints = [0,    20,20,0,    90,0,0;
               3,    50,20,0,    90,0,0;
               4,    58,15.5,0,  162,0,0;
               5.5,  59.5,0,0    180,0,0];

trajectory = waypointTrajectory(constraints(:,2:4), ...
    'TimeOfArrival',constraints(:,1), ...
    'Orientation',quaternion(constraints(:,5:7),'eulerd','ZYX','frame'));
```

Call `waypointInfo` on `trajectory` to return a table of your specified constraints. The creation properties `Waypoints`, `TimeOfArrival`, and `Orientation` are variables of the table. The table is convenient for indexing while plotting.

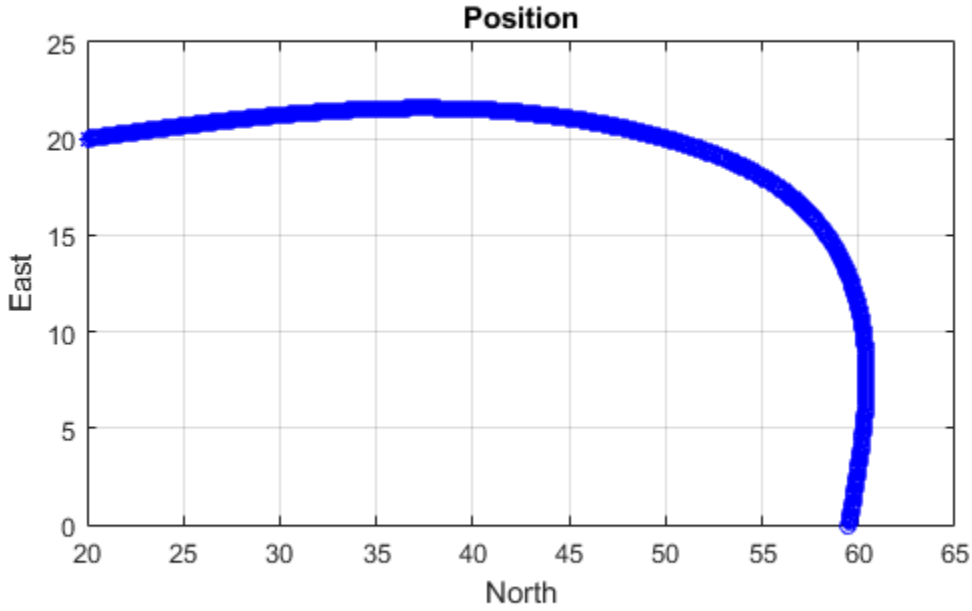
```
tInfo = waypointInfo(trajectory)
```

```
tInfo =

  4x3 table

    TimeOfArrival         Waypoints            Orientation
    _____________    ____________________    _________________

          0            20      20       0    [1x1 quaternion]
          3            50      20       0    [1x1 quaternion]
          4            58    15.5       0    [1x1 quaternion]
```

```
    5.5         59.5       0       0    [1x1 quaternion]
```

The trajectory object outputs the current position, velocity, acceleration, and angular velocity at each call. Call `trajectory` in a loop and plot the position over time. Cache the other outputs.

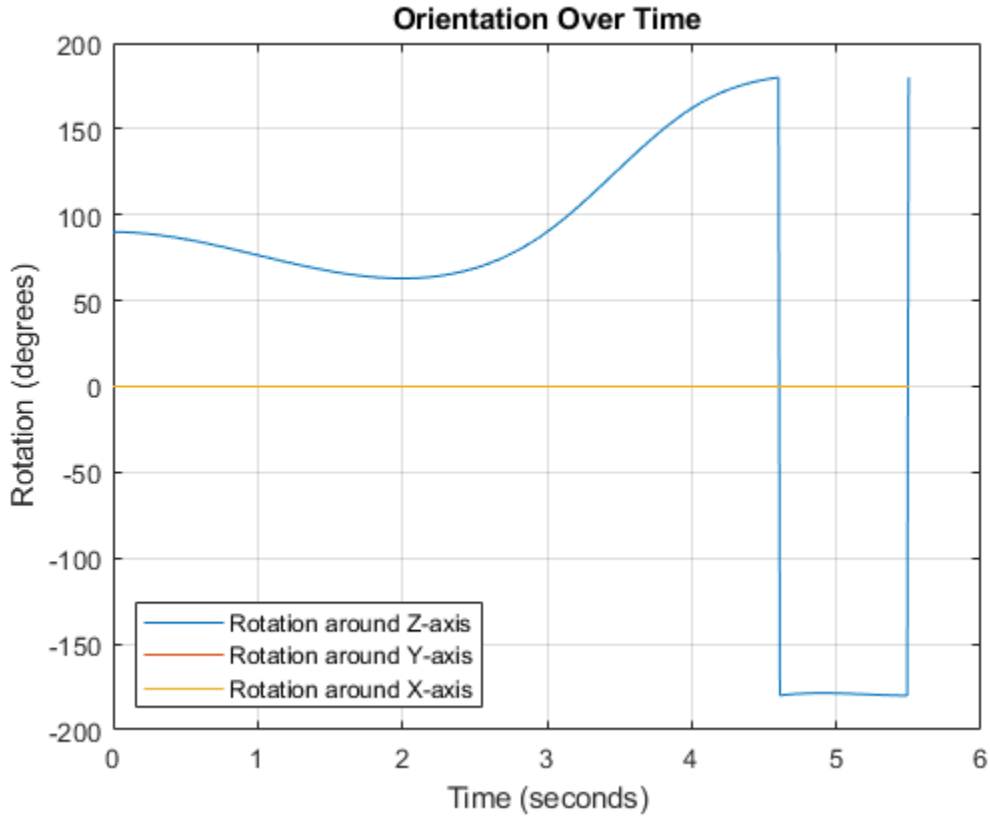
```
figure(1)
plot(tInfo.Waypoints(1,1),tInfo.Waypoints(1,2),'b*')
title('Position')
axis([20,65,0,25])
xlabel('North')
ylabel('East')
grid on
daspect([1 1 1])
hold on

orient = zeros(tInfo.TimeOfArrival(end)*trajectory.SampleRate,1,'quaternion');
vel = zeros(tInfo.TimeOfArrival(end)*trajectory.SampleRate,3);
acc = vel;
angVel = vel;

count = 1;
while ~isDone(trajectory)
   [pos,orient(count),vel(count,:),acc(count,:),angVel(count,:)] = trajectory();

   plot(pos(1),pos(2),'bo')

   pause(trajectory.SamplesPerFrame/trajectory.SampleRate)
   count = count + 1;
end
```

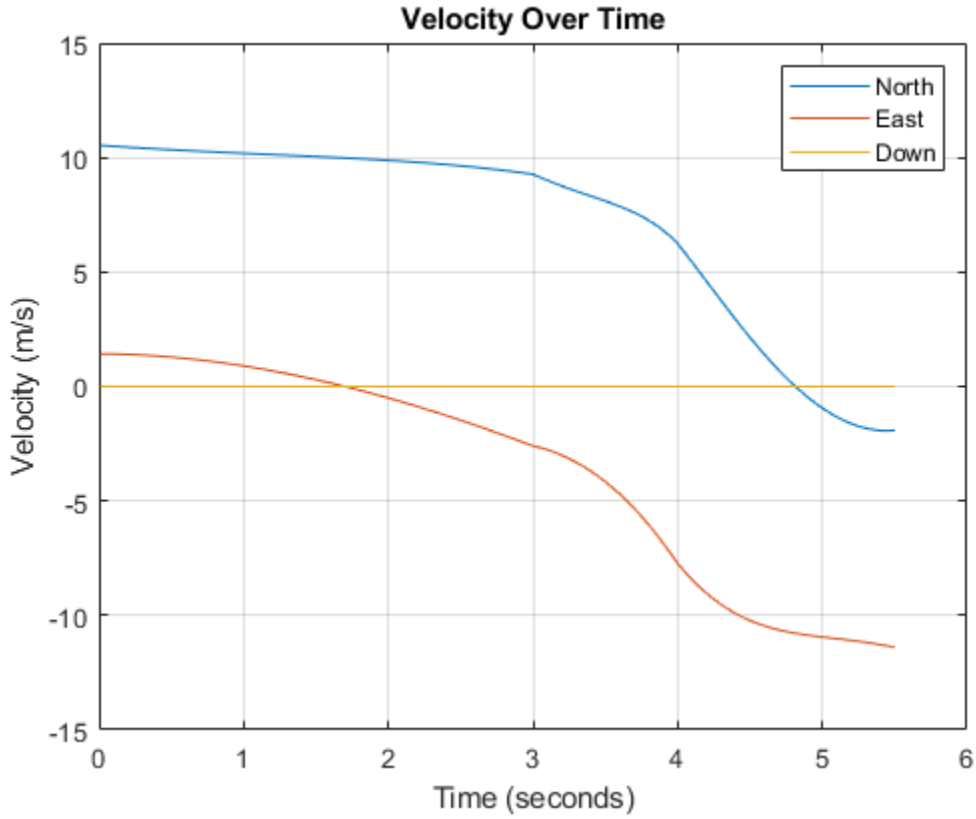


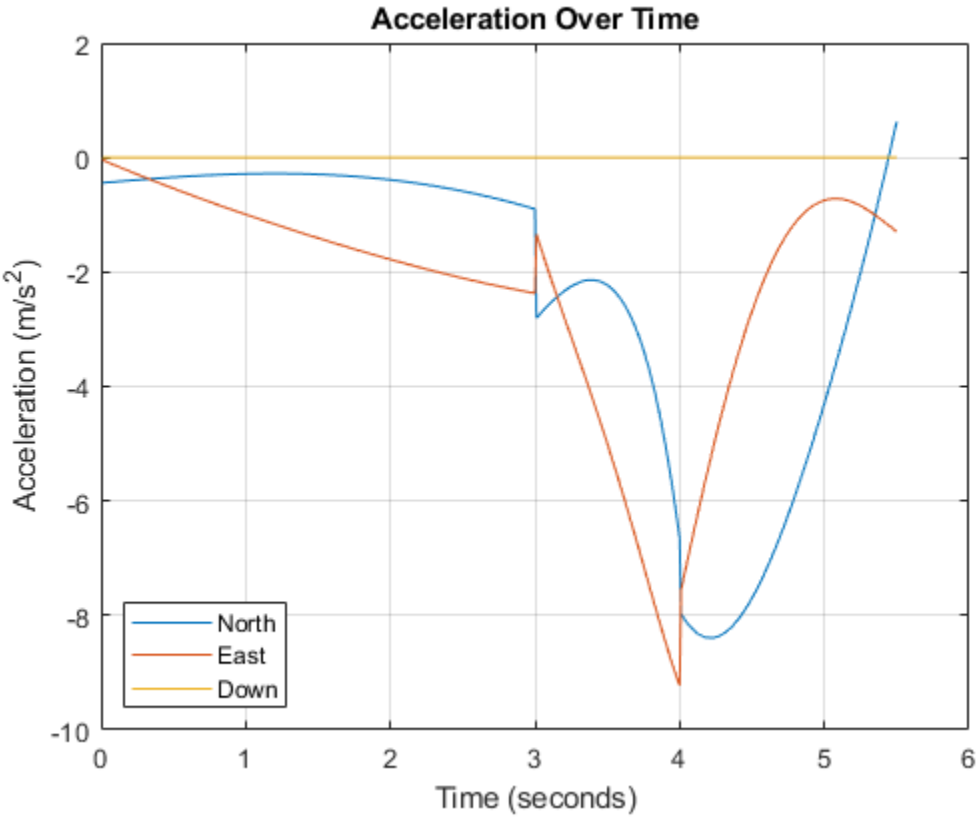
Inspect the orientation, velocity, acceleration, and angular velocity over time. The `waypointTrajectory` System object™ creates a path through the specified constraints that minimized acceleration and angular velocity.

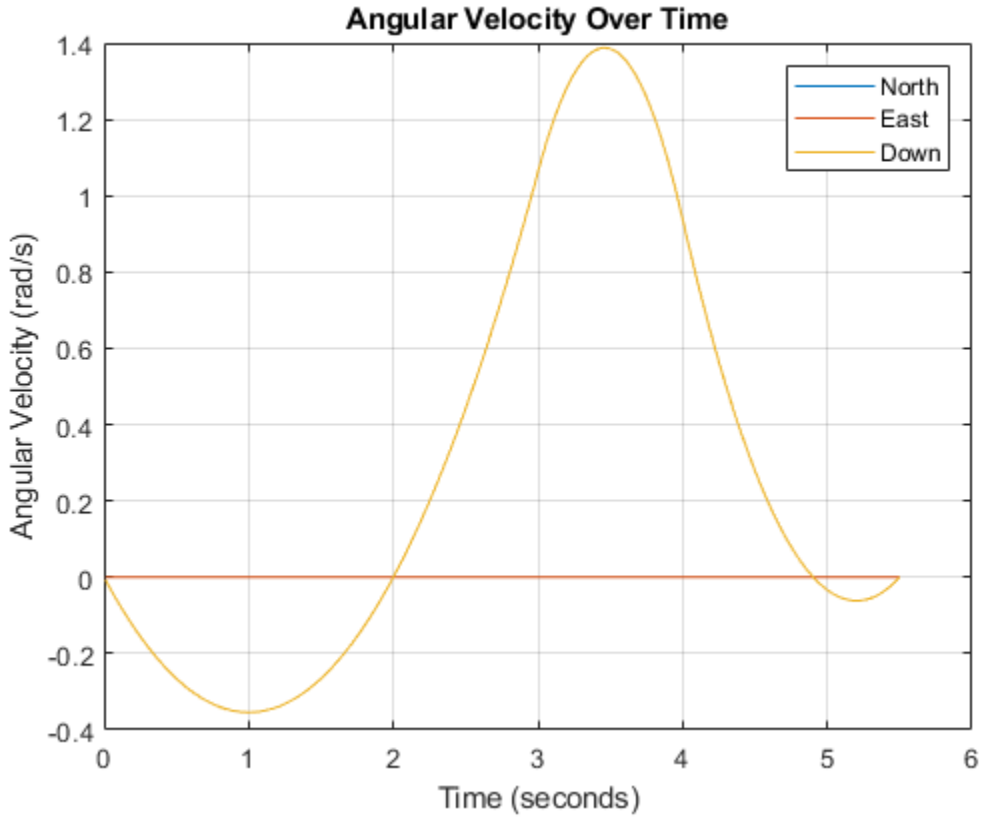
```
figure(2)
timeVector = 0:(1/trajectory.SampleRate):tInfo.TimeOfArrival(end);
eulerAngles = eulerd([tInfo.Orientation{1};orient],'ZYX','frame');
plot(timeVector,eulerAngles(:,1), ...
     timeVector,eulerAngles(:,2), ...
     timeVector,eulerAngles(:,3));
title('Orientation Over Time')
legend('Rotation around Z-axis', ...
       'Rotation around Y-axis', ...
       'Rotation around X-axis', ...
```

```
       'Location','southwest')
xlabel('Time (seconds)')
ylabel('Rotation (degrees)')
grid on

figure(3)
plot(timeVector(2:end),vel(:,1), ...
     timeVector(2:end),vel(:,2), ...
     timeVector(2:end),vel(:,3));
title('Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Velocity (m/s)')
grid on

figure(4)
plot(timeVector(2:end),acc(:,1), ...
     timeVector(2:end),acc(:,2), ...
     timeVector(2:end),acc(:,3));
title('Acceleration Over Time')
legend('North','East','Down','Location','southwest')
xlabel('Time (seconds)')
ylabel('Acceleration (m/s^2)')
grid on

figure(5)
plot(timeVector(2:end),angVel(:,1), ...
     timeVector(2:end),angVel(:,2), ...
     timeVector(2:end),angVel(:,3));
title('Angular Velocity Over Time')
legend('North','East','Down')
xlabel('Time (seconds)')
ylabel('Angular Velocity (rad/s)')
grid on
```


Orientation Over Time
Rotation (degrees)
Time (seconds)
200
150
100
50
0
-50
-100
-150
-200
0
1
2
3
4
5
6
Rotation around Z-axis
Rotation around Y-axis
Rotation around X-axis


Velocity Over Time
North
East
Down
Velocity (m/s)
Time (seconds)
15
10
5
0
-5
-10
-15
0
1
2
3
4
5
6

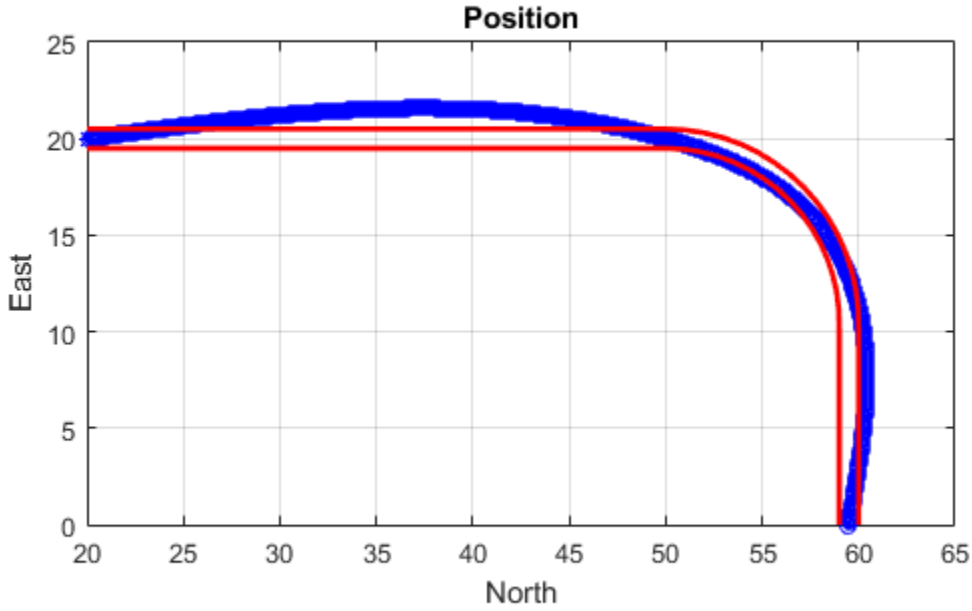

Acceleration Over Time
Acceleration (m/s^2)
Time (seconds)
North
East
Down
2
0
-2
-4
-6
-8
-10
0
1
2
3
4
5
6



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.

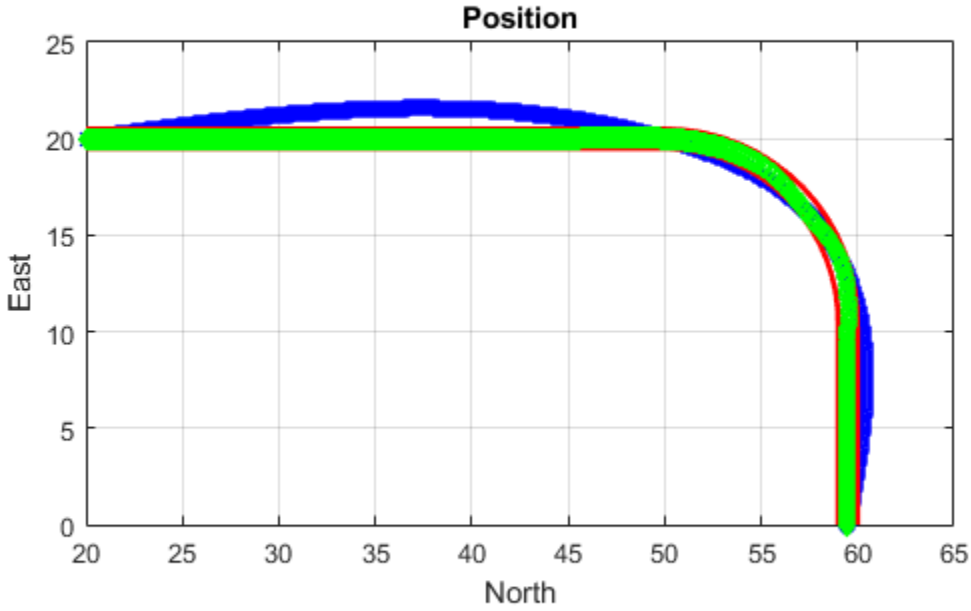
```
figure(1)
xUpperBound = [(20:50)';50+10*sin(0:0.1:pi/2)';60*ones(11,1)];
yUpperBound = [20.5.*ones(31,1);10.5+10*cos(0:0.1:pi/2)';(10:-1:0)'];

