Sensor Fusion and Tracking Toolbox™

Getting Started Guide

MATLAB® & SIMULINK®

R2018b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Sensor Fusion and Tracking Toolbox™ Getting Started Guide

Revision History

September 2018	Online only	New for Version 1.0 (Release 2018b)

Contents

1 Introduction

2 Inertial Sensor Models

3 Orientation

4 Spatial Representation

5 Pose

6 Tracking Scenario

7 Radar Detections

8 Multi-Object Tracking

1

Introduction

Sensor Fusion and Tracking Toolbox Product Description

Design and simulate multisensor tracking and navigation systems

Sensor Fusion and Tracking Toolbox includes algorithms and tools for the design, simulation, and analysis of systems that fuse data from multiple sensors to maintain position, orientation, and situational awareness. Reference examples provide a starting point for implementing components of airborne, ground-based, shipborne, and underwater surveillance, navigation, and autonomous systems.

The toolbox includes multi-object trackers, sensor fusion filters, motion and sensor models, and data association algorithms that let you evaluate fusion architectures using real and synthetic data. With Sensor Fusion and Tracking Toolbox you can import and define scenarios and trajectories, stream signals, and generate synthetic data for active and passive sensors, including RF, acoustic, EO/IR, and GPS/IMU sensors. You can also evaluate system accuracy and performance with standard benchmarks, metrics, and animated plots.

For simulation acceleration or desktop prototyping, the toolbox supports C code generation.

Key Features

- Algorithms for multi-object tracking, sensor fusion, and inertial sensing
- Active and passive sensor models, including RF, acoustic, EO/IR, and GPS/IMU sensors, for testing fusion algorithms
- Reference examples for airborne, ground-based, shipborne, and underwater surveillance, navigation, and autonomous systems
- Scenario and trajectory import and generation
- C code generation for simulation acceleration or desktop prototyping (with MATLAB® Coder™)

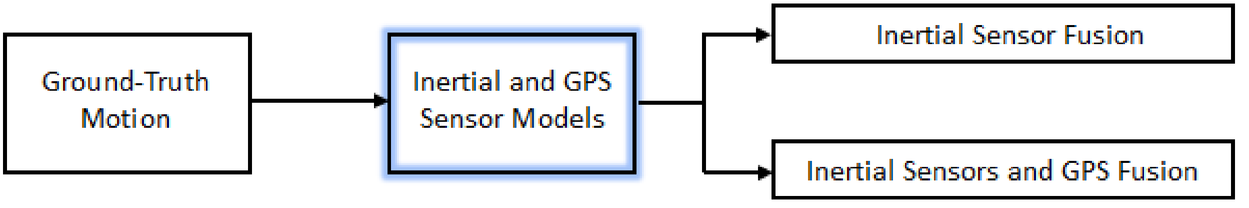
2

Inertial Sensor Models

Model IMU, GPS, and INS/GPS

Sensor Fusion and Tracking Toolbox enables you to model inertial measurement units (IMU), Global Positioning Systems (GPS), and inertial navigation systems (INS). You can model specific hardware by setting properties of your models to values from hardware datasheets. You can tune environmental and noise properties to mimic real-world environments. You can use these models to test and validate your fusion algorithms or as placeholders while developing larger applications.

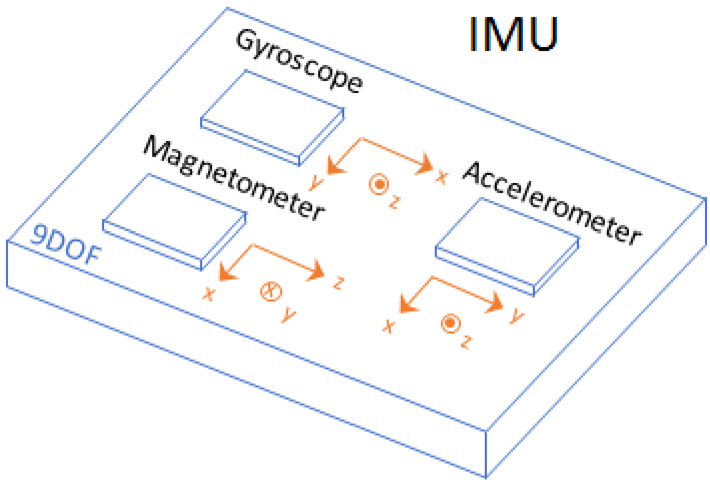
This tutorial provides an overview of inertial sensor and GPS models in Sensor Fusion and Tracking Toolbox.



To learn how to generate the ground-truth motion that drives the sensor models, see `waypointTrajectory` and `kinematicTrajectory`. For a tutorial on fusing inertial sensor data, see “Determine Orientation Using Inertial Sensors” on page 3-2.

Inertial Measurement Unit

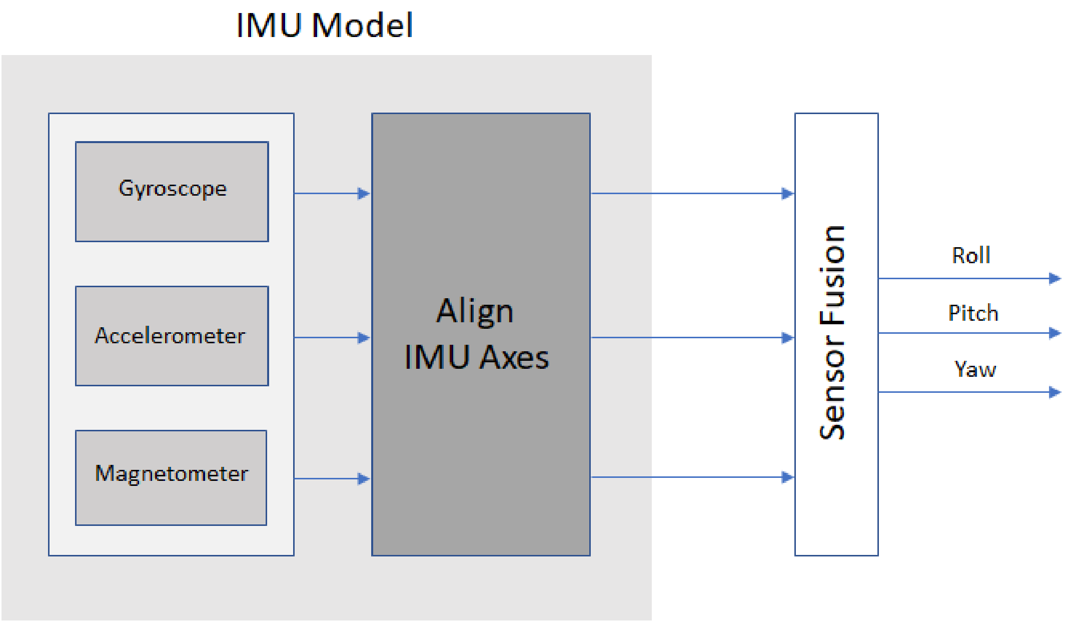
An IMU is an electronic device mounted on a platform. The IMU consists of individual sensors that report various information about the platform's motion. IMUs combine multiple sensors, which can include accelerometers, gyroscopes, and magnetometers.