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;
```

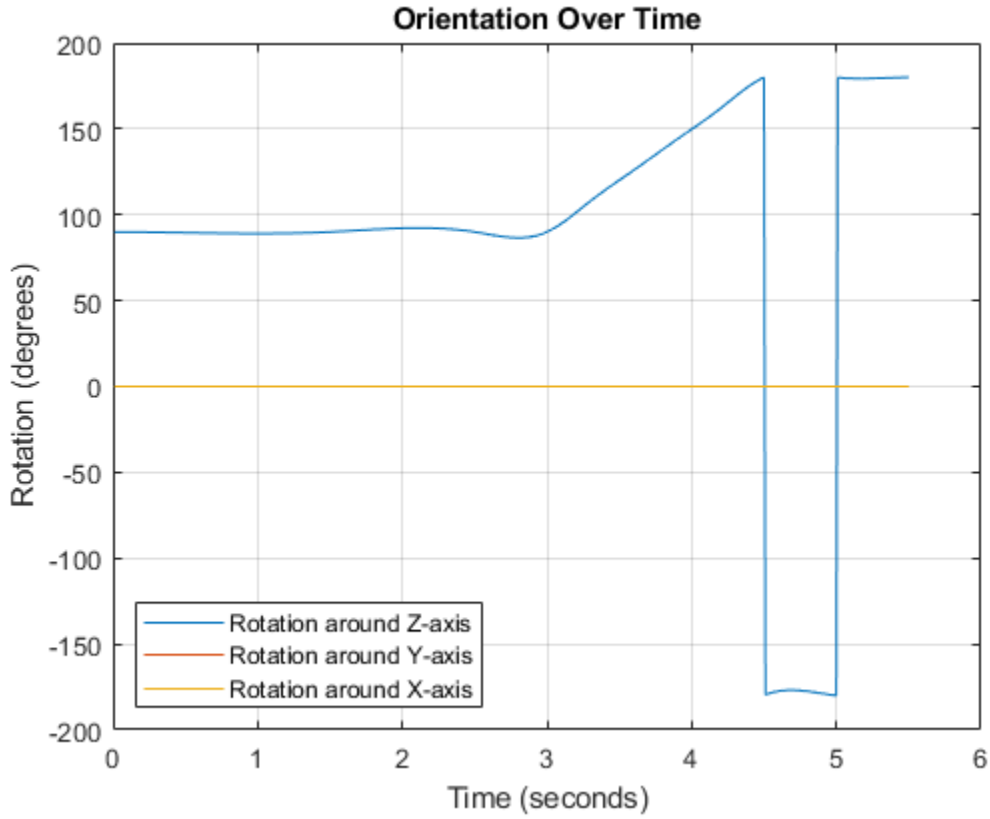
```
                4.5,  59.5,10,0     180,0,0;
                5,    59.5,5,0      180,0,0;
                5.5,  59.5,0,0      180,0,0];

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
```

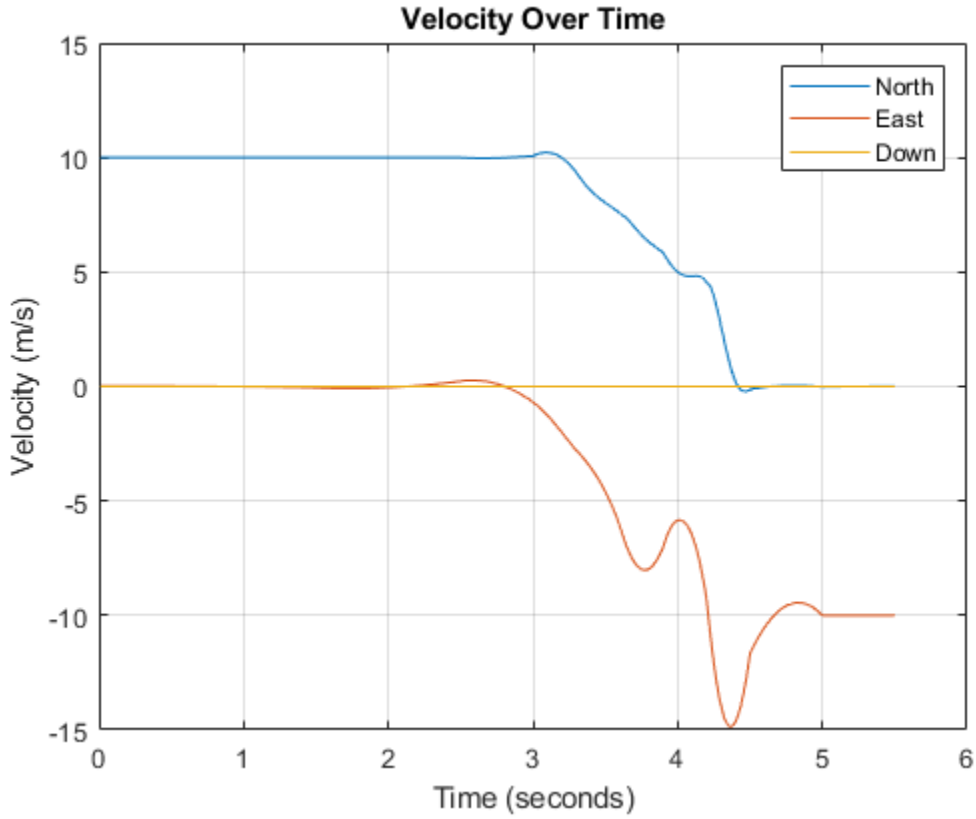


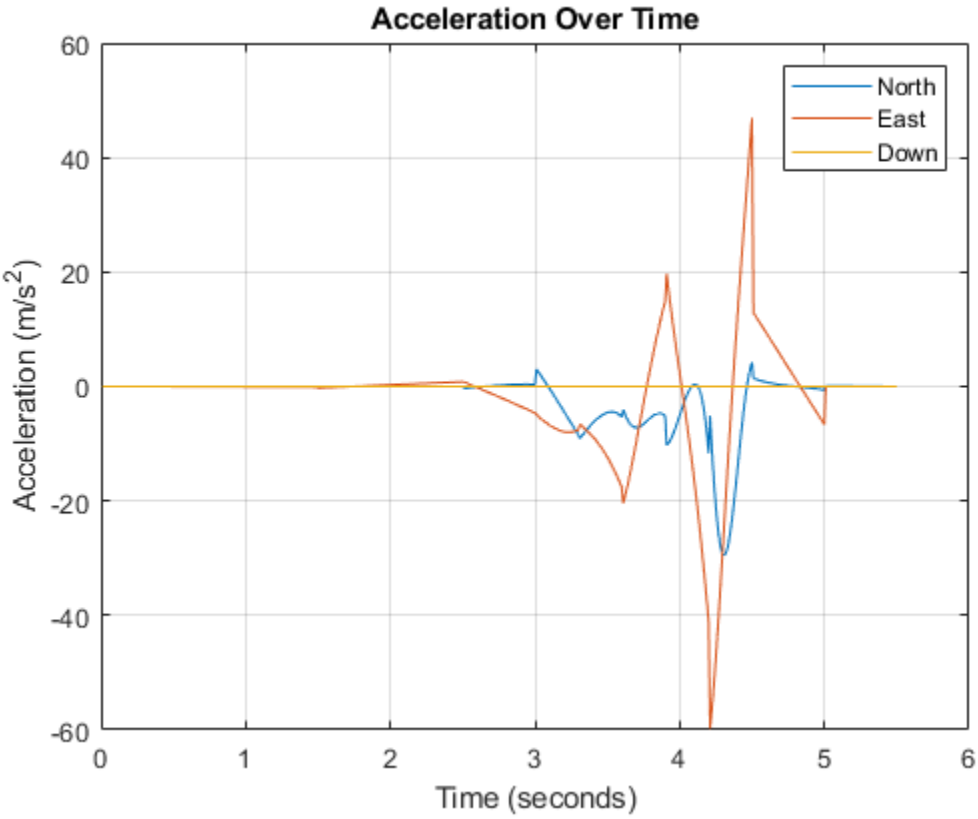
The generated trajectory now fits within the specified boundaries. Visualize the orientation, velocity, acceleration, and angular velocity of the generated trajectory.

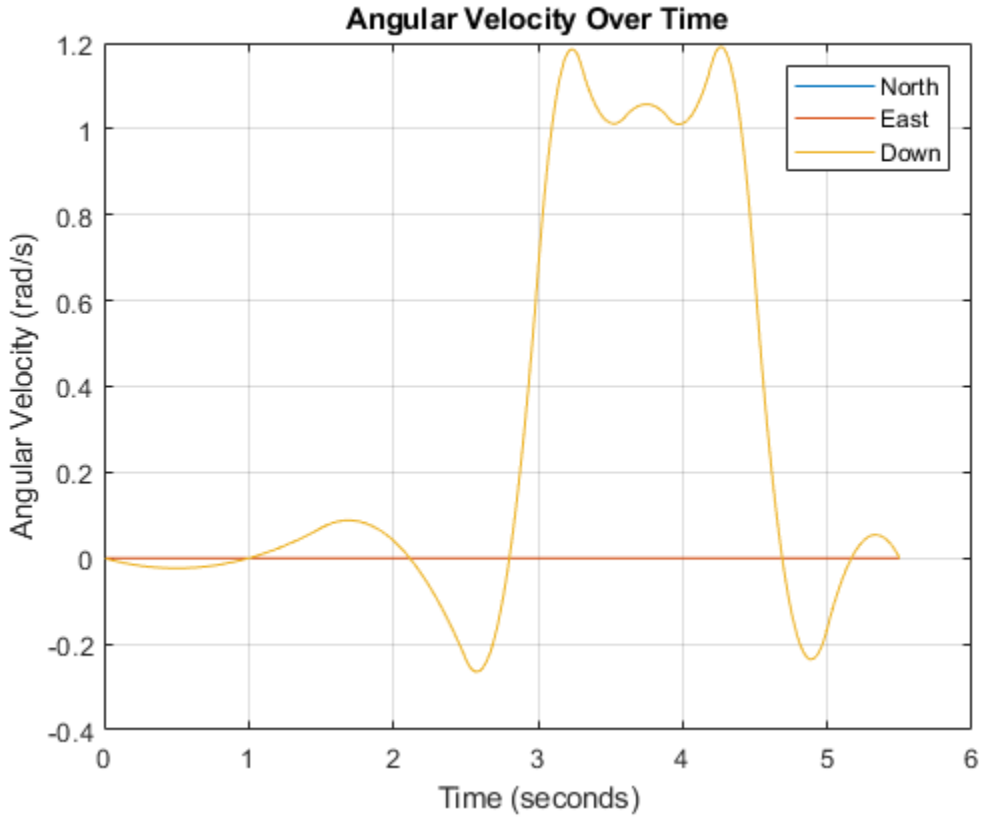
```
figure(2)
timeVector = 0:(1/trajectory.SampleRate):tInfo.TimeOfArrival(end);
eulerAngles = eulerd(orient,'ZYX','frame');
plot(timeVector(2:end),eulerAngles(:,1), ...
     timeVector(2:end),eulerAngles(:,2), ...
     timeVector(2:end),eulerAngles(:,3));
title('Orientation Over Time')
legend('Rotation 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
```


Orientation Over Time
Rotation (degrees)
Time (seconds)
200
150
100
50
0
-50
-100
-150
-200
0
1
2
3
4
5
6
Rotation around Z-axis
Rotation around Y-axis
Rotation around X-axis


Velocity Over Time
North
East
Down
Velocity (m/s)
Time (seconds)
15
10
5
0
-5
-10
-15
0
1
2
3
4
5
6


Acceleration Over Time
North
East
Down
Acceleration (m/s^2)
Time (seconds)
60
40
20
0
-20
-40
-60
0
1
2
3
4
5
6



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

`kinematicTrajectory` | `trackingScenario`

Introduced in R2018b

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

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? (MATLAB).

Creation

Syntax

```
sensor = monostaticRadarSensor
sensor = monostaticRadarSensor(Name,Value)

sensor = monostaticRadarSensor('No scanning')
sensor = monostaticRadarSensor('Raster')
sensor = monostaticRadarSensor('Rotator')
sensor = monostaticRadarSensor('Sector')
sensor = monostaticRadarSensor(SensorIndex, ___ )
```

Description

`sensor = monostaticRadarSensor` creates a radar detection generator object with default property values.

`sensor = monostaticRadarSensor(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes. For example, `monostaticRadarSensor('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.

`sensor = monostaticRadarSensor('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('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 3-311 for the properties set by this syntax.

`sensor = monostaticRadarSensor('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 3-311 for the properties set by this syntax.

`sensor = monostaticRadarSensor('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 3-311 for the properties set by this syntax.

`sensor = monostaticRadarSensor(SensorIndex, ___ )` specifies the sensor ID, `SensorIndex`, as an input argument instead of using the `SensorIndex` name-value pair.

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 (MATLAB).

`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. This property must be defined before you can use the object.

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
`[1;5]` | 2-by-1 vector of positive real values

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 radar are not detected. Units are in degrees.

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 for targets at 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,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 `[0,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 `[-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`.

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 `MechanicalScanLimits` 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 `ElectronicScanLimits` 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 `MountingAngles` 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° 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 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'`	`MechnicalAngle`
`'Electronic'`	`ElectronicAngle`
`'Mechanical and Electronic'`	`MechnicalAngle` + `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.

`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

$$SNR = \text{RadarLoopGain} + RCS - 40\log10(R) \quad (3\text{-}1)$$

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 dBmi.

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.
`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]`.

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 which has these fields:

Field	Definition
`Position`	Position of the GPS receiver in the local NED coordinate system specified as a real-valued 1-by-3 vector. Units are in meters.
`Velocity`	Velocity of the GPS receiver in the local NED coordinate system specified as a real-valued 1-by-3 vector. Units are in meters per second.
`Orientation`	Orientation of the INS with respect to the local NED coordinate system specified as a scalar quaternion or a 3-by-3 real-valued orthonormal frame rotation matrix. Defines the frame rotation from the local NED coordinate system to the current INS body coordinate system. 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 3-308.

Property	Definition
`Time`	Measurement time
`Measurement`	Object measurements
`MeasurementNoise`	Measurement noise covariance matrix
`SensorIndex`	Unique ID of the sensor
`ObjectClassID`	Object classification
`MeasurementParameters`	Parameters used by initialization functions of nonlinear Kalman tracking filters
`ObjectAttributes`	Additional information passed to tracker

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
`SensorIndex`	Unique sensor index
`IsValidTime`	Valid detection time, returned as `0` or `1`. `IsValidTime` is `0` when detection updates are requested at times that are between update intervals specified by `UpdateInterval`.
`IsScanDone`	`IsScanDone` is `true` when the sensor has completed a scan.
`FieldOfView`	Field of view of sensor determines which objects fall within the sensor beam during object execution. The field of view is defined as a 2-by-1 vector of positive real values, [azfov;elfov].
`MeasurementParameters`	`MeasurementParameters` is an array of structures containing the coordinate frame transforms needed to transform positions and velocities in the top-level frame to the current 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)
```

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

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],'Velocit
aircraft1 = platform(sc,'ClassID',2,'Trajectory',aircraft1motion);
aircraft2motion = kinematicTrajectory('SampleRate',1,'Position',[5000 5000 200],'Veloc
aircraft2 = platform(sc,'ClassID',2,'Trajectory',aircraft2motion);
```

```
updaterate =

    15
```

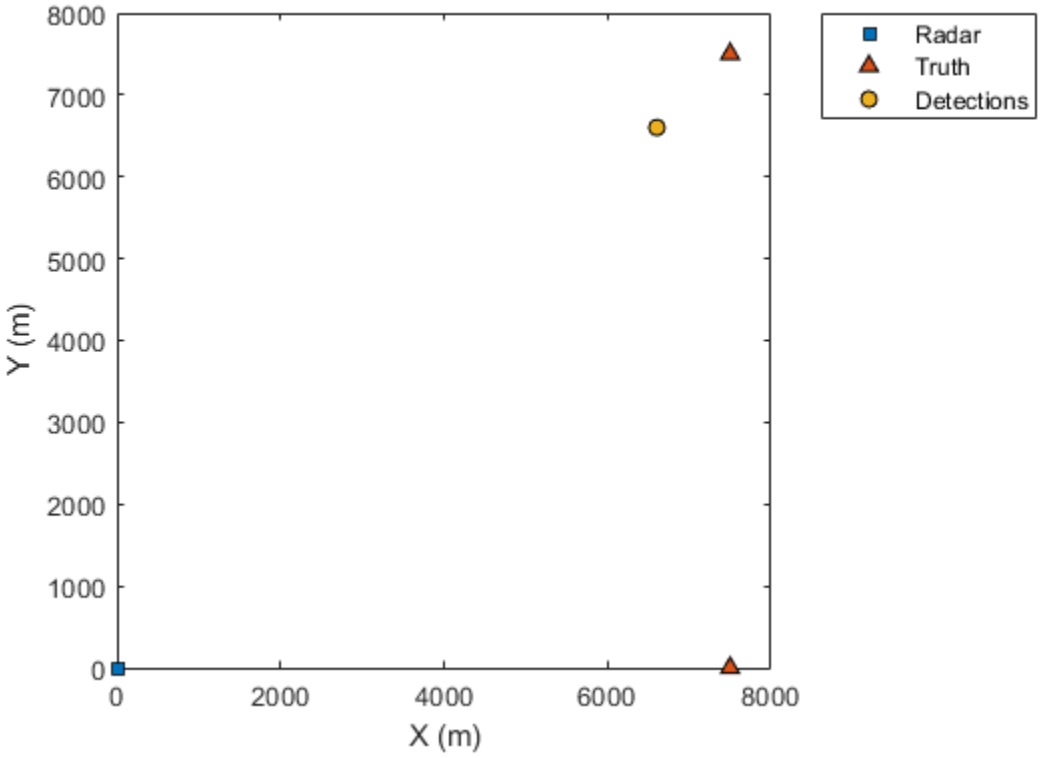
Perform 5 scans.

```
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
```



Definitions

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

<table>
<tr><th>DetectionCoordinates</th><th>Measurement and Measurement Noise Coordinates</th></tr>
<tr><td>'Scenario'
'Body'
'Sensor rectangular'</td><td>Coordinate Dependence on HasRangeRate<table><tr><th>HasRangeRate</th><th>Coordinates</th></tr><tr><td>true</td><td>[x; y; z; vx; vy; vz]</td></tr><tr><td>false</td><td>[x; y; z]</td></tr></table></td></tr>
<tr><td>'Sensor spherical'</td><td>Coordinate Dependence on HasRangeRate and HasElevation<table><tr><th>HasRangeRate</th><th>HasElevation</th><th>Coordinates</th></tr><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></table></td></tr>
</table>

Measurement Parameters

The `MeasurementParameters` field 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”). The longest possible sequence of

transformations is Sensor → Platform → Scenario. For example, if the detections are reported in sensor spherical coordinates and `HasINS` is set to `false`, then the sequence consists of one transformation from sensor to platform. If `HasINS` is `true`, the sequence of transformations consists of two transformations – first to platform coordinates then to scenario coordinates. Trivially, if the detections are reported in platform rectangular coordinates and `HasINS` is set to `false`, the transformation consists only of the identity.

The structure fields 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 `'rectangular'`. When detections are reported in spherical coordinates, `Frame` is set `'spherical'` for the first `struct`.
`OriginPosition`	Position offset of the origin of the frame relative to the parent frame, represented as a 3-by-1 vector.
`OriginVelocity`	Velocity offset of the origin of the frame relative to the parent frame, represented as a 3-by-1 vector.
`Orientation`	3-by-3 real-valued orthonormal frame rotation matrix.
`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 indication if azimuth is included in the measurement.
`HasRange`	A logical scalar indication 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.
`SNR`	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

Classes
`objectDetection` | `radarEmission`

Functions
`targetPoses`

System Objects
`trackerGNN` | `trackerTOMHT`

Introduced in R2018b

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? (MATLAB).

Creation

Syntax

```
assignmentMetics = trackAssignmentMetrics
assignmentMetics = trackAssignmentMetrics(Name,Value)
```

Description

`assignmentMetics = trackAssignmentMetrics` creates a `trackAssignmentMetrics` System object, `assignmentMetics`, with default property values.

`assignmentMetics = trackAssignmentMetrics(Name,Value)` sets properties for the `trackAssignmentMetrics` object using one or more name-value pairs. For example,

`assignmentMetics = 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 (MATLAB).

`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

using tracks reported by any built-in multi-object tracker, and truths reported by the `platformPoses` object function of a `trackingScenario` object.

- `'custom'` – Enable custom properties: `AssignmentDistanceFcn`, `DivergenceDistanceFcn`, `IsInsideCoverageAreaFcn`, `TruthIdentifierFcn`, `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'`

Desired platform motion model, specified as `'constvel'`, `'constacc'`, or `'constturn'`. This property selects the motion model used by the `tracks` input. The motion model governs the outputs when the object is executed.

The motion models expect the `'State'` field to have a column vector as follows:

- `'constvel'` - Position is in elements [1 3 5], velocity in elements [2 4 6].
- `'constacc'` - Position is in elements [1 4 7], velocity in elements [2 5 8], and acceleration in elements [3 6 9].
- `'constturn'` - Position is in elements [1 3 6], velocity in elements [2 4 7], and yaw rate in element 5.

The `'StateCovariance'` field of the `tracks` input must have position, velocity, and turn information in the rows and columns corresponding to the `'State'` field position and velocity input selector.

Dependencies

To enable this property, set the `DistanceFunctionFormat` property to `'built-in'`.

`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 of truth to tracks. Whenever a new track is detected or a track becomes divergent, the track is compared against all truths passed in at each object update. Use this function to find the closest truth 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 at object update. `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 of truths to 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 at object update. onetruth`truth` 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` — Detectable truth object

function handle

Detectable 'truth' object, specified as a function handle. This function determines when 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`. `status` is true when the truth objects, `truths`, are within the coverage area. `truths` is the same truths array input argument at object update. `status` must have the same size as `truths`.

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 track input at object update, specified as a function handle. The track identifiers are unique string 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. `tracks` is the same tracks array input argument at object update.

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 truth input at object update, specified as a function handle. The truth identifiers are unique string 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. `truths` is the same truth array input argument at object update.

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] = assignmentMetics(tracks,truths)
```

Description

`[tracksummary,truthsummary] = assignmentMetics(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
structure | array of structures

Track information, specified as a structure or array of structures. For built-in trackers such as `trackerGNN` or `trackerTOMHT`, the structure contains `'State'`, `'StateCovariance'`, and `'TrackID'` information.

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 `DivergenceDistanceFcn` 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)
```

```
ans =
```

```
  4x15 table

    TrackID    AssignedTruthID    Surviving    TotalLength    DeletionStatus    Deletio
    _______    _______________    _________    ___________    ______________    _______

       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.

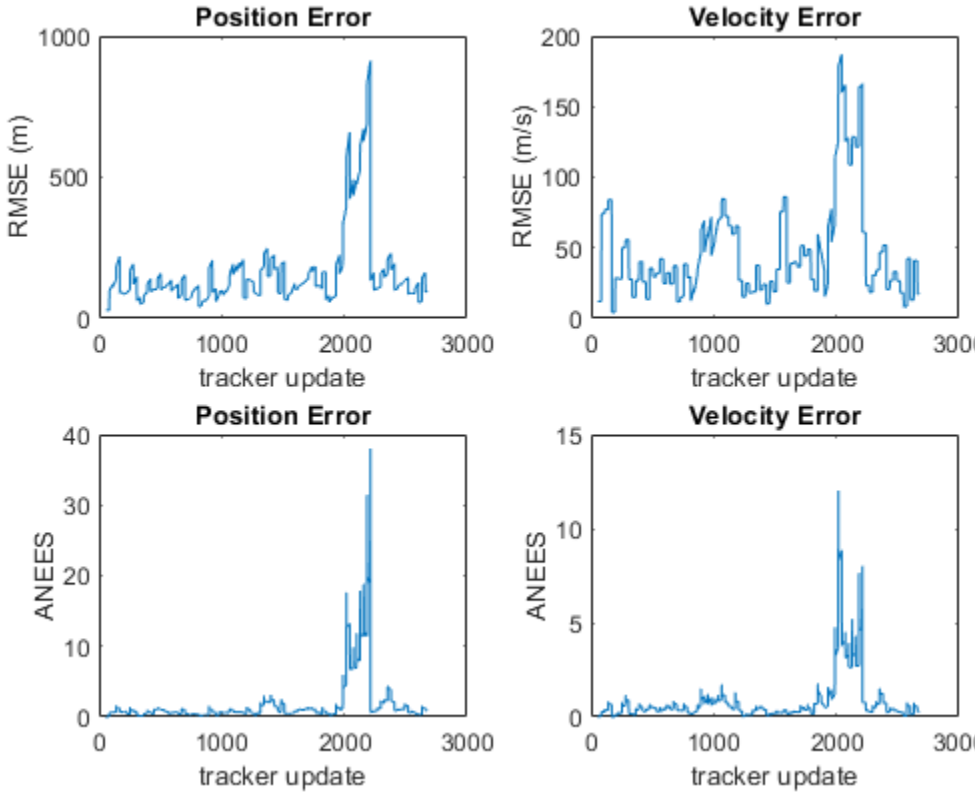
```
truthMetricsTable(tam)
```

```
ans =

  2x10 table

    TruthID    AssociatedTrackID    DeletionStatus    TotalLength    BreakStatus    Bre
    _______    _________________    ______________    ___________    ___________    ___

       2               8                false            2678           false
       3               6                false            2678           false
```

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 =
```

```
  2x5 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 =

  2x5 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
```

See Also

System Objects
`monostaticRadarSensor` | `trackErrorMetrics` | `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)
```

```
ans =

  4x15 table

    TrackID    AssignedTruthID    Surviving    TotalLength    DeletionStatus    Deletio
    _______    _______________    _________    ___________    ______________    _______

       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.

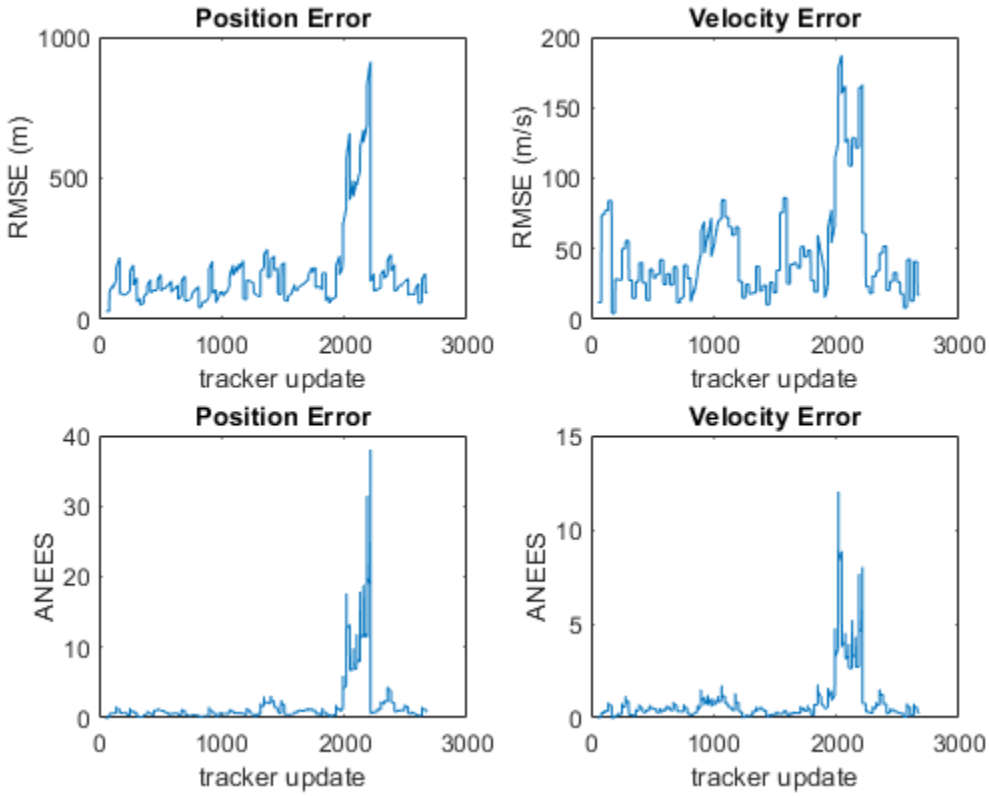
```
truthMetricsTable(tam)
```

```
ans =

  2x10 table
```

```
    TruthID    AssociatedTrackID    DeletionStatus    TotalLength    BreakStatus    Bre
    _______    _________________    ______________    ___________    ___________    ___

       2              8                false            2678           false
       3              6                false            2678           false
```

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 =

  2x5 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 =

  2x5 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 =

  4x15 table

    TrackID    AssignedTruthID    Surviving    TotalLength    DeletionStatus    Deletio
    _______    _______________    _________    ___________    ______________    _______

       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.

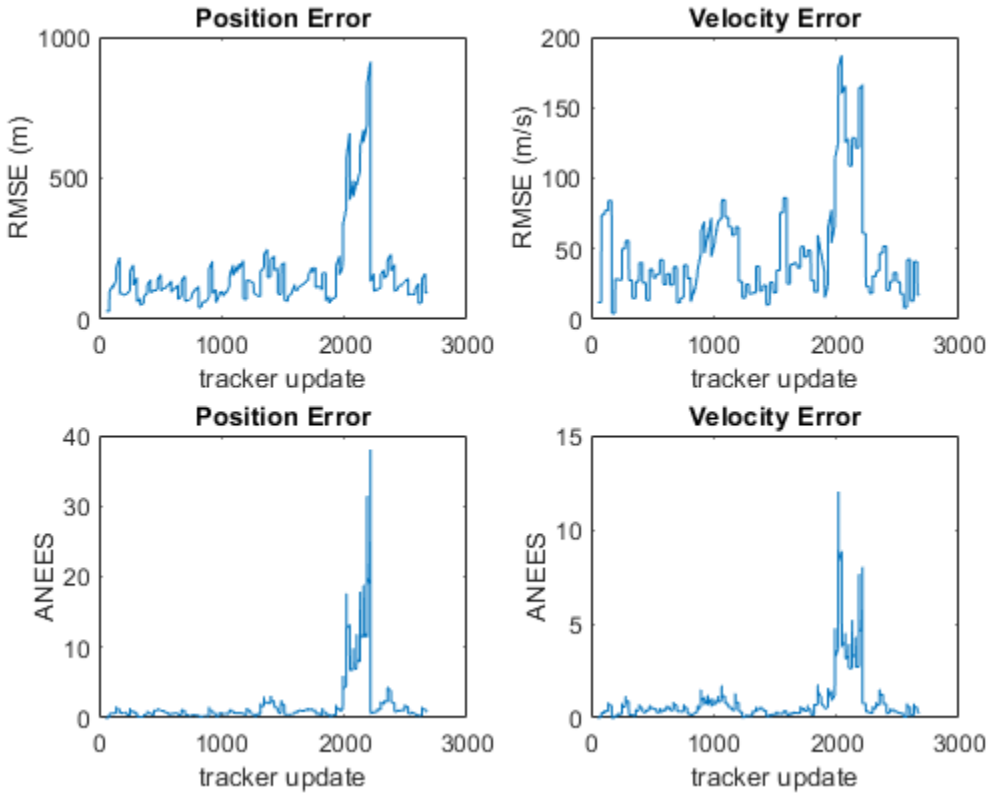
```
truthMetricsTable(tam)
```

```
ans =

  2x10 table

    TruthID    AssociatedTrackID    DeletionStatus    TotalLength    BreakStatus    Bre
    _______    _________________    ______________    ___________    ___________    ___
```

```
     2              8              false          2678           false
     3              6              false          2678           false
```

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 =

  2x5 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 =

  2x5 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

`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 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    Deletio
    _______    _______________    _________    ___________    ______________    _______

       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.

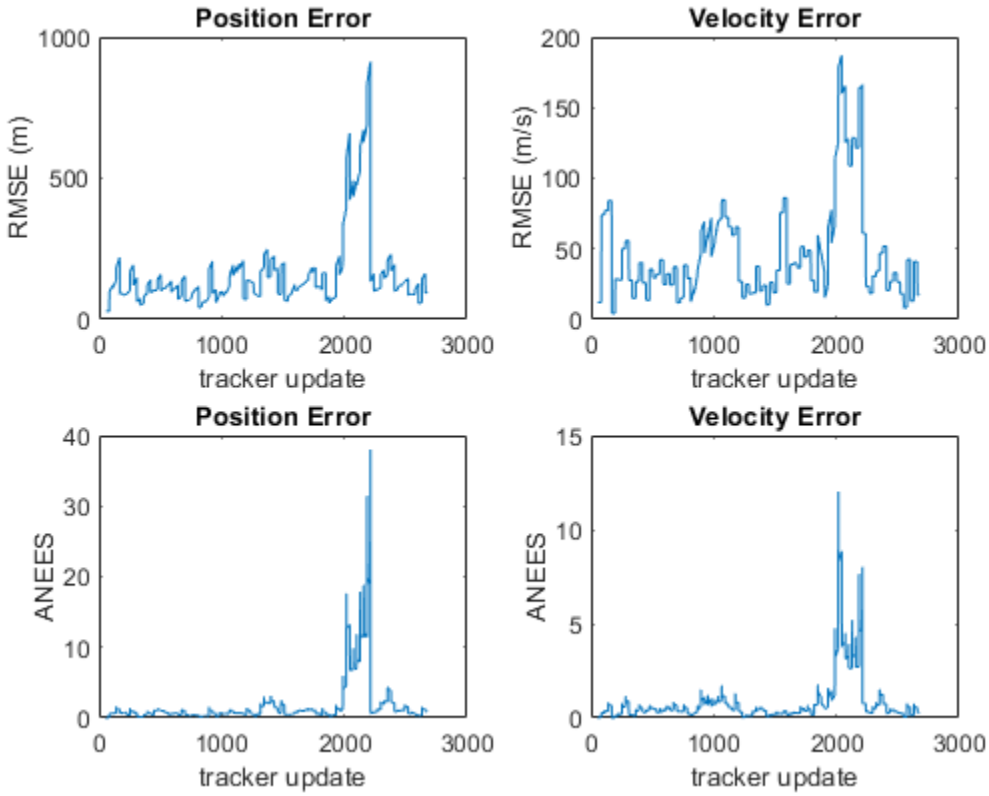
```
truthMetricsTable(tam)
```

```
ans =

  2x10 table

    TruthID    AssociatedTrackID    DeletionStatus    TotalLength    BreakStatus    Bre
    _______    _________________    ______________    ___________    ___________    ___

       2               8                false            2678           false
```

```
     3              6              false          2678           false
```

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 =

  2x5 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 =

  2x5 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

`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 compares tracks from a multi-object tracking system against known truth by automatic assignment of tracks to known truth at each track update.

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? (MATLAB).

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 (MATLAB).

`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.

`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 = trackIDentifier(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 = truthIDentifier(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. `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.

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

- `accRMS` – Acceleration root mean squared error
- `accANEES` – acceleration average NEES

for constant acceleration motion.

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.

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.

Input Arguments

`tracks` — Track information

structure | array of structures

Track information, specified as a structure or array of structures. For built-in trackers such as `trackerGNN` or `trackerTOMHT`, the structure 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

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`

`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, 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, 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, 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, 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, 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, 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, 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, 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 `EstmationErrorFcn` 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    Deletio
    _______    _______________    _________    ___________    ______________    _______

       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.

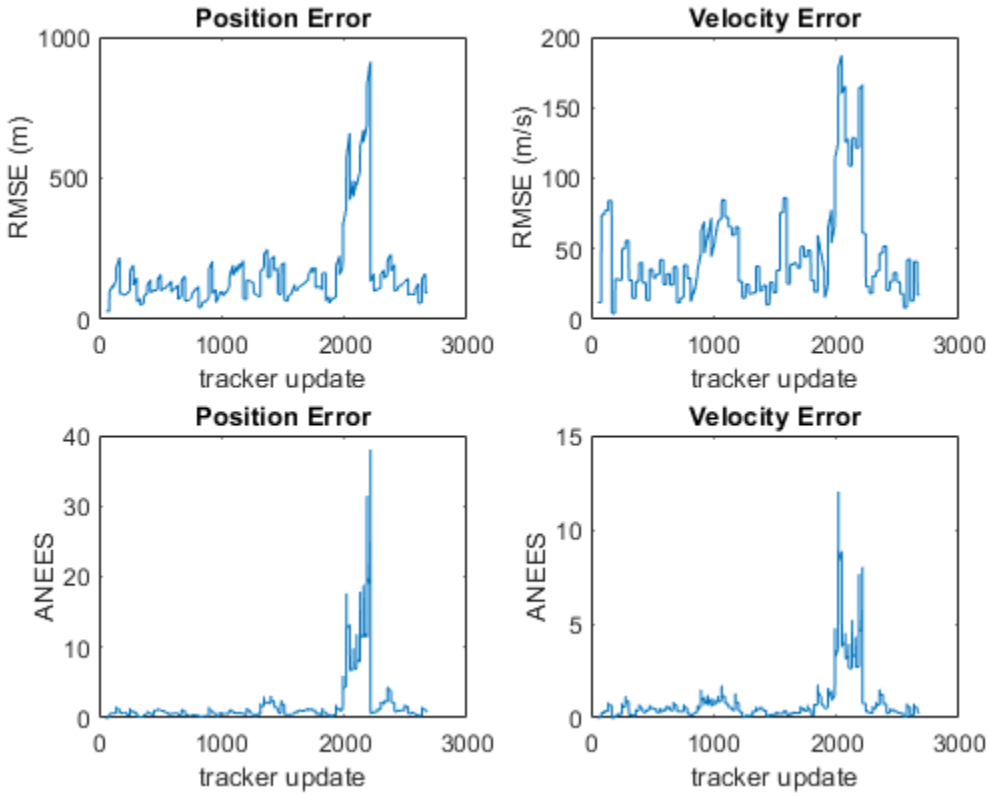
```
truthMetricsTable(tam)
```

```
ans =

  2x10 table
```

```
    TruthID    AssociatedTrackID    DeletionStatus    TotalLength    BreakStatus    Bre
    _______    _________________    ______________    ___________    ___________    ___

       2              8                false            2678           false
       3              6                false            2678           false
```

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 =

  2x5 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 =

  2x5 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
```

See Also

System Objects
`monostaticRadarSensor` | `trackAssignmentMetrics` | `trackerGNN` | `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 =

  4x15 table

    TrackID    AssignedTruthID    Surviving    TotalLength    DeletionStatus    Deletio
    _______    _______________    _________    ___________    ______________    _______

       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.