With this toolbox, measurements returned from an IMU model use the following unit and coordinate conventions.

Output	Description	Units	Coordinate System
Acceleration	Current accelerometer reading	m/s^2	Sensor Body
Angular velocity	Current gyroscope reading	rad/s	Sensor Body
Magnetic field	Current magnetometer reading	μT	Sensor Body

Usually, the data returned by IMUs is fused together and interpreted as roll, pitch, and yaw of the platform. Real-world IMU sensors can have different axes for each of the individual sensors. The models provided by Sensor Fusion and Tracking Toolbox assume that the individual sensor axes are aligned.



To create an IMU sensor model, use the `imuSensor` System object™.

```
IMU = imuSensor
```

```
IMU = 

  imuSensor with properties:

          IMUType: 'accel-gyro'
       SampleRate: 100
      Temperature: 25
    Accelerometer: [1×1 accelparams]
        Gyroscope: [1×1 gyroparams]
     RandomStream: 'Global stream'
```

The default IMU model contains an ideal accelerometer and an ideal gyroscope. The `accelparams` and `gyroparams` objects define the accelerometer and gyroscope configuration. You can set the properties of these objects to mimic specific hardware and environments. For more information on IMU parameter objects, see `accelparams`, `gyroparams`, and `magparams`.

To model receiving IMU sensor data, call the IMU model with the ground-truth acceleration and angular velocity of the platform:

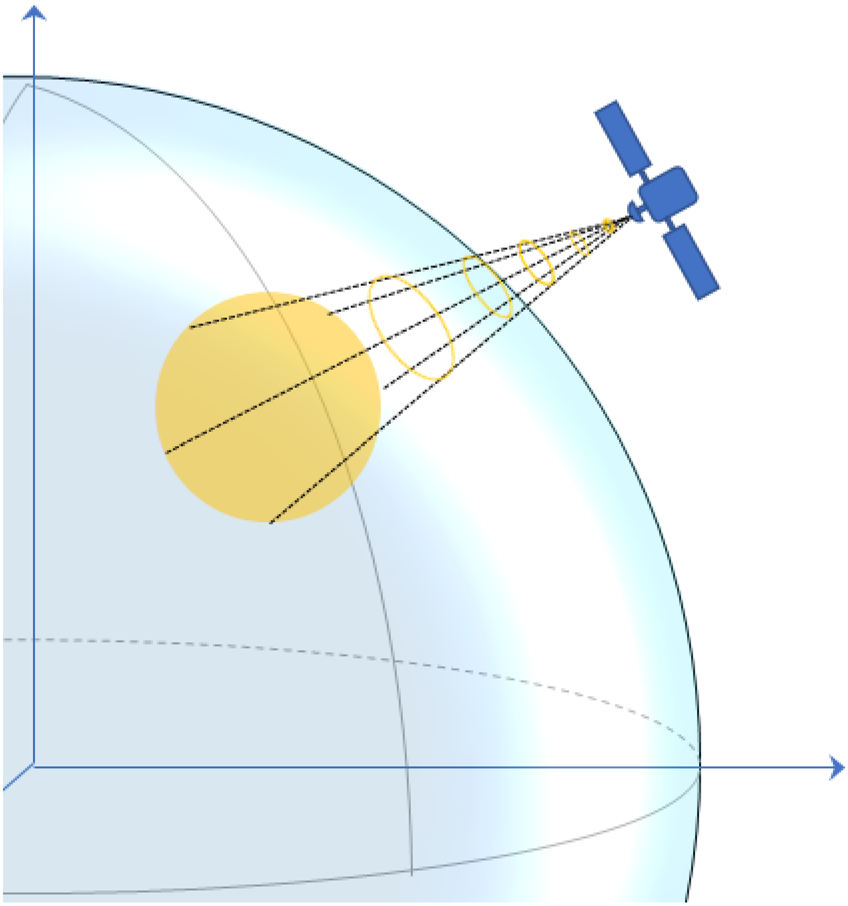
```
trueAcceleration = [1 0 0];
trueAngularVelocity = [1 0 0];
[accelerometerReadings,gyroscopeReadings] = IMU(trueAcceleration,trueAngularVelocity)
```

```
accelerometerReadings =

   -1.0000         0    9.8100


gyroscopeReadings =

     1     0     0
```

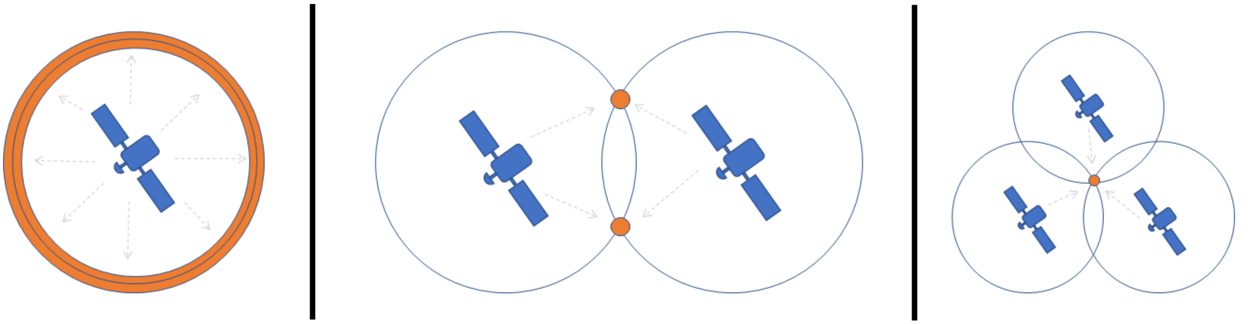
You can generate the ground-truth trajectories that you input to the IMU model using `kinematicTrajectory` and `waypointTrajectory`.

Global Positioning System

A global positioning system (GPS) provides 3-D position information for platforms (receivers) on the surface of the Earth.



GPS consists of a constellation of satellites that continuously orbit the earth. The satellites maintain a configuration such that a platform is always within view of at least four satellites. By measuring the flight time of signals from the satellites to the platform, the position of the platform can be trilaterated. Satellites timestamp a broadcast signal, which is compared to the platform's clock upon receipt. Three satellites are required to trilaterate a position in three dimensions. The fourth satellite is required to correct for clock synchronization errors between the platform and satellites.



The GPS simulation provided by Sensor Fusion and Tracking Toolbox models the platform (receiver) data that has already been processed and interpreted as altitude, latitude, longitude, velocity, groundspeed, and course.

Measurements returned from the GPS model use the following unit and coordinate conventions.

Output	Description	Units	Coordinate System
LLA	Current global position reading in geodetic coordinates, based on wgs84Ellipsoid Earth model	degrees (latitude), degrees (longitude), meters (altitude)	LLA
Velocity	Current velocity reading from GPS	m/s	local NED
Groundspeed	Current groundspeed reading from GPS	m/s	local NED
Course	Current course reading from GPS	degrees	local NED

The GPS model enables you to set high-level accuracy and noise parameters, as well as the receiver update rate and a reference location.

To create a GPS model, use the `gpsSensor` System object.

```
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
```

To model receiving GPS sensor data, call the GPS model with the ground-truth position and velocity of the platform:

```
truePosition = [1 0 0];
trueVelocity = [1 0 0];
[LLA,velocity,groundspeed,course] = GPS(truePosition,trueVelocity)
```

```
LLA =

    0.0000    0.0000    0.3031


velocity =

    1.0919   -0.0008   -0.1308


groundspeed =

    1.0919


course =

  359.9566
```

You can generate the ground-truth trajectories that you input to the GPS model using `kinematicTrajectory` and `waypointTrajectory`.

Inertial Navigation System and Global Positioning System

An inertial navigation system (INS) uses inertial sensors like those found on an IMU: accelerometers, gyroscopes, and magnetometers. An INS fuses the inertial sensor data to calculate position, orientation, and velocity of a platform. An INS/GPS uses GPS data to correct the INS. Typically, the INS and GPS readings are fused with an extended Kalman filter, where the INS readings are used in the prediction step, and the GPS readings are used in the update step. A common use for INS/GPS is dead-reckoning when the GPS signal is unreliable.

"INS/GPS" refers to the entire system, including the filtering. The INS/GPS simulation provided by Sensor Fusion and Tracking Toolbox models an INS/GPS and returns the position, velocity, and orientation reported by the inertial sensors and GPS receiver based on a ground-truth motion.

Measurements returned from the INS/GPS use the following unit and coordinate conventions.

Output	Description	Units	Coordinate System
Position	Current position reading from the INS/GPS	meters	local NED
Velocity	Current velocity reading from the INS/GPS	m/s	local NED
Orientation	Current orientation reading from the INS/GPS	quaternion or rotation matrix	N/A

To create a INS/GPS model, use the `insSensor` System object. You can model a real-world INS/GPS system by tuning the accuracy of your fused data: roll, pitch, yaw, position, and velocity.

```
INS = insSensor
```

```
INS = 

  insSensor with properties:

        RollAccuracy: 0.2               deg
```

```
       PitchAccuracy: 0.2               deg
         YawAccuracy: 1                 deg
    PositionAccuracy: 1                 m
    VelocityAccuracy: 0.05              m/s
        RandomStream: 'Global stream'
```

To model receiving INS/GPS sensor data, call the INS/GPS model with the ground-truth position and velocity and orientation of the platform:

```
trueMotion = struct( ...
        'Position',[0 0 0], ...
        'Velocity',[0 0 0], ...
        'Orientation',quaternion(1,0,0,0));
measurement = INS(trueMotion)
```

```
measurement = 

  struct with fields:

    Orientation: [1×1 quaternion]
       Position: [0.2939 -0.7873 0.8884]
       Velocity: [-0.0574 -0.0534 -0.0405]
```

See Also

gpsSensor | imuSensor | insSensor

More About

- "Introduction to Simulating IMU Measurements"
- "Inertial Sensor Noise Analysis Using Allan Variance"

External Websites

- https://www.gps.gov/systems/gps/

Orientation

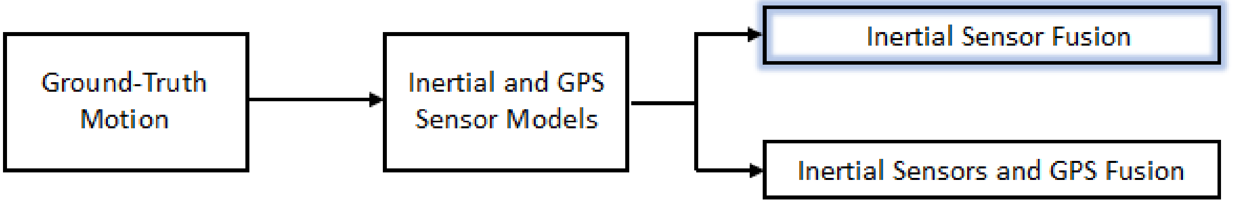
Determine Orientation Using Inertial Sensors

Sensor Fusion and Tracking Toolbox enables you to fuse data read from an inertial measurement unit (IMU) to estimate orientation and angular velocity:

- `ecompass` –– Fuse accelerometer and magnetometer readings
- `imufilter` –– Fuse accelerometer and gyroscope readings
- `ahrsfilter` –– Fuse accelerometer, gyroscope, and magnetometer readings

More sensors on an IMU result in a more robust orientation estimation. The sensor data can be cross-validated, and the information the sensors convey is orthogonal.

This tutorial provides an overview of inertial sensor fusion for IMUs in Sensor Fusion and Tracking Toolbox.



To learn how to model inertial sensors and GPS, see “Model IMU, GPS, and INS/GPS” on page 2-2. To learn how to generate the ground-truth motion that drives sensor models, see `waypointTrajectory` and `kinematicTrajectory`.

You can also fuse IMU readings with GPS readings to estimate pose. See “Determine Pose Using Inertial Sensors and GPS” on page 5-2 for an overview.

Estimate Orientation Through Inertial Sensor Fusion

This example shows how to use 6-axis and 9-axis fusion algorithms to compute orientation. Sensor Fusion and Tracking Toolbox™ includes several algorithms to compute orientation from inertial measurement units (IMUs) and magnetic-angular rate-gravity (MARG) units. This example covers the basics of orientation and how to use these algorithms.

Orientation

An object's orientation describes its rotation relative to some coordinate system, sometimes called a parent coordinate system, in three-dimensions.

Sensor Fusion and Tracking Toolbox uses North-East-Down (NED) as a fixed, parent coordinate system. NED is sometimes referred to as the global coordinate system or reference frame. In the NED reference frame, the X-axis points north, the Y-axis points east, and the Z-axis points downward. The X-Y plane of NED is considered to be the local tangent plane of the Earth. Depending on the algorithm, North may be either Magnetic North or True North. The algorithms in this example use Magnetic North.

An object can be thought of as having its own coordinate system, often called the local or child coordinate system. This child coordinate system rotates with the object relative to the parent coordinate system. If there is no translation, the origins of both coordinate systems overlap.