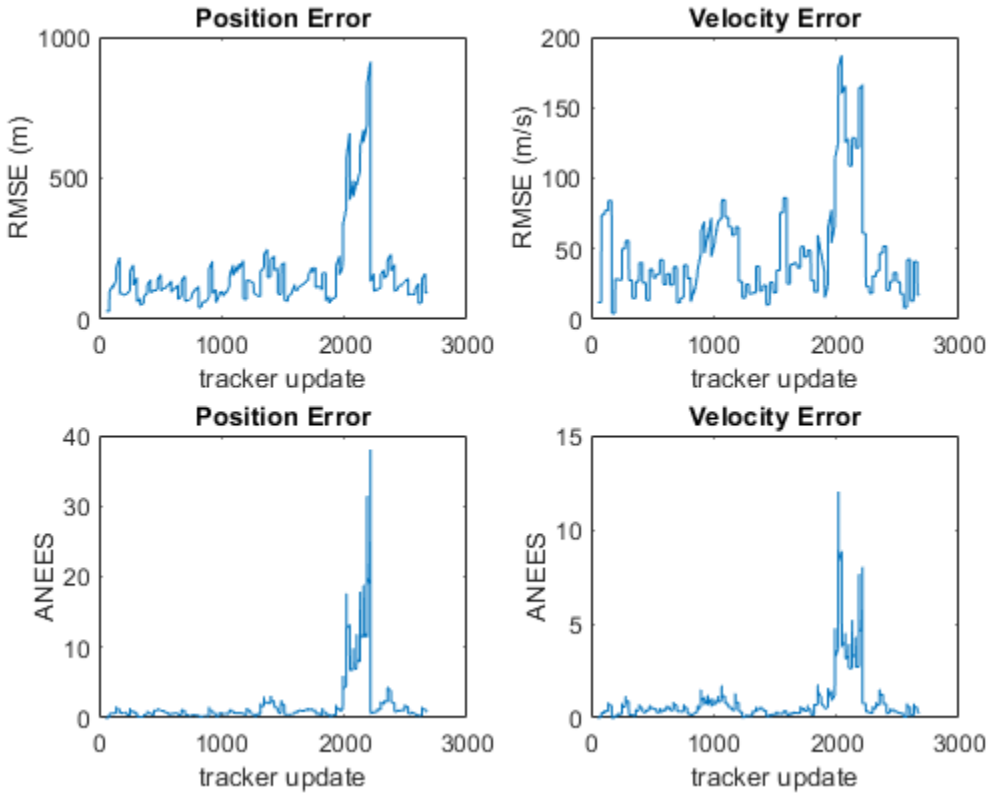
```
truthMetricsTable(tam)
```

```
ans =

  2x10 table

    TruthID    AssociatedTrackID    DeletionStatus    TotalLength    BreakStatus    Bre
    _______    _________________    ______________    ___________    ___________    ___
```

```
     2              8              false          2678           false
     3              6              false          2678           false
```

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 =

  2x5 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 =

  2x5 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 denotes root mean squared error and ANEES denotes average normalized estimation error.
- 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 =

  4x15 table

    TrackID    AssignedTruthID    Surviving    TotalLength    DeletionStatus    Deletio
    _______    _______________    _________    ___________    ______________    _______

       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.

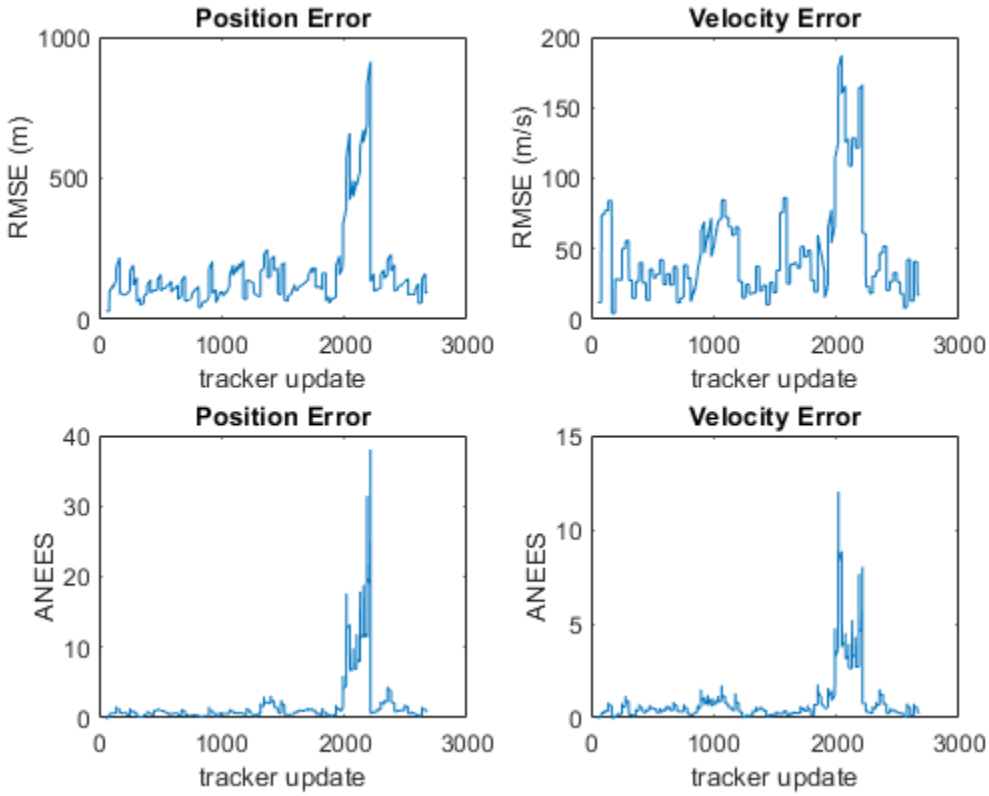
```
truthMetricsTable(tam)
```

```
ans =

  2x10 table

    TruthID    AssociatedTrackID    DeletionStatus    TotalLength    BreakStatus    Bre
    _______    _________________    ______________    ___________    ___________    ___
```

```
     2              8              false          2678           false
     3              6              false          2678           false
```

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 =

  2x5 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 =

  2x5 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 denotes root mean squared error and ANEES denotes average normalized estimation error.
- 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 =

  4x15 table

    TrackID    AssignedTruthID    Surviving    TotalLength    DeletionStatus    Deletio
    _______    _______________    _________    ___________    ______________    _______

       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.

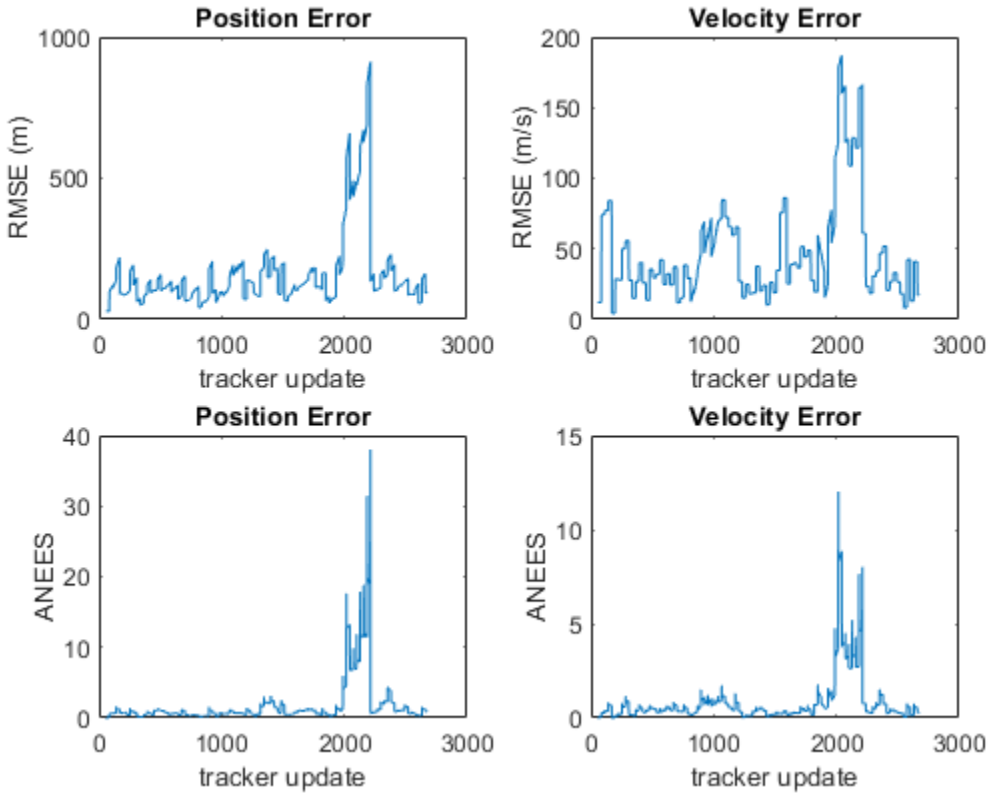
```
truthMetricsTable(tam)
```

```
ans =

  2x10 table

    TruthID    AssociatedTrackID    DeletionStatus    TotalLength    BreakStatus    Bre
    _______    _________________    ______________    ___________    ___________    ___

       2               8                false            2678           false
```

```
     3              6              false          2678           false
```

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 =

  2x5 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 =

  2x5 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 denotes root mean squared error and ANEES denotes average normalized estimation error.
- 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 =

  4x15 table

    TrackID    AssignedTruthID    Surviving    TotalLength    DeletionStatus    Deletio
    _______    _______________    _________    ___________    ______________    _______

       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.

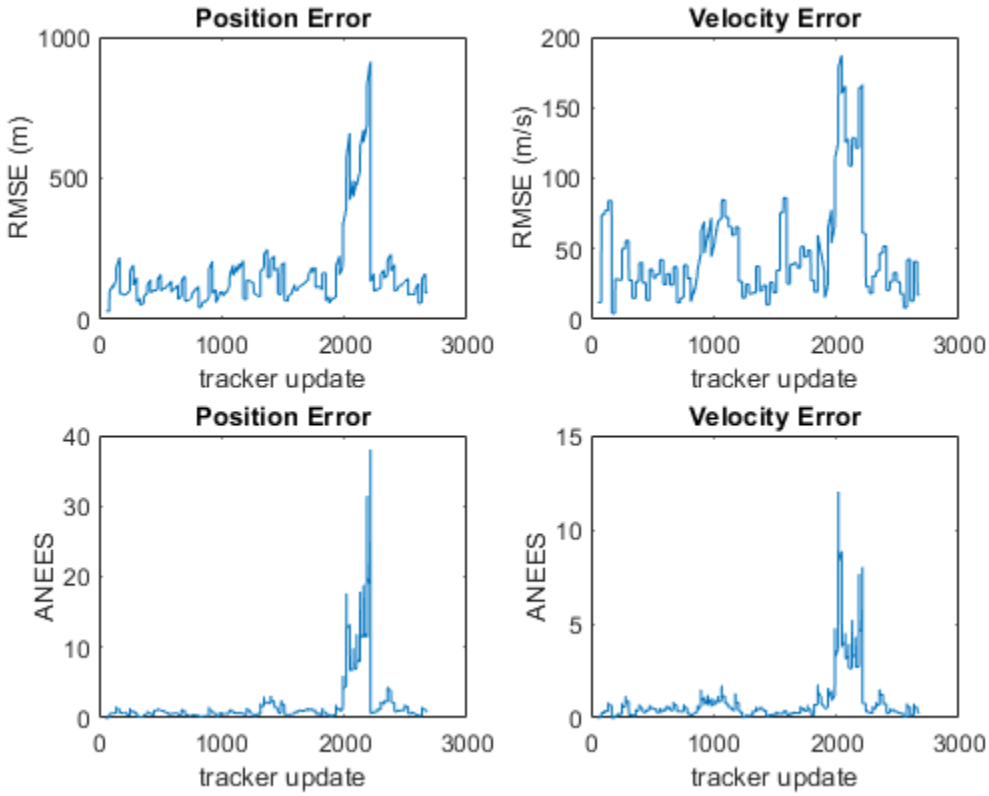
```
truthMetricsTable(tam)
```

```
ans =

  2x10 table

    TruthID    AssociatedTrackID    DeletionStatus    TotalLength    BreakStatus    Bre
    _______    _________________    ______________    ___________    ___________    ___

       2               8                false            2678           false
```

```
     3              6              false          2678           false
```

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 =

  2x5 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 =

  2x5 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 denotes root mean squared error and ANEES denotes average normalized estimation error.
- 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

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 3-406.

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? (MATLAB).

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 (MATLAB).

`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`.

Initialization Function	Function Definition
`initcvabf`	Initialize constant-velocity alpha-beta filter
`initcaabf`	Initialize constant-acceleration alpha-beta filter

Initialization Function	Function Definition
`initcvekf`	Initialize constant-velocity extended Kalman filter.
`initcackf`	Initialize constant-acceleration cubature filter.
`initctckf`	Initialize constant-turn-rate cubature filter.
`initcvckf`	Initialize constant-velocity cubature filter.
`initcapf`	Initialize constant-acceleration particle filter.
`initctpf`	Initialize constant-turn-rate particle filter.
`initcvpf`	Initialize constant-velocity particle filter.
`initcvkf`	Initialize constant-velocity linear Kalman filter.
`initcvukf`	Initialize constant-velocity unscented Kalman filter.
`initcaekf`	Initialize constant-acceleration extended Kalman filter.
`initcakf`	Initialize constant-acceleration linear Kalman filter.
`initcaukf`	Initialize constant-acceleration unscented Kalman filter.
`initctekf`	Initialize constant-turn-rate extended Kalman filter.
`initctukf`	Initialize constant-turn-rate unscented Kalman filter.
`initcvmscekf`	Initialize constant-velocity modified spherical coordinates extended Kalman filter.
`initrpekf`	Initialize constant-velocity range-parametrized extended Kalman filter.
`initapekf`	Initialize constant-velocity angle-parametrized extended Kalman filter.
`initekfimm`	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 `objectDetection`. The output of this function must be an object belonging to one of the filter classes: `trackingKF`, `trackingEKF`, `trackingUKF`, `trackingCKF`, `trackingPF`, `trackingMSCEKF`, `trackingGSF`, `trackingIMM`, or `trackingAB`.

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`

`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 an1-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 <= C_2 <= C_3<=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( ___ )
[ ___ ,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`.

`[ ___ ,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.

`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

structure | array of structures

Confirmed tracks, returned as a structure or array of structures. Each structure corresponds to a track. A track is confirmed if its score is greater than the confirmation threshold specified in the `ConfirmationThreshold` property. In that case, the `IsConfirmed` field of the structure is `true`. The fields of the structure are defined “Track Structure” on page 3-405.

Data Types: `struct`

`tentativeTracks` — Tentative tracks

structure | array of structures

Tentative tracks, returned as a structure or array of structures. Each structure corresponds to a track. A track is tentative if its score is less than or equal to the confirmation threshold specified in 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 3-405.

Data Types: `struct`

`allTracks` — All tracks

structure

All tracks, returned as a structure or 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 3-405.