The orientation quantity computed in Sensor Fusion and Tracking Toolbox is a rotation that takes quantities from the parent reference frame to the child reference frame. The rotation is represented by a quaternion or rotation matrix.

Types of Sensors

For orientation estimation, three types of sensors are commonly used: accelerometers, gyroscopes and magnetometers. Accelerometers measure proper acceleration. Gyroscopes measure angular velocity. Magnetometers measure the local magnetic field. Different algorithms are used to fuse different combinations of sensors to estimate orientation.

Sensor Data

Through most of this example the same set of sensor data is used. Accelerometer, gyroscope and magnetometer sensor data was recorded while a device moved in three different directions: first around its local Y-axis, then around its Z-axis, and finally around its X-axis. The device's X-axis was generally pointed southward for the duration of the experiment.

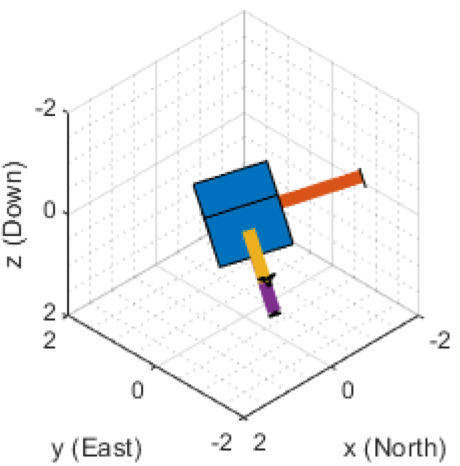
```
ld = load('rpy_9axis.mat');

acc = ld.sensorData.Acceleration;
gyro = ld.sensorData.AngularVelocity;
mag = ld.sensorData.MagneticField;
```

```
viewer = fusiondemo.OrientationViewer;
```

Accelerometer-Magnetometer Fusion

The `ecompass` function fuses accelerometer and magnetometer data. This is a memoryless algorithm that requires no parameter tuning, but is highly susceptible to sensor noise.

```
qe = ecompass(acc, mag);
for ii=1:size(acc,1)
    viewer(qe(ii));
    pause(0.01);
end
```

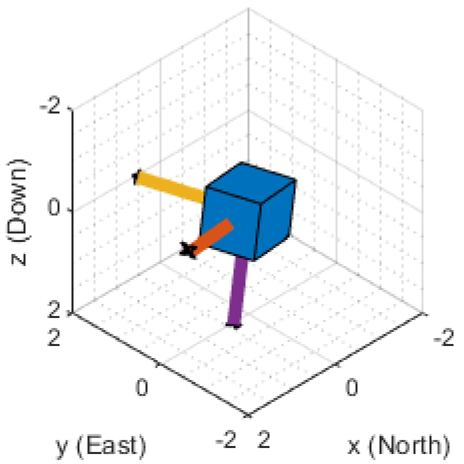


Note that the `ecompass` algorithm correctly finds the location of north. However, because the function is memoryless the estimated motion is not smooth. It is dramatically affected by the noise in the accelerometer and magnetometer. Some of the techniques presented in the Lowpass Filter Orientation Using Quaternion SLERP could be used to smooth the motion.

Accelerometer-Gyroscope Fusion

The `imufilter` System object fuses accelerometer and gyroscope data using an internal error-state Kalman filter. The filter is capable of removing the gyroscope's bias noise, which drifts over time.

```
ifilt = imufilter('SampleRate', ld.Fs);
for ii=1:size(acc,1)
    qimu = ifilt(acc(ii,:), gyro(ii,:));
    viewer(qimu);
    pause(0.01);
end
```



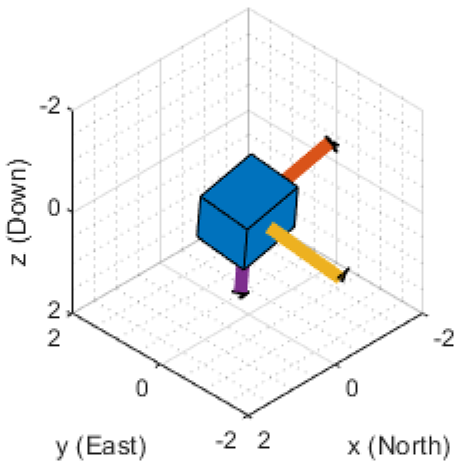
Although the `imufilter` algorithm produces a significantly smoother estimate of the motion, as compared to the `ecompass`, it does not correctly estimate the direction of North. The `imufilter` does not process magnetometer data so it simply assumes the device's X-axis is initially pointing northward. The motion estimate given by `imufilter` is relative to the initial estimated orientation.

Accelerometer-Gyroscope-Magnetometer Fusion

An attitude and heading reference system (AHRS) consists of a 9-axis system that uses an accelerometer, gyroscope and magnetometer to compute orientation. The `ahrsfilter` System object combines the best of the previous algorithms to produce a smoothly changing estimate of the device orientation, while correctly estimating the direction of

North. This algorithm also uses an error-state Kalman filter. In addition to gyroscope bias removal, the `ahrsfilter` has some ability to detect and reject mild magnetic jamming.

```
ifilt = ahrsfilter('SampleRate', ld.Fs);
for ii=1:size(acc,1)
    qahrs = ifilt(acc(ii,:), gyro(ii,:), mag(ii,:));
    viewer(qahrs);
    pause(0.01);
end
```



Tuning Filter Parameters

Tuning the parameters of the `ahrsfilter` and `imufilter` to match specific hardware sensors can improve performance. It is important to also take into account the environment of the sensor. The `imufilter` parameters are a subset of the `ahrsfilter` parameters. The `AccelerometerNoise`, `GyroscopeNoise`, `MagnetometerNoise`, and `GyroscopeDriftNoise` are measurement noises. The sensors' datasheets help determine those values.