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

`BranchIDsAtStepBeginning`	Branch IDs when update began.
`CostMatrix`	Cost of assignment matrix.
`Assignments`	Assignments returned from `assignTOMHT`.
`UnassignedTracks`	IDs of unassigned branches returned from the tracker
`UnassignedDetections`	IDs of unassigned detections returned from `trackerTOMHT`.
`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. Such branches are incompatible.
`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 `pruneTrackBranches` function determines to be pruned.
`GlobalBranchProbabilities`	Global probability of each branch existing in the global hypotheses.
`BranchesDeletedByPruning`	Branches deleted by the tracker.
`BranchIDsAtStepEnd`	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`

getTrackFilterProperties	Obtain track filter properties
setTrackFilterProperties	Set track filter properties
getBranches	Lists track branches
predictTrackToTime	Predict track state

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 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|s . 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)
velocities = getTrackVelocities(tracks,velocitySelector)
```

```
positions =

   10.0000         0
         0   10.0000


velocities =

     0     0
     0     0
```

Definitions

Track Structure

Track information is returned as an array of structures with the following fields:

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`	The current state of the track logic. • For `'History'` track logic, a 1-by-*Q* logical array, where *Q* is the greater of *N* or *R* from the confirmation and deletion thresholds. • For `'Score'` track logic, a 1-by-2 numerical array in the form: `[currentScore, maxScore]`.
`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.

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.

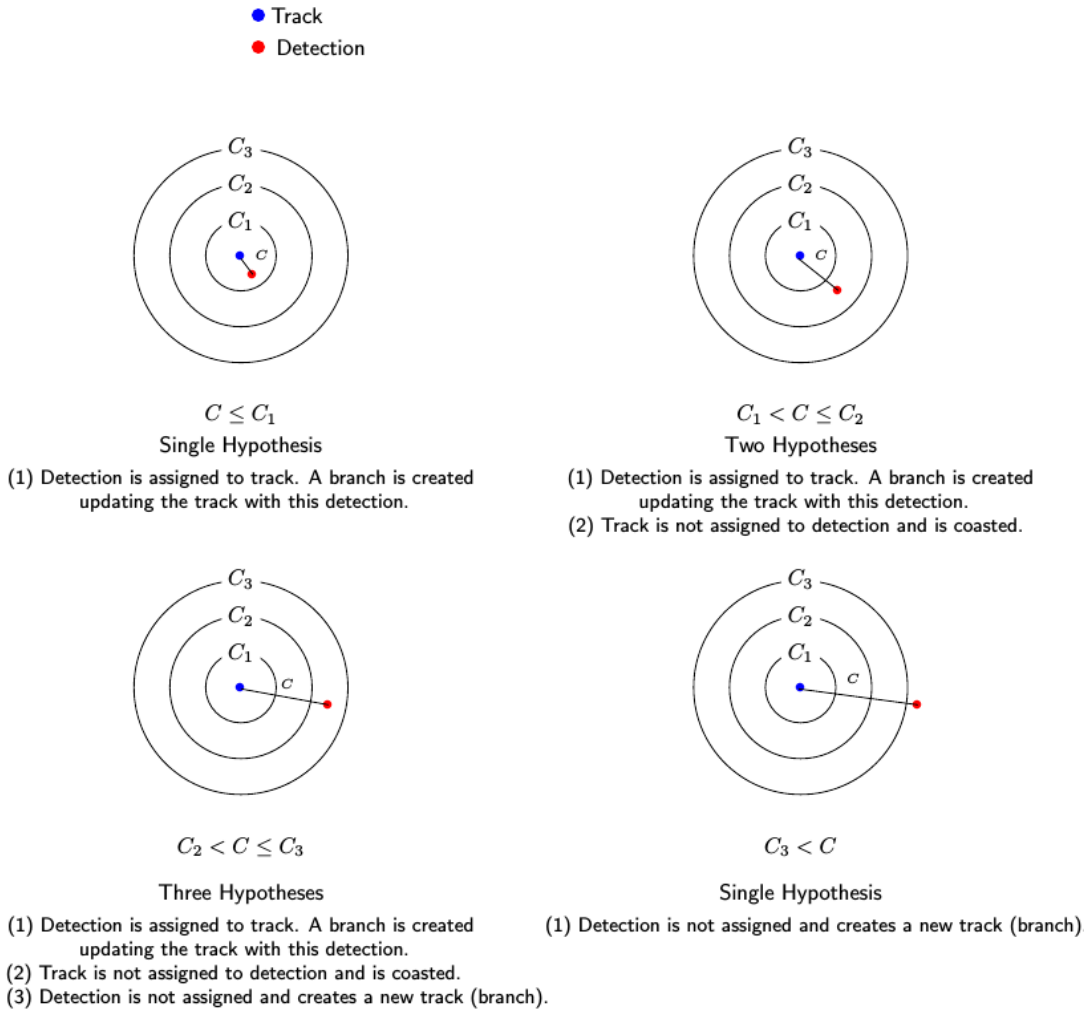
10 Additional pruning is performed based on N-scan history.

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 <= C_2 <= C_3.

If the cost of an assignment is `C = 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.



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`

Classes
`objectDetection` | `trackingAB` | `trackingCKF` | `trackingEKF` | `trackingGSF` | `trackingIMM` | `trackingKF` | `trackingMSCEKF` | `trackingPF` | `trackingUKF`

System Objects
`irSensor` | `monostaticRadarSensor` | `radarSensor` | `sonarSensor` | `trackerGNN`

Introduced in R2018b

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')
disp(values{1})
```

```
values =

  2x1 cell array

    {3x3 double}
    {3x3 double}

     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')
disp(values{1})
```

```
values =

  2x1 cell array

    {3x3 double}
    {3x3 double}

     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     0     0
     0     2     0
     0     0     2
```

Show the process noise.

```
disp(values{2})
```

```
     5     0     0
     0     5     0
     0     0     5
```

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     0     0
     0     2     0
     0     0     2
```

Show the process noise.

```
disp(values{2})
```

```
     5     0     0
     0     5     0
     0     0     5
```

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 =

  struct with fields:

            TrackID: 1
           BranchID: 1
         UpdateTime: 0
                Age: 1
              State: [6x1 double]
    StateCovariance: [6x6 double]
```

```
      TrackLogic: 'Score'
 TrackLogicState: [13.7102 13.7102]
     IsConfirmed: 0
       IsCoasted: 0
   ObjectClassID: 0
ObjectAttributes: {}
```

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`	The current state of the track logic. • For `'History'` track logic, a 1-by-*Q* logical array, where *Q* is the greater of *N* or *R* from the confirmation and deletion thresholds. • For `'Score'` track logic, a 1-by-2 numerical array in the form: `[currentScore, maxScore]`.
`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(tracker,type,id,time)
predictedtracks = predictTracksToTime(tracker,type,category,time)
predictedtracks = predictTracksToTime(tracker,trackid,time)
predictedtracks = predictTracksToTime(tracker,category,time)
predictedtracks = predictTracksToTime( ___ ,'WithCovariance',tf)
```

Description

`predictedtracks = predictTracksToTime(tracker,type,id,time)` returns the predicted tracks or branches, `predictedtracks`, of the tracker, `tracker`, at the specified time, `time`. Specify the type, `type`, of tracked object and the object ID, `id`. The tracker must be updated at least once before calling this object function. Use `isLocked(tracker)` to test whether the tracker has been updated.

This syntax applies when you create the tracker using `trackerTOMHT`.

`predictedtracks = predictTracksToTime(tracker,type,category,time)` returns all predicted tracks or branches for a specified category, `category`, of track objects.

This syntax applies when you create the tracker using `trackerTOMHT`.

`predictedtracks = predictTracksToTime(tracker,trackid,time)` returns the predicted tracks, `predictedtracks`, of the tracker, `tracker`, at the specified time, `time`. Specify the track identifier, `trackid`. The tracker must be updated at least once before calling this object function. Use `isLocked(tracker)` to test whether the tracker has been updated.

This syntax applies when you create the tracker using `trackerGNN`, `trackerJPDA`, or `trackerPHD`.

`predictedtracks = predictTracksToTime(tracker,category,time)` returns all predicted tracks for a specified category, `category`, of track objects.

This syntax applies when you create the tracker using `trackerGNN`, `trackerJPDA`, or `trackerPHD`.

`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 =

  struct with fields:

            TrackID: 1
           BranchID: 1
         UpdateTime: 1
                Age: 1
              State: [6x1 double]
    StateCovariance: [6x6 double]
         TrackLogic: 'Score'
    TrackLogicState: [13.7102 13.7102]
        IsConfirmed: 0
          IsCoasted: 0
      ObjectClassID: 0
```

```
    ObjectAttributes: {}
```

Input Arguments

`tracker` — Target tracker
`trackerTOMHT` object | `trackerGNN` object

Target tracker, specified as a `trackerTOMHT` or `trackerGNN` 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, specified as a positive integer.

Example: `15`

Data Types: `single` | `double`

`time` — Prediction time
scalar

Prediction time, specified as a scalar. Tracks states 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
structures | array of structures

List of tracks or branches, returned as an 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`	The current state of the track logic. • For `'History'` track logic, a 1-by-Q logical array, where Q is the greater of N or R from the confirmation and deletion thresholds. • For `'Score'` track logic, a 1-by-2 numerical array in the form: `[currentScore, maxScore]`.

`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

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 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 tracker considers the track to represent a physical object. If detections are not assign 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 3-406.

To track objects using a 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? (MATLAB).

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 (MATLAB).

`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`.

Initialization Function	Function Definition
`initcvabf`	Initialize constant-velocity alpha-beta filter
`initcaabf`	Initialize constant-acceleration alpha-beta filter

Initialization Function	Function Definition
`initcvekf`	Initialize constant-velocity extended Kalman filter.
`initcackf`	Initialize constant-acceleration cubature filter.
`initctckf`	Initialize constant-turn-rate cubature filter.
`initcvckf`	Initialize constant-velocity cubature filter.
`initcapf`	Initialize constant-acceleration particle filter.
`initctpf`	Initialize constant-turn-rate particle filter.
`initcvpf`	Initialize constant-velocity particle filter.
`initcvkf`	Initialize constant-velocity linear Kalman filter.
`initcvukf`	Initialize constant-velocity unscented Kalman filter.
`initcaekf`	Initialize constant-acceleration extended Kalman filter.
`initcakf`	Initialize constant-acceleration linear Kalman filter.
`initcaukf`	Initialize constant-acceleration unscented Kalman filter.
`initctekf`	Initialize constant-turn-rate extended Kalman filter.
`initctukf`	Initialize constant-turn-rate unscented Kalman filter.
`initcvmscekf`	Initialize constant-velocity modified spherical coordinates extended Kalman filter.
`initrpekf`	Initialize constant-velocity range-parametrized extended Kalman filter.
`initapekf`	Initialize constant-velocity angle-parametrized extended Kalman filter.
`initekfimm`	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 `objectDetection`. The output of this function must be an object belonging to one of the filter classes: `trackingKF`, `trackingEKF`, `trackingUKF`, `trackingCKF`, `trackingPF`, `trackingMSCEKF`, `trackingGSF`, `trackingIMM`, or `trackingAB`.

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

`'Munkres'` (default) | `'Jonker-Volgenant'` | `'Auction'` | `'Custom'`

Assignment algorithm, specified as `'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` method of each tracking filter (e.g., `trackingCKF` and `trackingEKF`) for 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]`. A track is deleted if, in the last `R` updates, it was assigned less than `P` detections.
- 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`

`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( ___ )
[ ___ ,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`.

`[ ___ ,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.

`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 branch 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

structure | array of structures

Confirmed tracks, returned as a structure or array of structures. Each structure corresponds to a track. A track is confirmed if its score is greater than the confirmation threshold specified in the `ConfirmationThreshold` property. In that case, the `IsConfirmed` field of the structure is `true`. The fields of the structure are defined in “Track Structure” on page 3-439.

Data Types: `struct`

`tentativeTracks` — Tentative tracks

structure | array of structures

Tentative tracks, returned as a structure or array of structures. Each structure corresponds to a track. A track is tentative if its score is less than or equal to the confirmation threshold specified in 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 3-439.

Data Types: `struct`

`allTracks` — All tracks

structure

All tracks, returned as a structure or 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 3-439.

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

`TrackIDsAtStepBeginning`	Track IDs when step began
`CostMatrix`	Cost of assignment matrix
`Assignments`	Assignments returned from `assignTOMHT`
`UnassignedTracks`	IDs of unassigned tracks returned from the tracker
`UnassignedDetections`	IDs of unassigned detections returned from `trackerGNN`
`InitiatedTrackIDs`	IDs of tracks initiated during the step
`DeletedTrackIDs`	IDs of tracks deleted during the step
`TrackIDsAtStepEnd`	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

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

```
positions =

    10     0
     0    10


velocities =

     0     0
```

```
     0     0
```

Definitions

Track Structure

Track information is returned as an array of structures having the following fields:

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`	The current state of the track logic. • For `'History'` track logic, a 1-by-Q logical array, where Q is the greater of N or R from the confirmation and deletion thresholds. • For `'Score'` track logic, a 1-by-2 numerical array in the form: `[currentScore, maxScore]`.
`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.

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

Classes
objectDetection | trackHistoryLogic | trackScoreLogic | trackingABF | trackingCKF | trackingEKF | trackingGSF | trackingIMM | trackingKF | trackingMSCEKF | trackingPF | trackingUKF

System Objects
monostaticRadarSensor | staticDetectionFuser | trackBranchHistory | trackerTOMHT

Introduced in R2018b

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 3-456.

By default, the `trackerPHD` can track extended objects using the `ggiwphd` filter, which models detections from an extended object as a parse points cloud. 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? (MATLAB).

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 (MATLAB).

`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 ΔT 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`

`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.

`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 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 3-455.

Data Types: `struct`

`tentativeTracks` — Tentative tracks

structure | array of structures

Tentative tracks, returned as a structure or 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 3-455.

Data Types: `struct`

`allTracks` — All tracks

structure | array of structures

All tracks, returned as a structure or 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 3-455.

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

`CorrectionOrder`	The order in which sensors are used for state estimate correction, returned as a row vector of `SensorIndex`. For example, [1 3 2 4].
`TrackIDsAtStepBeginning`	Track IDs when step began.
`DeletedTrackIDs`	IDs of tracks deleted during the step.
`TrackIDsAtStepEnd`	Track IDs when the step ended.
`SensorAnalysisInfo`	Cell array of sensor analysis information.

The `SensorAnalysisInfo` field can include multiple sensor information reports. Each report is a structure containing:

Field	**Description**
`SensorIndex`	Sensor index.
`DetectionCells`	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.
`DetectionLikelihoods`	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.
`IsBirthCells`	Indicates if the detection cells listed in `DetectionCells` give birth to new tracks, returned as a 1-by-*P* logical vector, where *P* is the number of detection cells.
`NumPartitions`	Number of partitions.
`DetectionProbability`	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.

`LabelsBeforeCorrection`	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 `TrackID`. For example, [1 1 2 0 0]. Note that multiple components can share the same `TrackID`.
`LabelsAfterCorrection`	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 `TrackID`. For example, [1 1 1 2 2 0 0]. Note that multiple components can share the same `TrackID`.
`WeightsBeforeCorrection`	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 `LabelsBeforeCorrection`. For example, [0.1 0.5 0.7 0.3 0.2].
`WeightsAfterCorrection`	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 `LabelsAfterCorrection`. 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`

predictTracksToTime	Predict track state

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 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
```

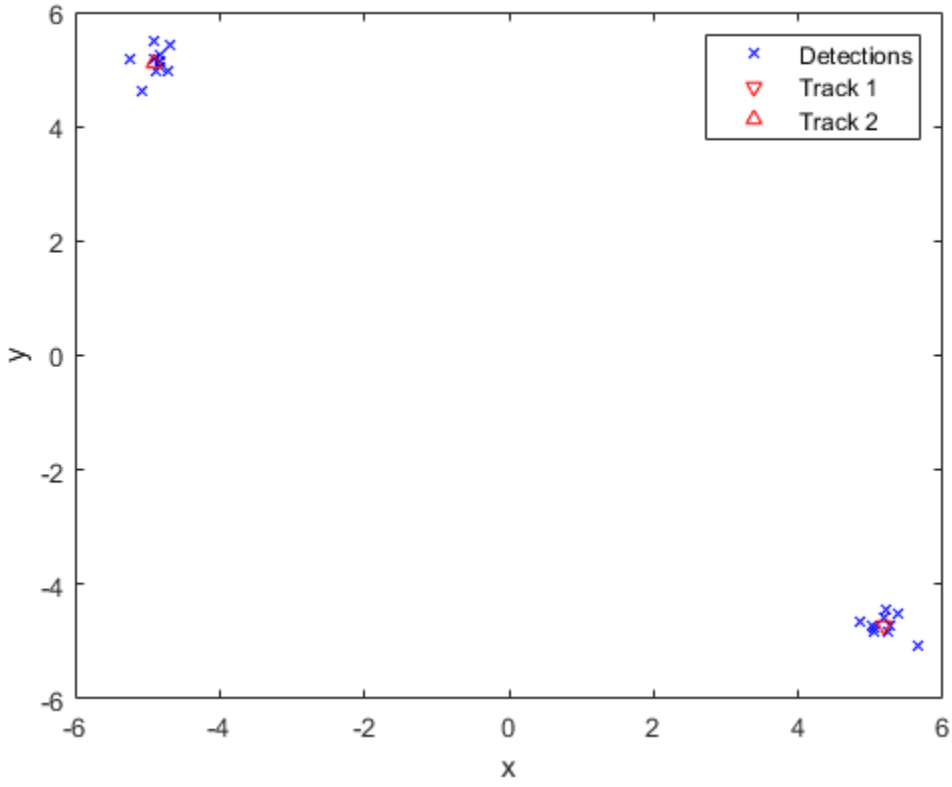
```
    detections{i} = objectDetection(0,[5;-5;0] + 0.2*randn(3,1));
end
for j = 11:20
    detections{j} = objectDetection(0,[-5;5;0] + 0.2*randn(3,1));
end

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, initcvggiwphd, 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 1 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')
```



Definitions

Track Structure

Track information is returned as an array of structures having the following fields:

Field	Description
`TrackID`	Unique integer that identifies the track.

`IsConfirmed`	True if the track is assumed to be of a real target.
`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.
`MeasurementRate`	Expected number of detections from the tracked object.
`IsCoasted`	`trackerPHD` does not support the `IsCoasted` field. The value is always 0.
`ObjectClassID`	`trackerPHD` does not support the `ObjectClassID` field. The value is always 0.

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

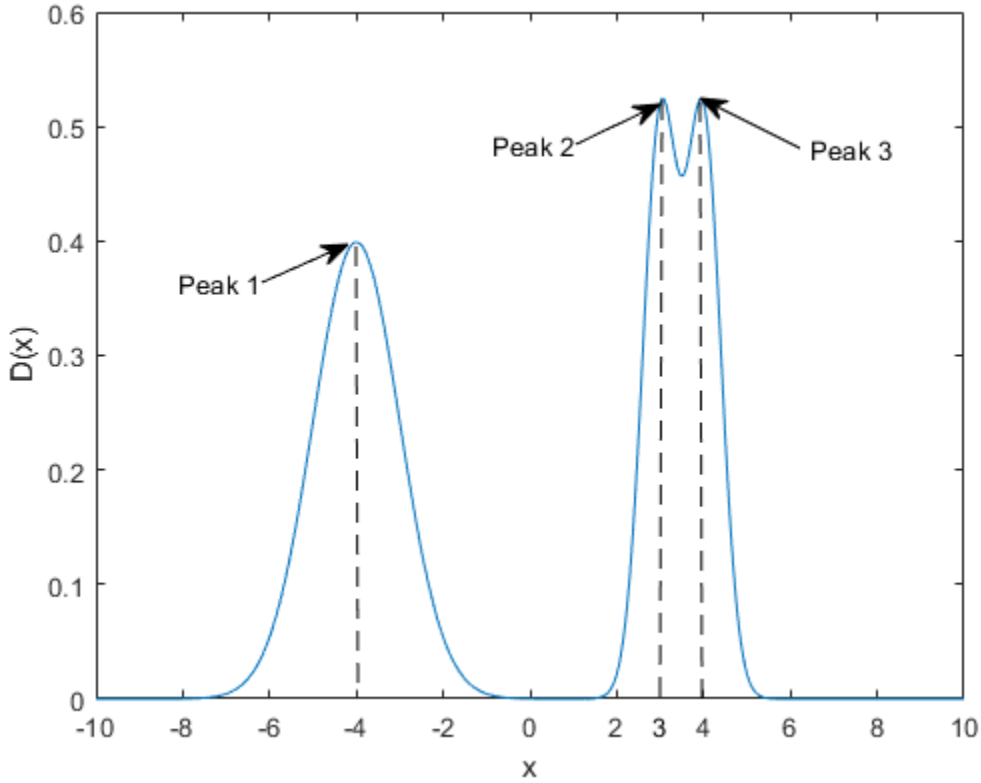
Probability hypothesis density (PHD) is a function defined over the state-space of the tracking system, and its value at a state is defined as the expected number of targets per unit state-space volume. The PHD is usually approximated by a mixture of components, and each component corresponds to an estimate 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

$$D(x) = \sum_{i=1}^{M} w_i N(x \mid m_i, P_i)$$

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 ith component. The weight w_i denotes the number (can be fractional) of targets represented by the ith 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] Granstorm, K., C. Lundquiest, 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] Granstorm, 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] Granstorm, 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`

Classes

`objectDetection` | `trackingSensorConfiguration`

System Objects

`staticDetectionFuser` | `trackerGNN` | `trackerJPDA` | `trackerTOMHT`

Introduced in R2019a

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? (MATLAB).

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 (MATLAB).

`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
`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 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 an object belonging to one of the filter classes: `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 “Initialization”.

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`

`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 $C_1 \le C_2$. 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` method of each tracking filter (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 `HitMissThrehold`. 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*]. A track is deleted if it recorded at least *P* misses in the last *R* updates. The `trackerJPDA` will register a miss on a track's history logic according to the `HitMissThrehold` property. 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*) :

$$PTE(t+\delta t) = \left(1 - DeathRate\right)^{\delta t} PTE(t)$$

where δt 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 `InitialExistenceProbability = NewTargetDensity*DetectionProbability/(ClutterDensity + NewTargetDensity*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.

`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.

`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 contains the corresponding 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 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

structure | array of structures

Confirmed tracks updated to the current time, returned as a structure or array of structures. Each structure corresponds to a track. The confirmation of a track depends on the track logic.

- `'History'` – A track is confirmed if it recorded enough hits during the last few updates to satisfy the `ConfirmationThreshold`.
- `'Integrated'` – A track is confirmed if its probability of existence is higher than the `ConfirmationThreshold` property value.

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 3-479.

Data Types: `struct`

`tentativeTracks` — Tentative tracks

structure | array of structures

Tentative tracks, returned as a structure or array of structures. Each structure corresponds to a track. A track is tentative if the track is not assigned to enough detections and the track cannot be associated with any classified object given by the `ObjectClassID`. In that case, the `IsConfirmed` field of the structure is `false`. The fields of the structure are defined in “Track Structure” on page 3-479.

Data Types: `struct`

`allTracks` — All tracks

structure | array of structures

All tracks, returned as a structure or 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 3-479.

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
`TrackIDsAtStepBeginning`	Track IDs when step began.

`CostMatrix`	Cost matrix for assignment.
`Clusters`	Cell array of cluster reports.
`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 `jpadEvents` 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`

predictTracksToTime	Predict track state
getTrackFilterProperties	Obtain track filter properties
setTrackFilterProperties	Set track filter properties

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
isLocked	Determine if System object is in use
clone	Create duplicate System object
reset	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.

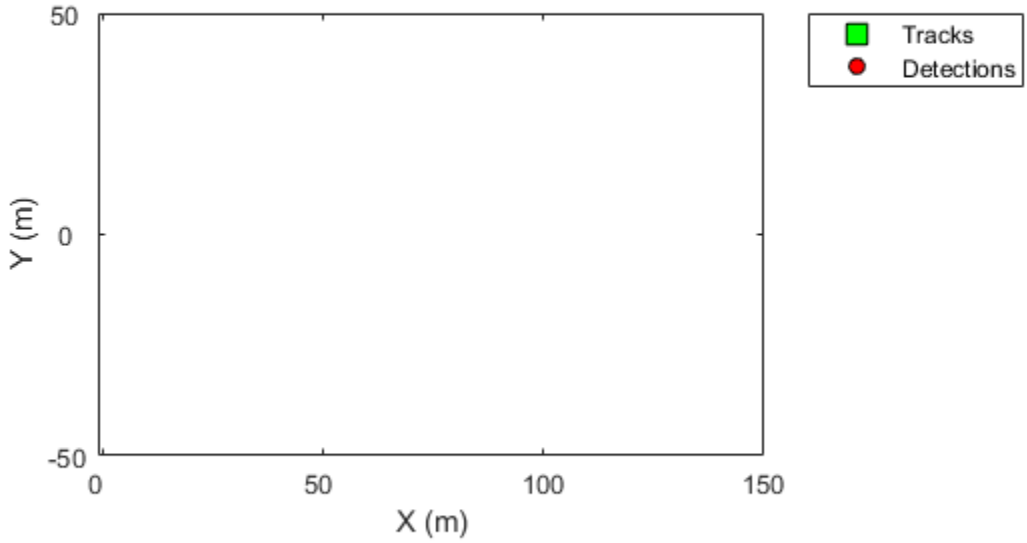
```
tracker = trackerJPDA('TrackLogic','History', 'AssignmentThreshold',100,...
    'ConfirmationThreshold', [4 5], ...
    'DeletionThreshold', [10 10]);
```

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]
```