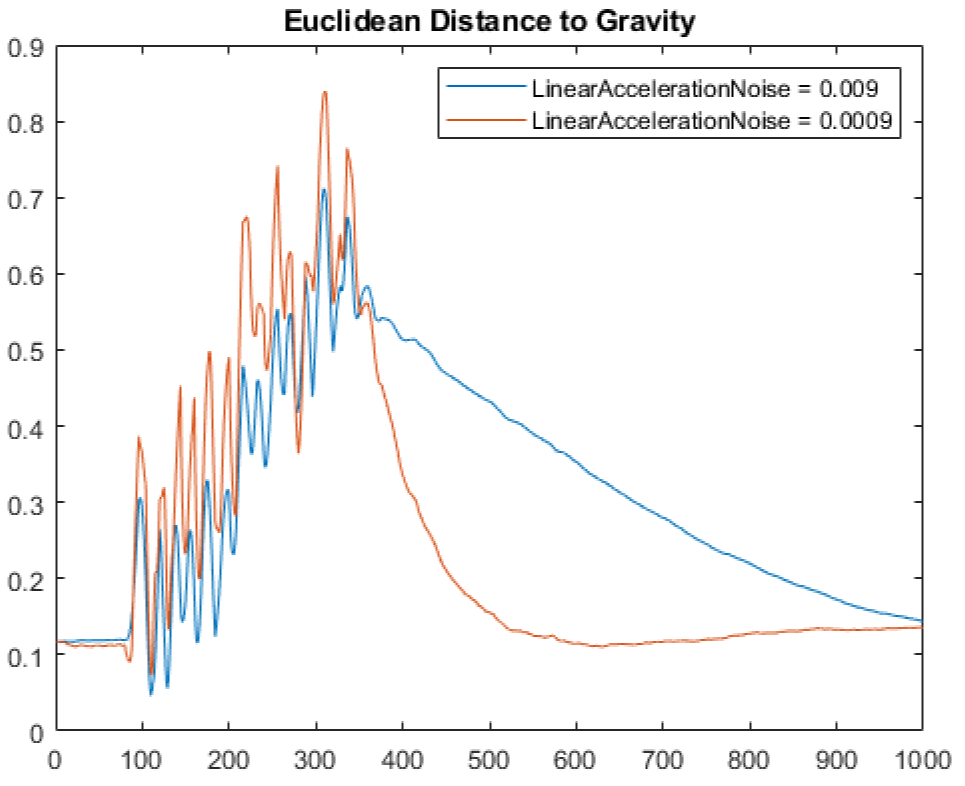
The `LinearAccelerationNoise` and `LinearAccelerationDecayFactor` govern the filter's response to linear (translational) acceleration. Shaking a device is a simple example of adding linear acceleration.

Consider how an `imufilter` with a `LinearAccelerationNoise` of 9e-3 $(m/s^2)^2$ responds to a shaking trajectory, compared to one with a `LinearAccelerationNoise` of 9e-4 $(m/s^2)^2$.

```
ld = load('shakingDevice.mat');
accel = ld.sensorData.Acceleration;
gyro = ld.sensorData.AngularVelocity;
viewer = fusiondemo.OrientationViewer;

highVarFilt = imufilter('SampleRate', ld.Fs, ...
    'LinearAccelerationNoise', 0.009);
qHighLANoise = highVarFilt(accel, gyro);

lowVarFilt = imufilter('SampleRate', ld.Fs, ...
    'LinearAccelerationNoise', 0.0009);
qLowLANoise = lowVarFilt(accel, gyro);
```

One way to see the effect of the `LinearAccelerationNoise` is to look at the output gravity vector. The gravity vector is simply the 3rd column of the orientation rotation matrix.

```
rmatHigh = rotmat(qHighLANoise, 'frame');
rmatLow = rotmat(qLowLANoise, 'frame');

gravDistHigh = sqrt(sum( (rmatHigh(:,3,:) - [0;0;1]).^2, 1));
gravDistLow = sqrt(sum( (rmatLow(:,3,:) - [0;0;1]).^2, 1));

figure;
plot([squeeze(gravDistHigh), squeeze(gravDistLow)]);
title('Euclidean Distance to Gravity');
legend('LinearAccelerationNoise = 0.009', ...
    'LinearAccelerationNoise = 0.0009');
```



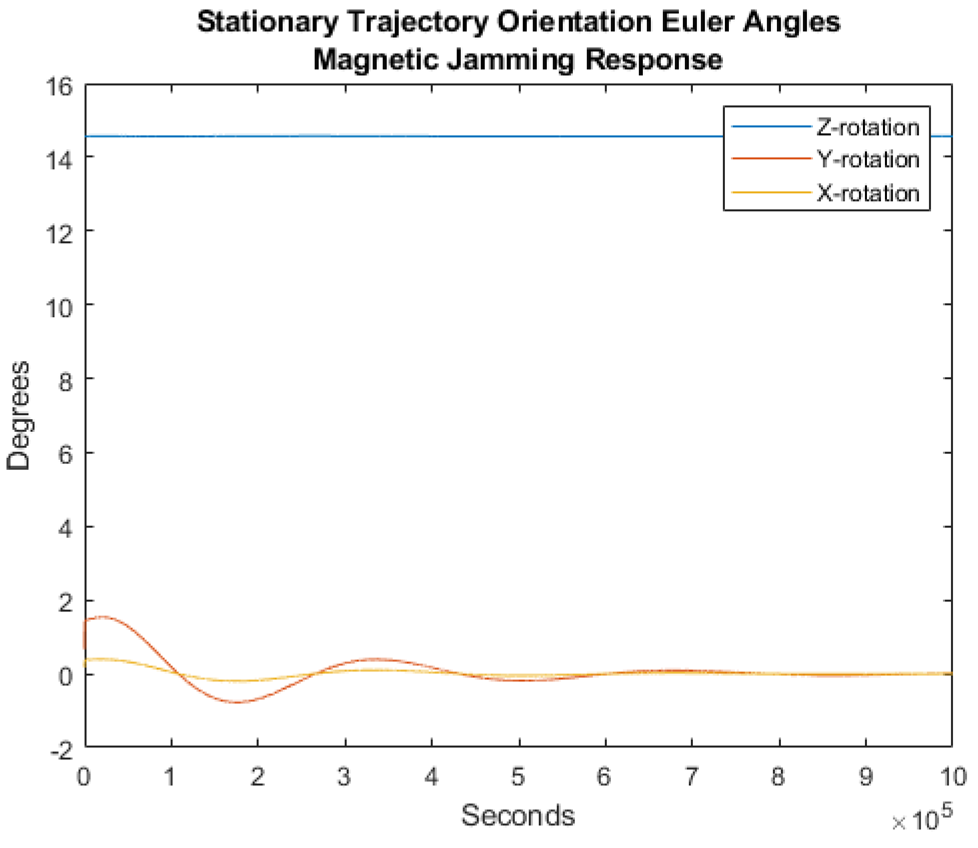
The `lowVarFilt` has a low `LinearAccelerationNoise` so it expects to be in an environment with low linear acceleration. Therefore, it is more susceptible to linear acceleration, as illustrated by the large variations earlier in the plot. However, because it expects to be in an environment with a low linear acceleration, higher trust is placed in the accelerometer signal. As such, the orientation estimate converges quickly back to vertical once the shaking has ended. The converse is true for `highVarFilt`. The filter is less affected by shaking but the orientation estimate takes longer to converge to vertical when the shaking has stopped.

The `MagneticDisturbanceNoise` property enables modeling magnetic disturbances (non-geomagnetic noise sources) in much the same way `LinearAccelerationNoise` models linear acceleration.

The two decay factor properties (`MagneticDisturbanceDecayFactor` and `LinearAccelerationDecayFactor`) model the rate of variation of the noises. For slowly varying noise sources, set these parameters to a value closer to 1. For quickly varying, uncorrelated noises, set these parameters closer to 0. A lower `LinearAccelerationDecayFactor` enables the orientation estimate to find "down" more quickly. A lower `MagneticDisturbanceDecayFactor` enables the orientation estimate to find North more quickly.

Very large, short magnetic disturbances are rejected almost entirely by the `ahrsfilter`. Consider a pulse of [0 250 0] uT applied while recording from a stationary sensor. Ideally, there should be no change in orientation estimate.

```
ld = load('magJamming.mat');
hpulse = ahrsfilter('SampleRate', ld.Fs);
len = 1:10000;
qpulse = hpulse(ld.sensorData.Acceleration(len,:), ...
    ld.sensorData.AngularVelocity(len,:), ...
    ld.sensorData.MagneticField(len,:));

figure;
timevec = 0:ld.Fs:(ld.Fs*numel(qpulse) - 1);
plot( timevec, eulerd(qpulse, 'ZYX', 'frame') );
title(['Stationary Trajectory Orientation Euler Angles' newline ...
    'Magnetic Jamming Response']);
legend('Z-rotation', 'Y-rotation', 'X-rotation');
ylabel('Degrees');
xlabel('Seconds');
```



Note that the filter almost totally rejects this magnetic pulse as interference. Any magnetic field strength greater than four times the `ExpectedMagneticFieldStrength` is considered a jamming source and the magnetometer signal is ignored for those samples.

Conclusion

The algorithms presented here, when properly tuned, enable estimation of orientation and are robust against environmental noise sources. It is important to consider the situations in which the sensors are used and tune the filters accordingly.

See Also

`ahrsfilter` | `ecompass` | `imuSensor` | `imufilter` | `insfilter`

More About

- “IMU and GPS Fusion for Inertial Navigation”
- “Estimate Position and Orientation of a Ground Vehicle”
- “Estimate Orientation and Height Using IMU, Magnetometer, and Altimeter”

4

Spatial Representation

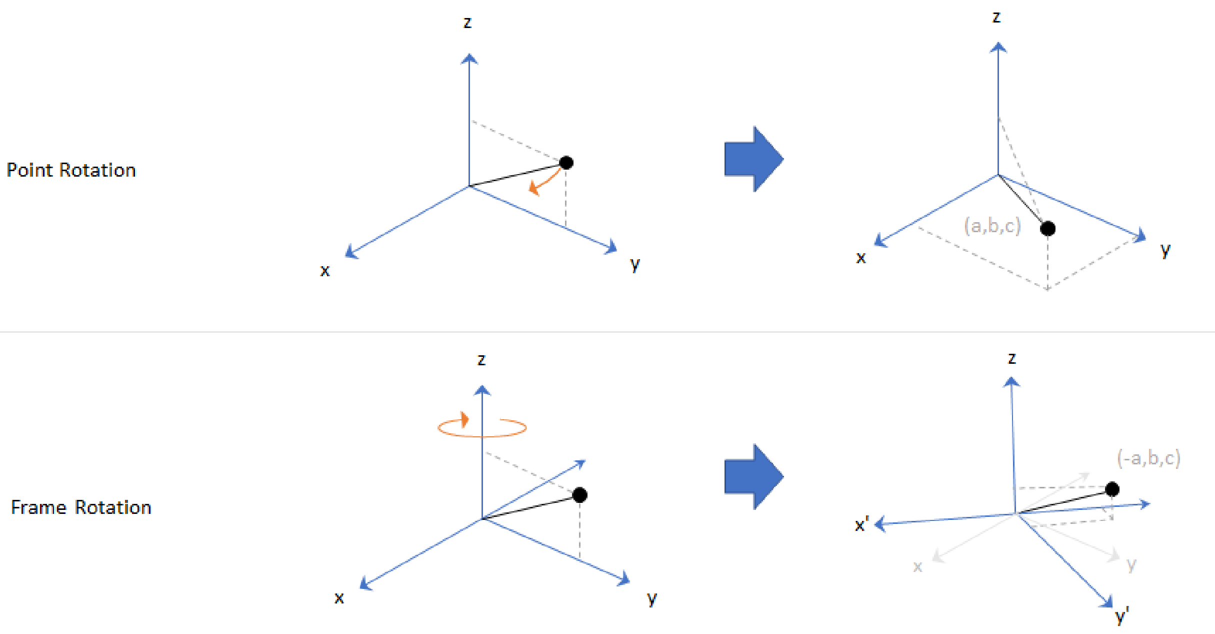
Orientation, Position, and Coordinate Systems

The Sensor Fusion and Tracking Toolbox enables you to track orientation, position, pose, and trajectory of a platform. A platform refers generally to any object you want to track.

Orientation

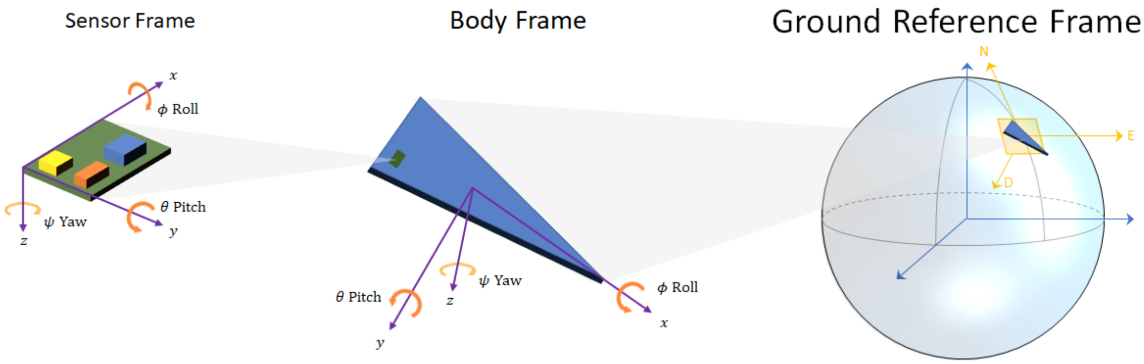
Orientation is defined by angular displacement. Orientation can be described in terms of point or frame rotation. In point rotation, the coordinate system is static and the point moves. In frame rotation, the point is static and the coordinate system moves. For a given axis and angle of rotation, point rotation and frame rotation define equivalent angular displacement but in opposite directions.

Sensor Fusion and Tracking Toolbox defaults to frame rotation.



Orientation is defined as the frame rotation that takes the parent frame to the child frame.

The choice of parent frame depends on the problem space. For example, manipulating sensor frames is necessary to align various axes of independent sensors. Tracking the body frame is often used for stabilization tasks. The ground reference frame is useful for tracking multiple independent platforms and locating platforms in an absolute sense.



Frame Rotation

To relate one orientation to another you must rotate a frame. The table summarizes the rotation conventions that Sensor Fusion and Tracking Toolbox uses. A three-axis coordinate is always specified in order [*x*,*y*,*z*].