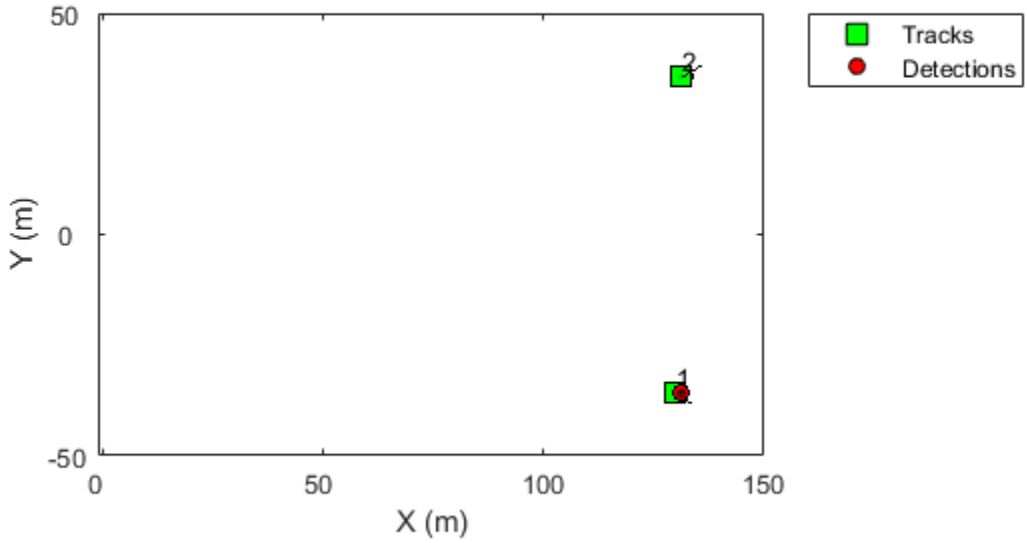
Update the tracker with detections, display cost and marginal probability of association information, and visualize tracks with detections.

```
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
------------------------------
```



Definitions

Track Structure

Track information is returned as an array of structures having the following fields:

Field	Description
`TrackID`	Unique integer that identifies the track.
`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 the update time.
`StateCovariance`	Uncertainty covariance matrix.
`TrackLogic`	The track logic used. The value is either `'Integrated'` or `'History'`.
`TrackLogicState`	The current state of the track logic. • For `'History'` track logic, it is a 1-by-*Q* logical array, where *Q* is the greater of *N* or *R* from the confirmation and deletion thresholds. • For `'Integrated'` track logic, it is the probability of track existence.
`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.

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:

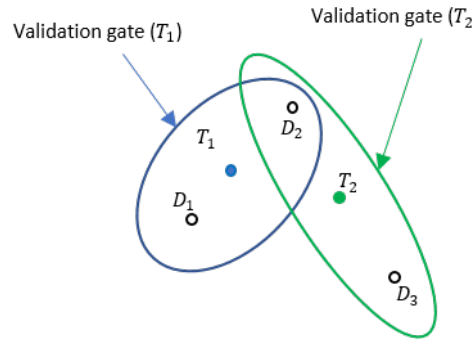
   a. The tracker calculates the distances from detections to existing tracks and forms a `costMatrix`.
   b. The tracker creates a validation matrix based on the assignment threshold (or gate threshold) of the existing tracks. A validation matrix is a binary matrix listing all possible detections-to-track associations. For details, see “Feasible Joint Events” on page 3-481.
   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 3-481.
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 `jpadEvents`.
   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 `IntitializationThreshold`.
   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 Ω 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 Ω 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 Ω_p from the initial validation matrix Ω. For example, with the validation matrix Ω, eight FJE matrices can be obtained:

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

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

As a direct consequence of the two assumptions, the Ω_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`

Classes
`objectDetection` | `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? (MATLAB).

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 ½ 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 (MATLAB).

`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.

Example: `250`

Data Types: `single` | `double`

`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.

Example: `[100 500 2000]`

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.

Example: `[1 0.50 0.12]`

Data Types: `single` | `double`

`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.

Data Types: `single` | `double`

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.

Data Types: `single` | `double`

`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: `single` | `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.

Data Types: `single` | `double`

`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: `single` | `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.

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)
```

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];
```

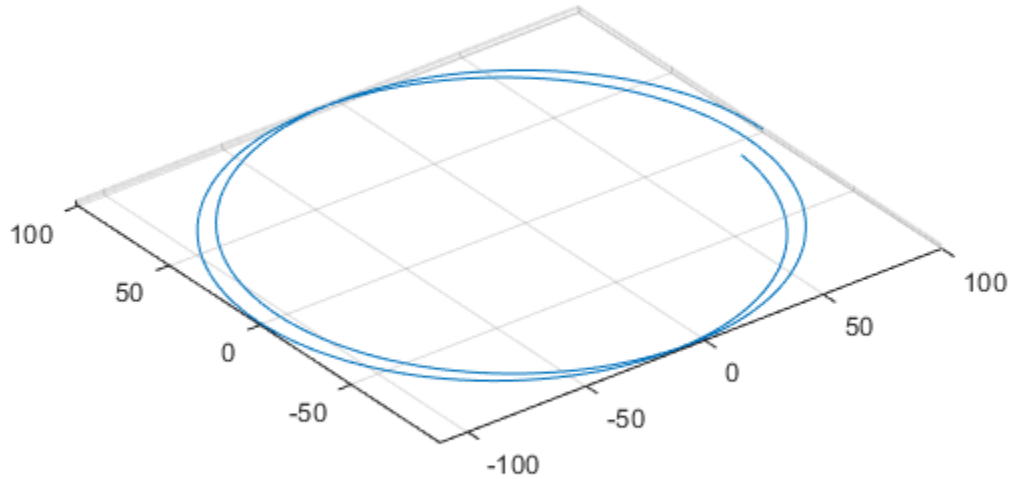
Orient the body along the direction of motion. Apply an acceleration orthogonal to the body in the xy-plane. Acceleration is always in the body frame. Rotate the body as it moves by an angular rotation rate equal to the rotation rate around the origin.

```
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

Classes
`trackingScenario`

System Objects
`kinematicTrajectory` | `waypointTrajectory`

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? (MATLAB).

Creation

Syntax

```
IMU = imuSensor
IMU = imuSensor('accel-gyro')
IMU = imuSensor('accel-mag')
IMU = imuSensor('accel-gyro-mag')
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 accelerator and gyroscope. `imuSensor` and `imuSensor('accel-gyro')` are equivalent creation syntaxes.

`IMU = imuSensor('accel-mag')` returns an `imuSensor` System object with an ideal accelerator and magnetometer.

`IMU = imuSensor('accel-gyro-mag')` returns an `imuSensor` System object with an ideal accelerator, gyroscope, and magnetometer.

`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 (MATLAB).

`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 (ºC)
`25` (default) | real scalar

Operating temperature of the IMU in degrees Celsius, specified as a real scalar.

Tunable: Yes

Data Types: `single` | `double`

`MagneticField` — Magnetic field vector in local NED 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 NED 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 NED coordinate system (m/s^2)

N-by-3 matrix

Acceleration of the IMU in the local NED 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 NED coordinate system (rad/s)

N-by-3 matrix

Angular velocity of the IMU in the local NED 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 NED coordinate system

N-element quaternion column vector | 3-by-3-by-*N*-element rotation matrix

Orientation of the IMU with respect to the local NED 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 NED 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^2)

N-by-3 matrix

Accelerator 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 microtelsa. *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

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.

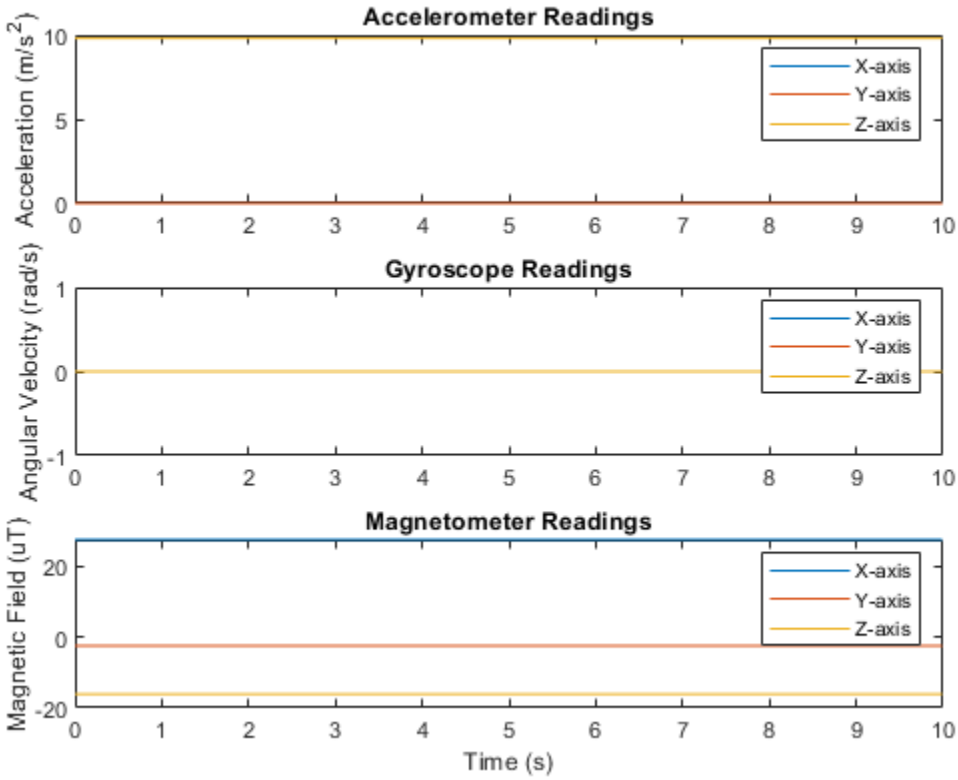
```
numSamples = 1000;
acceleration = zeros(numSamples,3);
angularVelocity = zeros(numSamples,3);
```

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^2 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.

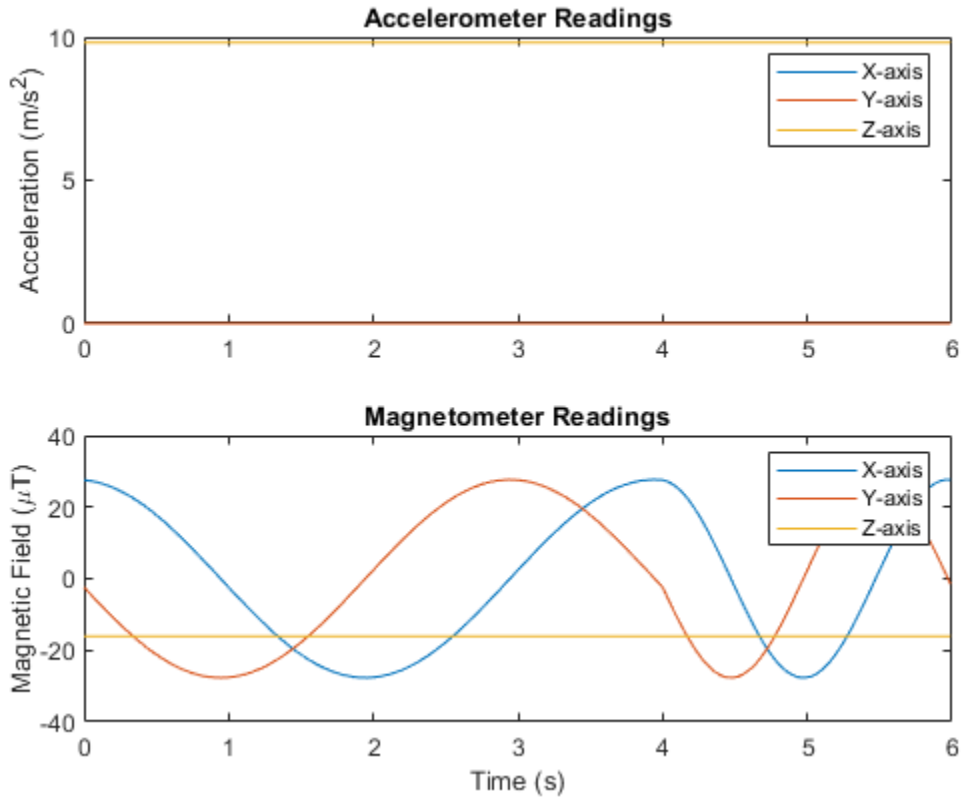
Define the ground-truth motion for a platform that rotates 360 degrees in four seconds, and then another 360 degrees in two seconds. Use `kinematicTrajectory` to output the orientation, acceleration, and angular velocity in the NED coordinate system.

```
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')
```

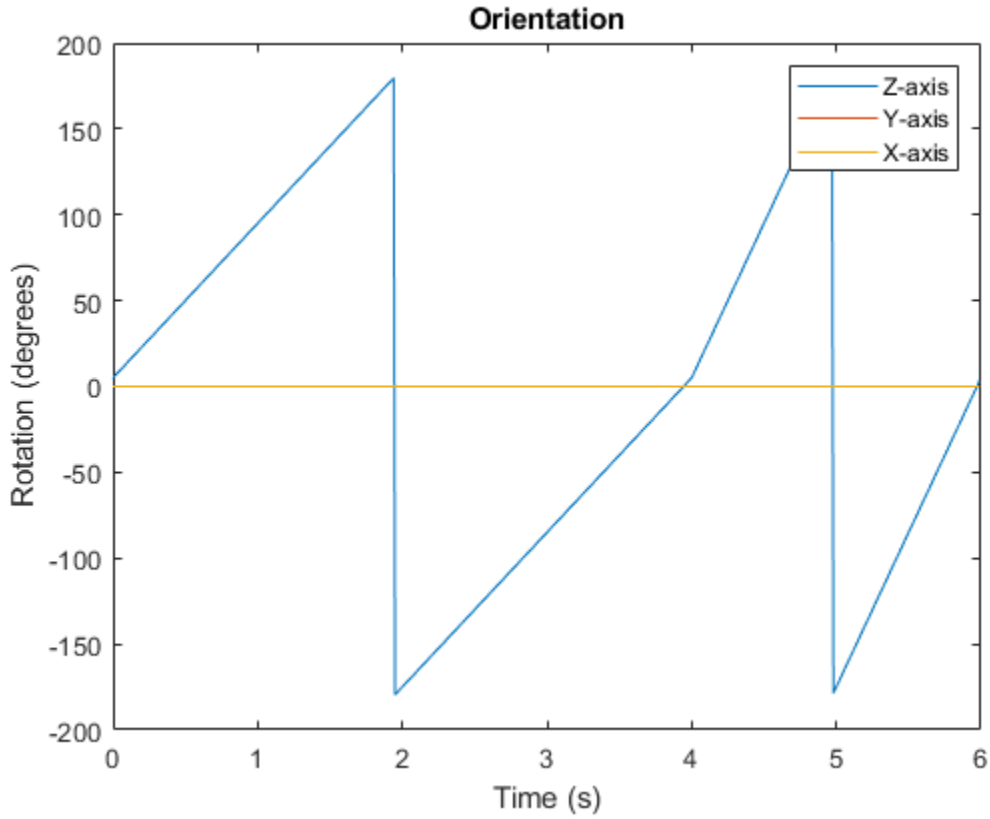


The accelerometer readings indicate that the platform has no translation. The magnetometer readings indicate that the platform is rotating around the *z*-axis.

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.

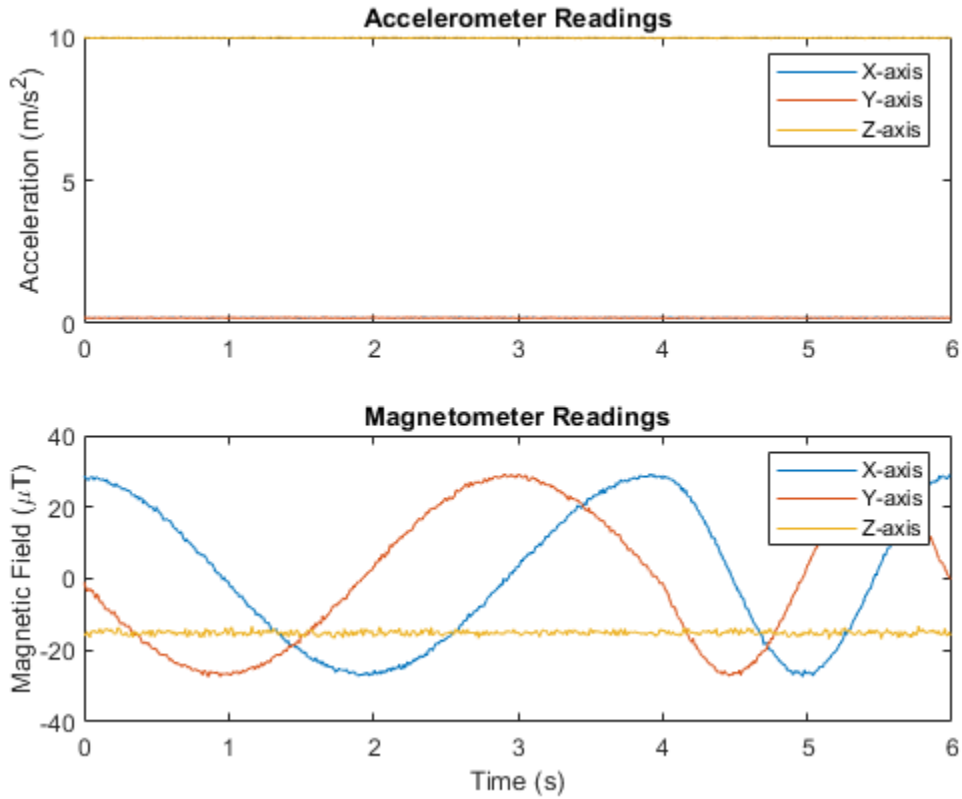
Define the ground-truth motion for a platform that rotates 360 degrees in four seconds, and then another 360 degrees in two seconds. Use `kinematicTrajectory` to output the orientation, acceleration, and angular velocity in the NED coordinate system.

```
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 = accelparams( ...
    '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 (\muT)')
xlabel('Time (s)')
title('Magnetometer Readings')
```

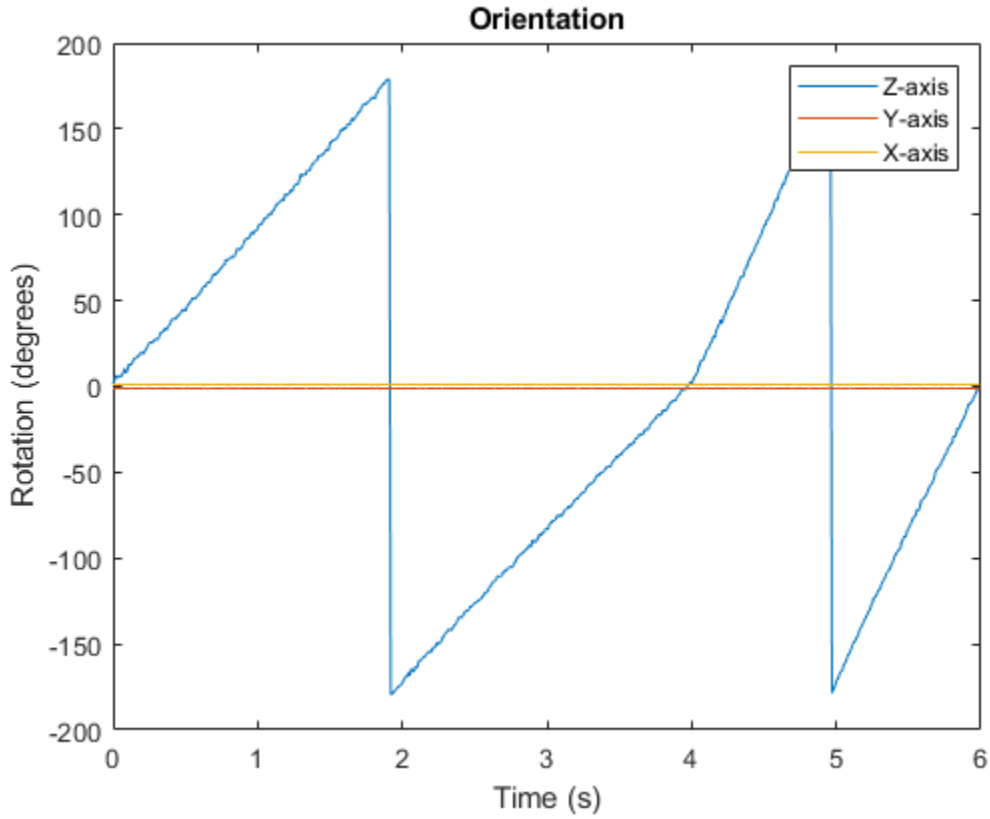


The accelerometer readings indicate that the platform has no translation. The magnetometer readings indicate that the platform is rotating around the *z*-axis.

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 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.

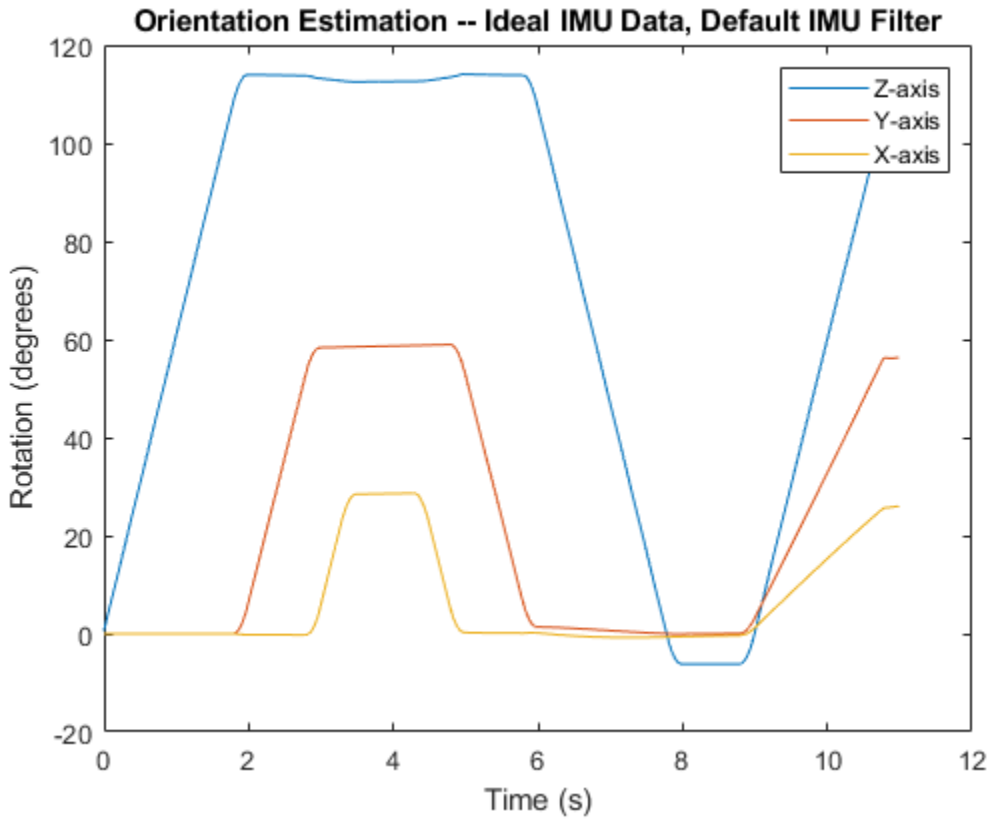
2 Filter the IMU output using the default IMU filter object.

```
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)')
```

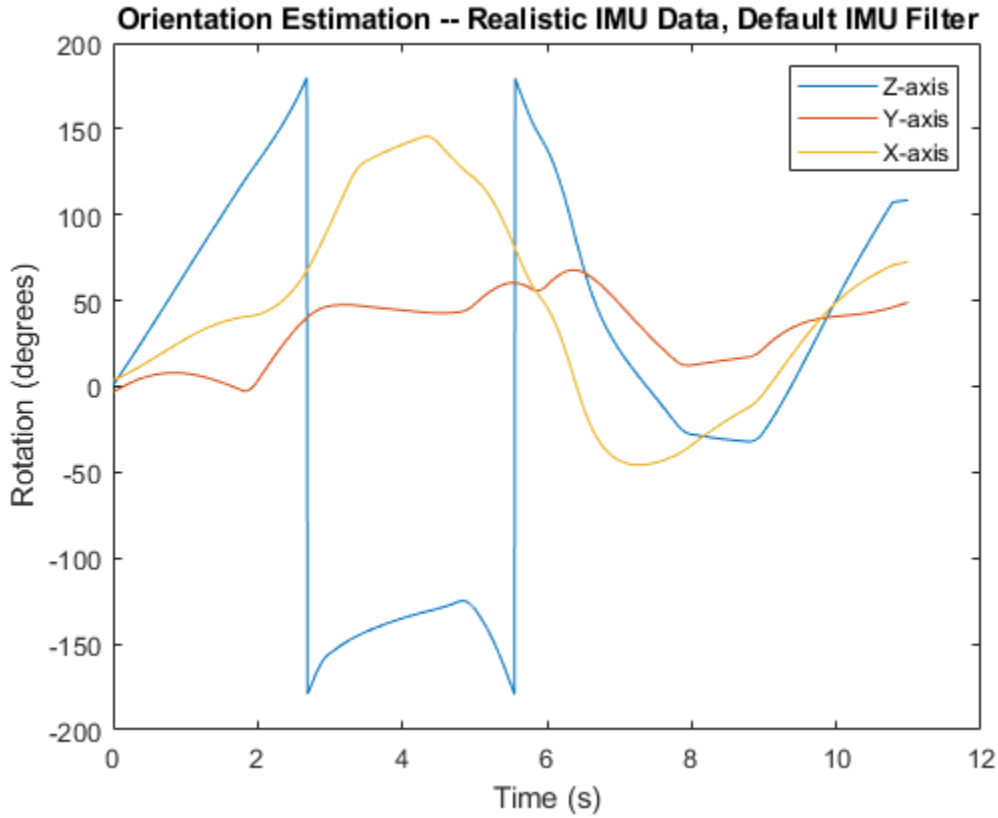
```
ylabel('Rotation (degrees)')
title('Orientation Estimation -- Ideal IMU Data, Default IMU Filter')
legend('Z-axis','Y-axis','X-axis')
```



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, ...
```

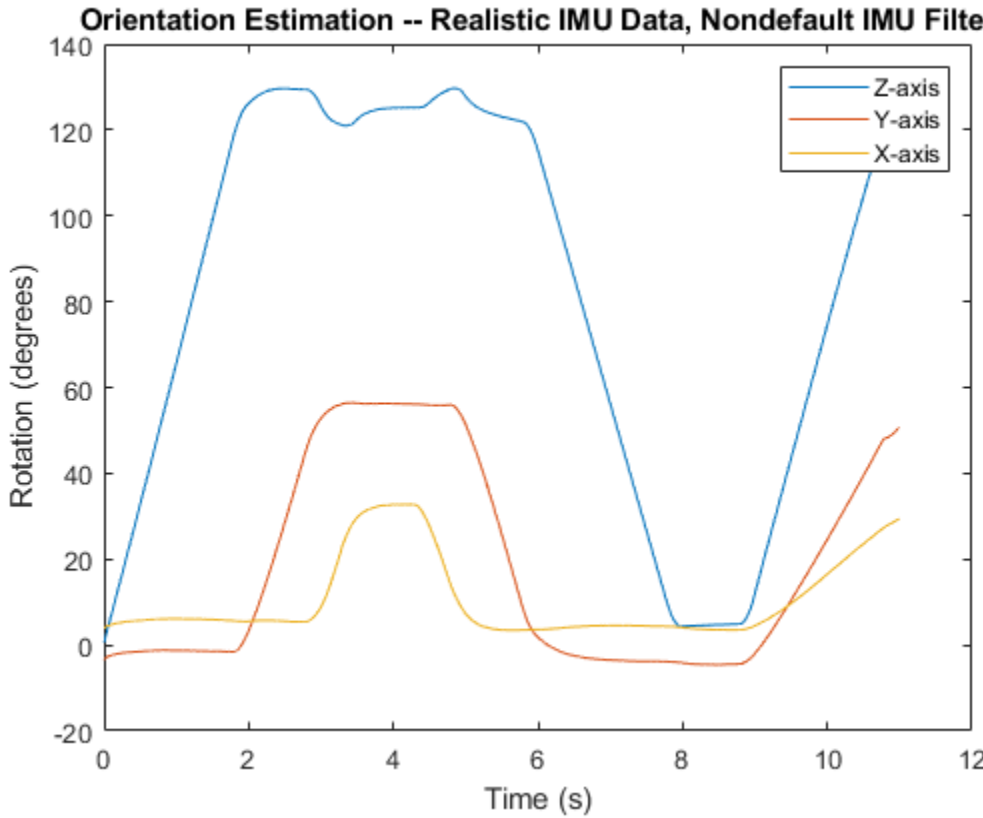
```
    'TemperatureBias', [0.34335 0.34335 0.5886], ...
    'TemperatureScaleFactor',0.02);
IMU.Gyroscope = gyroparams( ...
    'MeasurementRange',4.3633, ...
    'Resolution',0.00013323, ...
    'AxesMisalignment',2, ...
    'NoiseDensity',8.7266e-05, ...
    'TemperatureBias',0.34907, ...
    'TemperatureScaleFactor',0.02, ...
    '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.

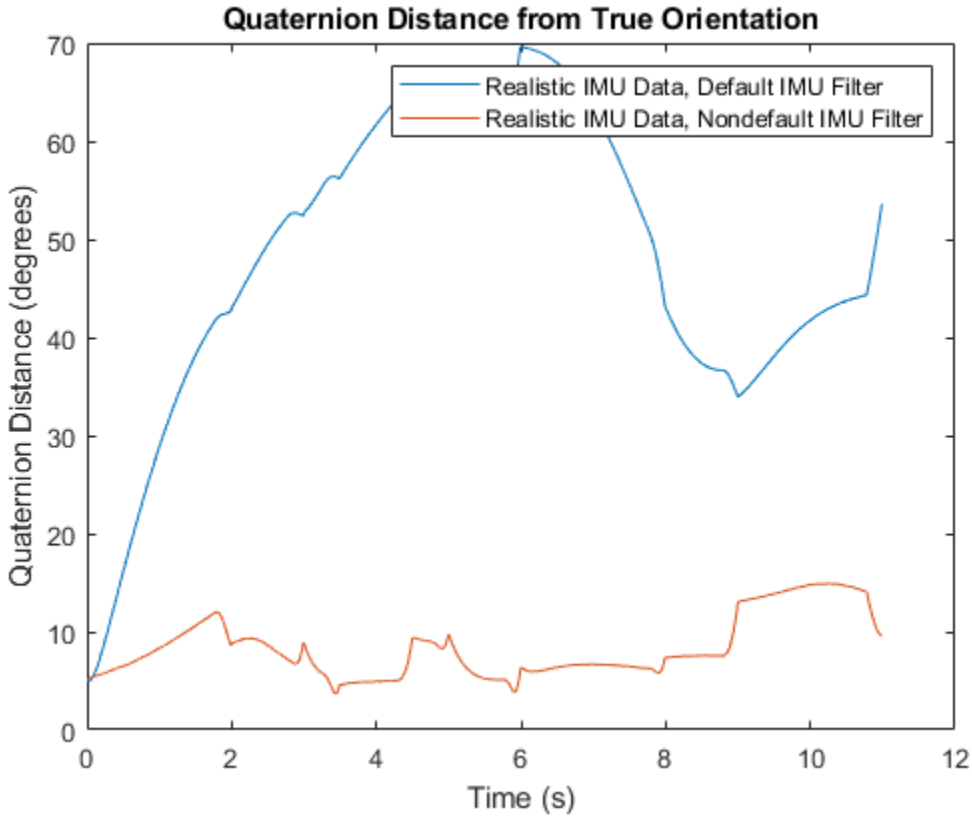
```
aFilter.GyroscopeNoise          = 7.6154e-7;
aFilter.AccelerometerNoise      = 0.0015398;
aFilter.GyroscopeDriftNoise     = 3.0462e-12;
aFilter.LinearAccelerationNoise = 0.00096236;
aFilter.InitialProcessNoise     = aFilter.InitialProcessNoise*10;

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.

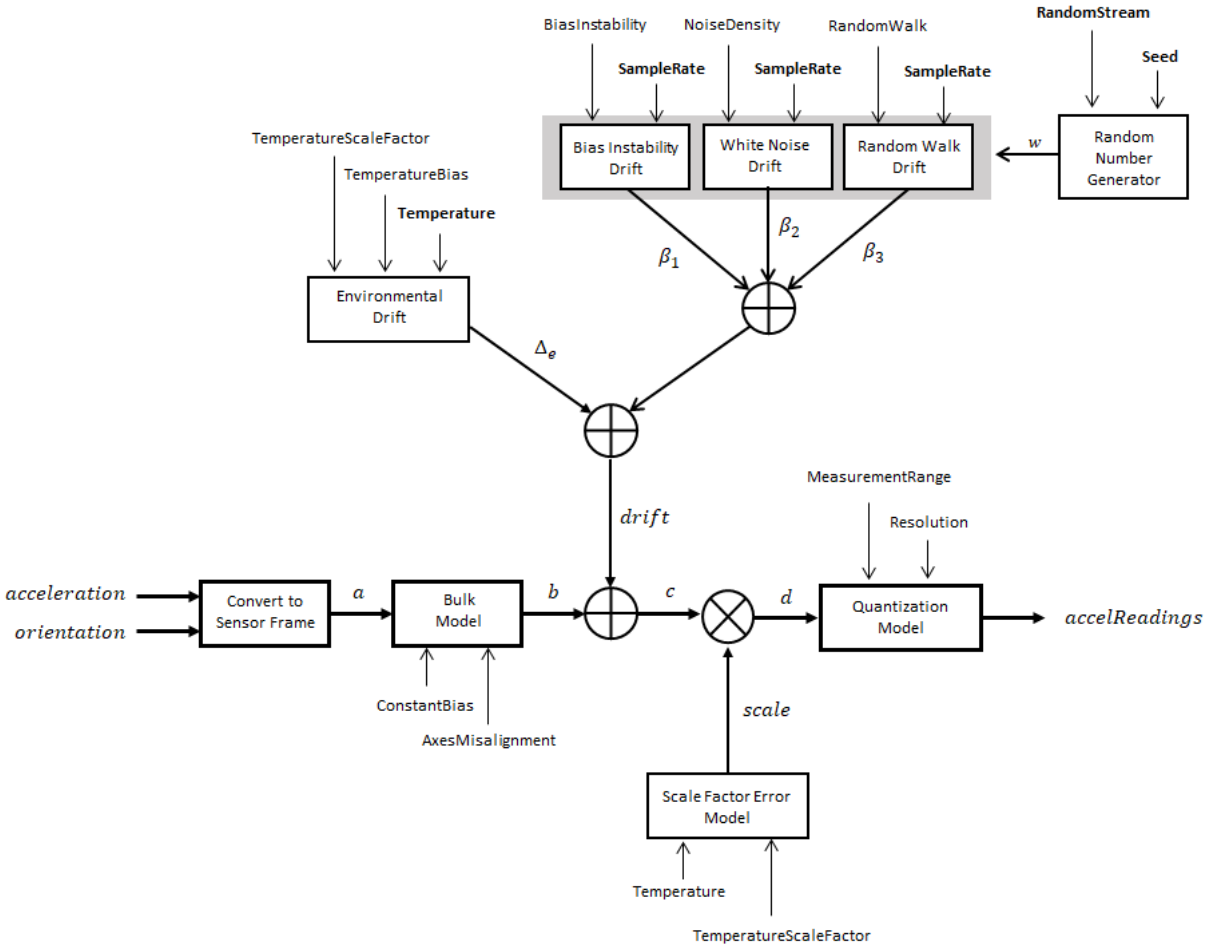
```
qDistDefault = rad2deg(dist(orientationNED,orientationDefault));
qDistNondefault = rad2deg(dist(orientationNED,orientationNondefault));

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.