Variable	Euler Angle	Symbol	Output Interval (Degrees)
z	Yaw	ψ	$-180 \leq \psi < 180$
y	Pitch	θ	$-90 \leq \theta \leq 90$
x	Roll	ϕ	$-180 \leq \phi < 180$

A positive rotation angle corresponds to a clockwise rotation about an axis when viewing from the origin along the positive direction of the axis. The right-hand convention is equivalent, where positive rotation is indicated by the direction in which the fingers on your right hand curl when your thumb is pointing in the direction of the axis of rotation.

To define three-dimensional frame rotation, you must rotate sequentially about the axes. Sensor Fusion and Tracking Toolbox uses intrinsic (carried frame) rotation, in which, after each rotation, the axis is updated before the next rotation. For example, to rotate an axis using the ZYX convention:

1 Rotate the parent frame about the *z*-axis to yield a new set of axes, (x',y',z), where the *x*- and *y*-axes have changed to x'- and y'-axes and the *z*-axis remains unchanged.

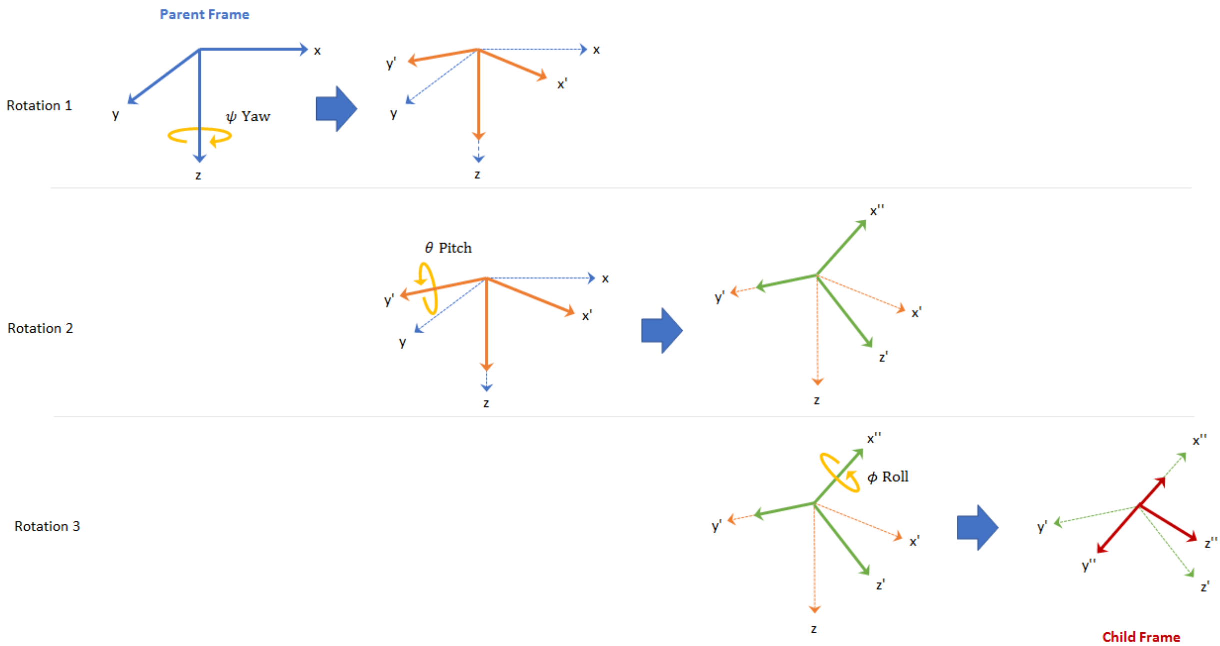
$$\begin{bmatrix} x' \\ y' \\ z \end{bmatrix} = R_z(\psi) \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

2 Rotate the new set of axes about the y'-axis, yielding another new set of axes, (x'',y',z').

$$\begin{bmatrix} x'' \\ y' \\ z' \end{bmatrix} = R_y(\theta) \begin{bmatrix} x' \\ y' \\ z \end{bmatrix}$$

3 Rotate this new set of axes about the x''-axis, arriving at the desired child frame, (x'',y'',z'').

$$\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = R_x(\phi) \begin{bmatrix} x'' \\ y' \\ z' \end{bmatrix}$$



This sequence of rotations follows the convention outlined in [1]. The rotation matrix required to convert a vector in the parent frame to a vector in the child frame for a given yaw, pitch, and roll is computed as:

$$\mathrm{R}\left(\psi,\ \theta,\ \phi\right) = \mathrm{R}_x\left(\psi\right)\mathrm{R}_y\left(\theta\right)\mathrm{R}_z\left(\phi\right) = \begin{bmatrix} \cos\psi\cos\theta & \\ \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi & \mathrm{s} \\ \cos\psi\sin\theta\cos\phi + \sin\psi\cos\phi & \mathrm{s} \end{bmatrix}$$

For features that support frame-based processing, Sensor Fusion and Tracking Toolbox provides coordinates as an *N*-by-3 matrix, where *N* is the number of samples in time and the three columns correspond to the *x*-, *y*-, and *z*-axes. The following calculation rotates a parent frame to a child frame:

$$a_{\mathrm{child}} = \left(\mathrm{R}\left(\psi,\ \theta,\ \phi\right) \times \left(a_{\mathrm{parent}}\right)'\right)'$$

Sensor Fusion and Tracking Toolbox enables efficient orientation computation using the `quaternion` data type. To create a rotation matrix using quaternions, use the `rotmat` function.

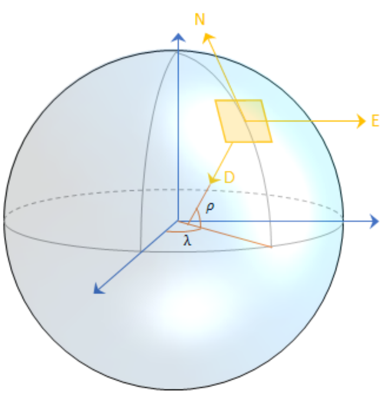
```
% Euler angles defining orientation of local axes
yaw = 20;
pitch = 5;
roll = 10;

% Create orientation matrix from Euler angles using quaternion class
q = quaternion([yaw pitch roll],'eulerd','zyx','frame');
myRotationMatrix = rotmat(q,'frame');
```

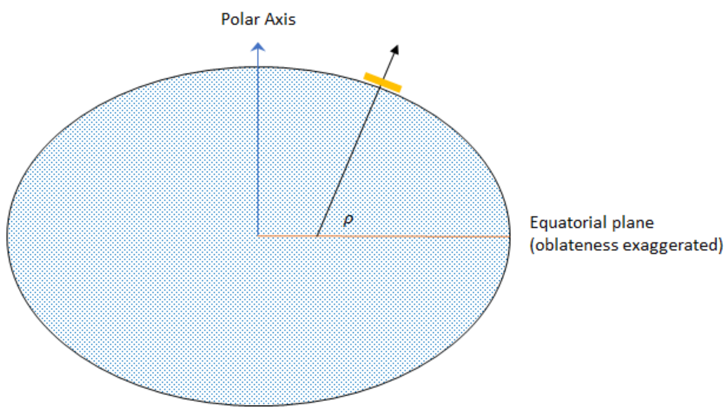
See “Rotations, Orientation and Quaternions” for more information on using quaternions in Sensor Fusion and Tracking Toolbox.