Convert to Sensor Frame

The ground-truth acceleration is converted from the local frame to the sensor frame using the ground-truth orientation:

$$a = (orientation)(acceleration)^T$$

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:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} \left(a^T\right) \right)^T + \text{ConstantBias}$$

where ConstantBias is a property of `accelparams`, and α_1, α_2, and α_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:

$$H_1(z) = \frac{1}{1 + \left(\frac{2}{\text{SampleRate}} - 1\right) z^{-1}}$$

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 `accelparams`. 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 `accelparams`. The constant 25 corresponds to a standard temperature.

Quantization Model

The quantization is modeled by first saturating the continuous signal model:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

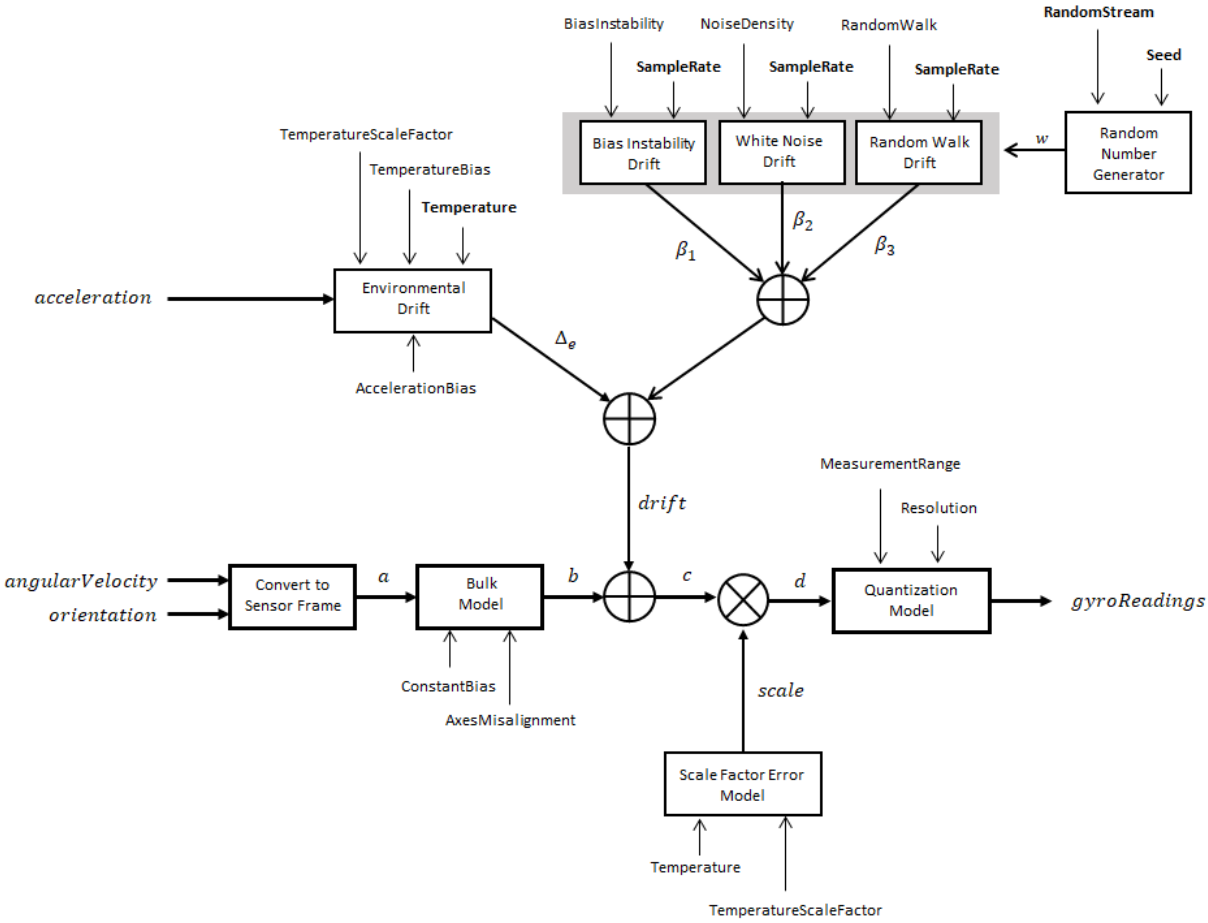
and then setting the resolution:

$$accelReadings = (\text{Resolution})\left(\text{round}\left(\frac{e}{\text{Resolution}}\right)\right)$$

where MeasurementRange is a property of `accelparams`.

Gyroscope

The gyroscope model uses the ground-truth orientation, acceleration, and angular velocity inputs, and the `imuSensor` and `gyroparams` 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:

$$a = (orientation)(angularVelocity)^T$$

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:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} \left(a^T\right) \right)^T + \text{ConstantBias}$$

where ConstantBias is a property of `gyroparams`, and α_1, α_2, and α_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:

$$H_1(z) = \frac{1}{1 + \left(\frac{2}{\text{SampleRate}} - 1\right) z^{-1}}$$

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:

$$e = \begin{cases} \text{MeasurementRange} & \text{if } d > \text{MeasurementRange} \\ -\text{MeasurementRange} & \text{if } -d > \text{MeasurementRange} \\ d & \text{else} \end{cases}$$

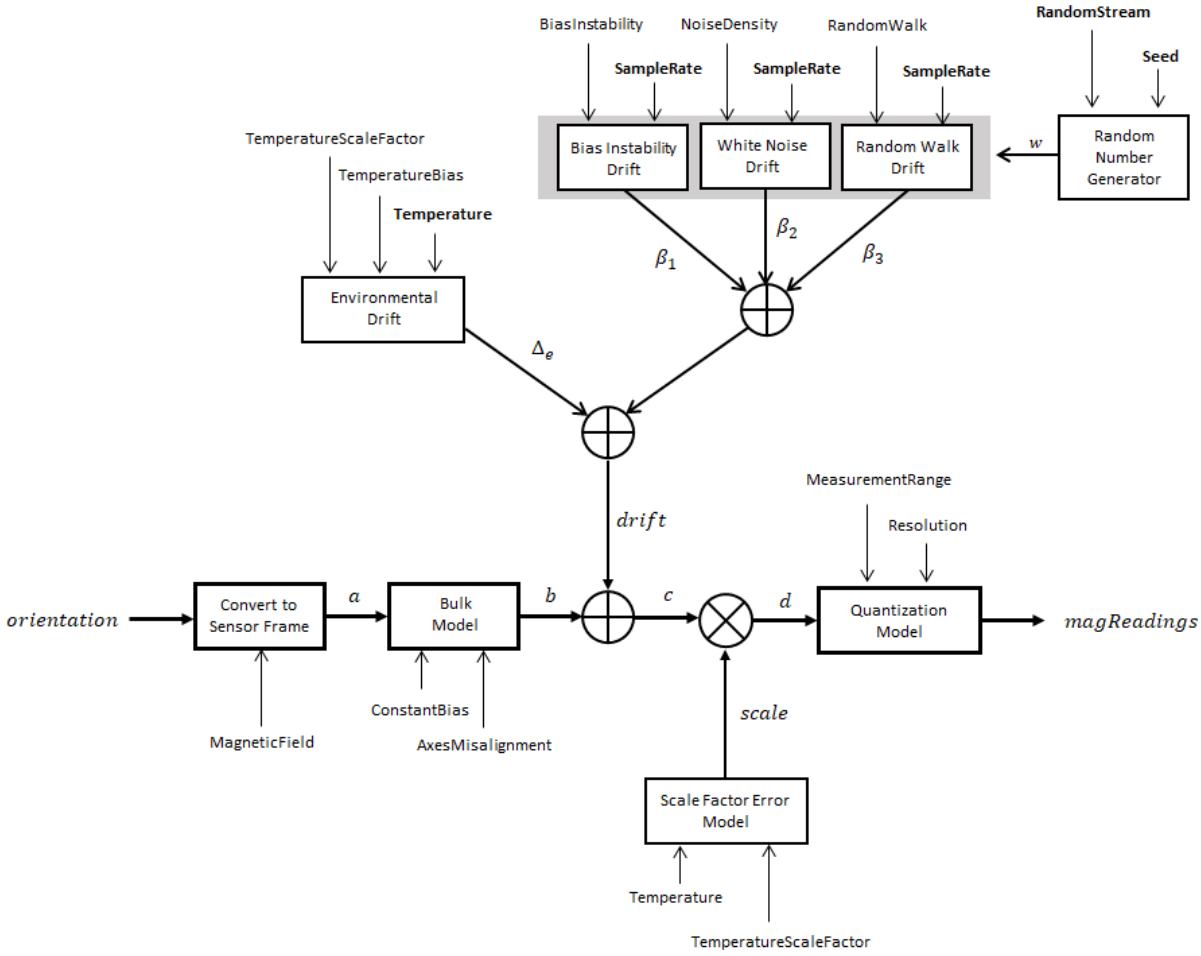
and then setting the resolution:

$$gyroReadings = (\text{Resolution})\left(\text{round}\left(\frac{e}{\text{Resolution}}\right)\right)$$

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:

$$a = (orientation)(acceleration)^T$$

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:

$$b = \left(\begin{bmatrix} 1 & \frac{\alpha_2}{100} & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & 1 & \frac{\alpha_3}{100} \\ \frac{\alpha_1}{100} & \frac{\alpha_2}{100} & 1 \end{bmatrix} \left(a^T\right) \right)^T + \text{ConstantBias}$$

where ConstantBias is a property of `magparams`, and α_1, α_2, and α_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:

$$H_1(z) = \frac{1}{1 + \left(\frac{2}{\text{SampleRate}} - 1\right) z^{-1}}$$

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:

$$magReadings = (\text{Resolution})\left(\text{round}\left(\frac{e}{\text{Resolution}}\right)\right)$$

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`

System Objects
`gpsSensor` | `insSensor`

Topics

“Model IMU, GPS, and INS/GPS”

Introduced in R2018b

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? (MATLAB).

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 (MATLAB).

`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×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×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***			
Sensor – 1	**Sensor – 2**	**. . .**	**Sensor – *S***	**Sensor – 1**	**Sensor – 2**	**. . .**	**Sensor – *S***	**. . .**	**Sensor – 1**	**Sensor – 2**	**. . .**	**Sensor – *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)
```

```
ans =

  3x5 table

    TrackID    ParentID    BranchID                  Scan2
                                                Sensor1    Sensor2    Sensor3    Sensor4    Sens
    _______    ________    ________    ____________________________________________________    ____

       1          0           1            1          0          0          0          0
       2          0           2            2          0          0          0          0
       3          0           3            0          3          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)
```

```
ans =

  9x5 table

    TrackID    ParentID    BranchID                    Scan2
```

			Sensor1	Sensor2	Sensor3	Sensor4	Sens
1	1	1	0	0	0	0	1
3	3	3	0	0	0	0	0
4	0	4	1	0	0	0	0
5	0	5	2	0	0	0	0
6	0	6	0	3	0	0	0
1	1	7	1	0	0	0	1
1	1	8	2	0	0	0	1
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)
```

```
ans =

  3x5 table

    TrackID    ParentID    BranchID                  Scan2
                                      Sensor1    Sensor2    Sensor3    Sensor4    Sens

    _______    ________    ________    ____________________________________________    ____

       1          0           1           1          0          0          0          0
       2          0           2           2          0          0          0          0
       3          0           3           0          3          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)
```

```
ans =

  9x5 table

    TrackID    ParentID    BranchID                  Scan2
                                        Sensor1    Sensor2    Sensor3    Sensor4    Sens
    _______    ________    ________    ____________________________________________    ____

       1          1           1           0          0          0          0          1
       3          3           3           0          0          0          0          0
       4          0           4           1          0          0          0          0
       5          0           5           2          0          0          0          0
       6          0           6           0          3          0          0          0
       1          1           7           1          0          0          0          1
       1          1           8           2          0          0          0          1
       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.

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***			
Sensor – 1	**Sensor – 2**	**. . .**	**Sensor – *S***	**Sensor – 1**	**Sensor – 2**	**. . .**	**Sensor – *S***	**. . .**	**Sensor – 1**	**Sensor – 2**	**. . .**	**Sensor – *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? (MATLAB).

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 (MATLAB).

`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

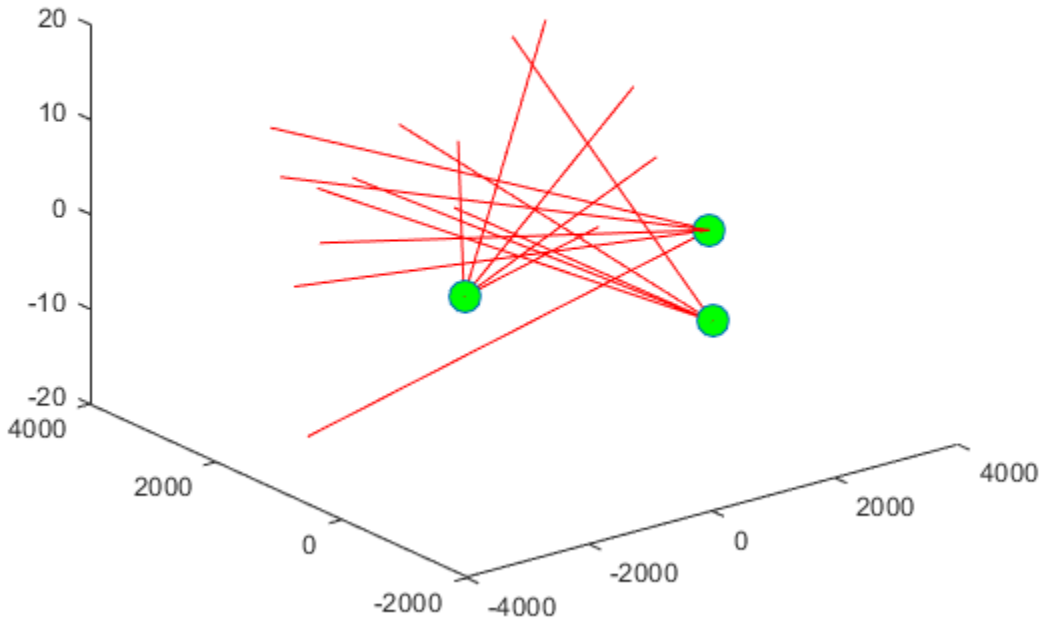
Fuse angle-only detections from three ESM sensors.

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(:);
    plotData(:,3*i-2) = [xt;yt;zt] + originPos(:);
    plotData(:,3*i-1) = originPos(:);
    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,'Marke
```



Create a `staticDetectionFuser` to fuse angle-only detections using the measurement fusion function `triangulateLOS`.

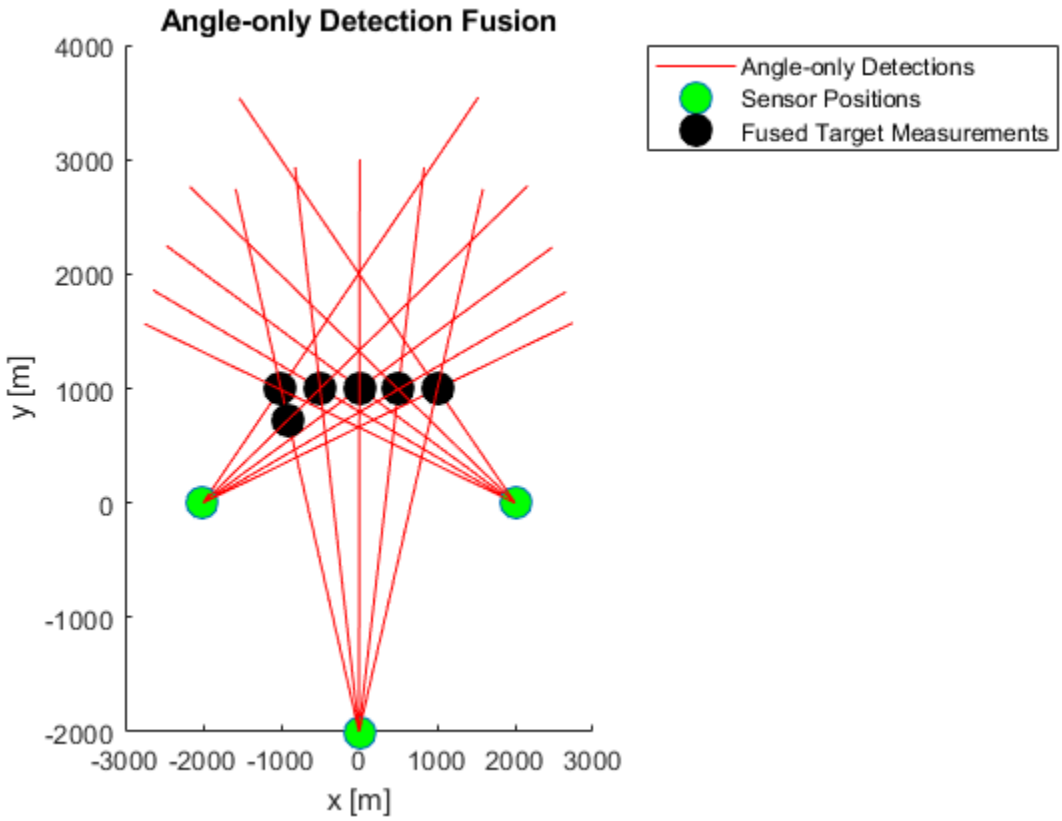
```
fuser = staticDetectionFuser('MeasurementFusionFcn','triangulateLOS','MaxNumSensors',3)
```

```
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.

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

4

Blocks in Sensor Fusion and Tracking Toolbox

5

Apps in Sensor Fusion and Tracking Toolbox