Position

Position is defined as the translational distance from a parent frame origin to a child frame origin. This toolbox uses the local north-east-down (NED) coordinate system as the parent frame. In the NED coordinate system:

- The origin is arbitrarily fixed to a point on the surface of the Earth. This makes the NED coordinate system local.
- The *x*-axis points toward the ellipsoid north.
- The *y*-axis points toward the ellipsoid east.
- The *z*-axis points downward along the ellipsoid normal (geodetic latitude, ρ).

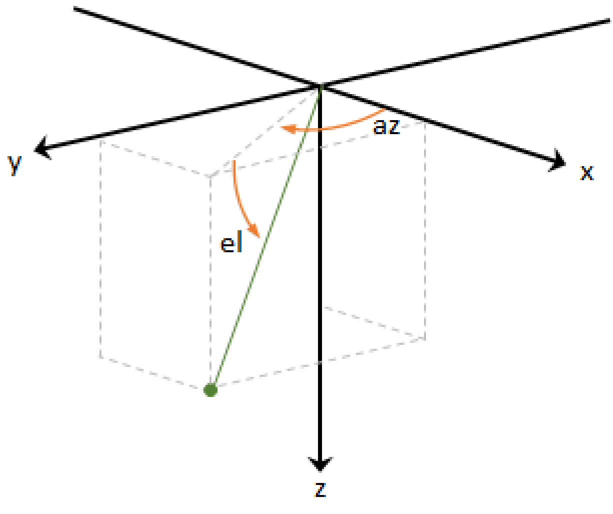




Azimuth and Elevation

Given a vector in $\mathbf{R}^3$:

- Azimuth is defined as the angle from the *x*-axis to the orthogonal projection of the vector onto the *xy*-plane. The angle is positive going from the *x*-axis toward the *y*-axis. Azimuth is given in degrees in the range [−180, 180).
- Elevation is defined as the angle from the projection onto the *xy*-plane to the vector. The angle is positive going from the *xy*-plane to the *z*-axis. Elevation is given in degrees in the range [−90, 90].



Pose

To specify an object in 3-D space fully, you can combine position and orientation. Pose is defined as the combination of position and orientation. Sensor Fusion and Tracking Toolbox uses the following conventions when describing pose.

Property/Field	Description	Units	Coordinate Frame
Position	Current position of platform in scenario	m	NED

Property/Field	Description	Units	Coordinate Frame
Velocity	Current velocity of platform in scenario	m/s	NED
Acceleration	Current acceleration of platform in scenario	m/s^2	NED
Orientation	Current orientation of platform in scenario	unit quaternion / orientation matrix	N/A
Angular velocity	Current angular velocity of platform in scenario	rad/s	NED

Trajectory

Trajectory defines how pose changes over time. To generate ground-truth trajectories in Sensor Fusion and Tracking Toolbox, use `kinematicTrajectory` or `waypointTrajectory`. To simulate tracking multiple platforms, use `trackingScenario`.

See Also

More About

- "Rotations, Orientation and Quaternions"

References

[1] IEEE. Standard for Distributed Interactive Simulation – Application Protocols. IEEE P1278.1/D16 Rev 18, May 2012.

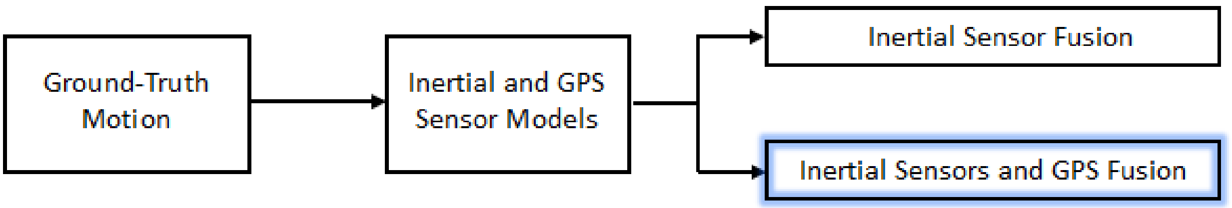
Pose

Determine Pose Using Inertial Sensors and GPS

Sensor Fusion and Tracking Toolbox enables you to fuse data read from IMUs and GPS to estimate pose. Use the `insfilter` function to create an INS/GPS fusion filter suited to your system:

- `MARGGPSFuser` -- Estimate pose using a magnetometer, gyroscope, accelerometer, and GPS data.
- `NHConstrainedIMUGPSFuser` -- Estimate pose using a gyroscope, accelerometer, and GPS data. This algorithm uses nonholonomic constraints.

This tutorial provides an overview of inertial sensor fusion with GPS in Sensor Fusion and Tracking Toolbox.



To learn how to model inertial sensors and GPS, see “Model IMU, GPS, and INS/GPS” on page 2-2. To learn how to generate the ground-truth motion that drives sensor models, see `waypointTrajectory` and `kinematicTrajectory`.

You can also fuse inertial sensor data without GPS to estimate orientation. See “Determine Orientation Using Inertial Sensors” on page 3-2.

Fuse Inertial Sensor and GPS data

An inertial navigation system (INS) consists of sensors that detect translational motion (accelerometers), rotation (gyroscopes), and in some systems magnetic fields (magnetometers). By fusing the sensor data continuously, an INS can dead reckon a platform's pose without external sensors. However, INS pose estimation generally decreases in accuracy over time and needs to be corrected using an external reference, such as GPS readings. Common configurations for INS/GPS fusion include MARG+GPS for aerial vehicles and accelerometer+gyroscope+GPS with nonholonomic constraints for ground vehicles.

Fuse MARG and GPS

A magnetic, angular rate, and gravity (MARG) system consists of a magnetometer, gyroscope, and accelerometer. To fuse MARG and GPS data, create a `MARGGPSFuser` object using the `insfilter` function:

```
FUSE = insfilter('NonholonomicHeading',false,'Magnetometer',true)
```

```
FUSE = 

  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
```

`MARGGPSFuser` uses an extended Kalman filter with the following methods:

- `predict` –– Update states using accelerometer and gyroscope data
- `fusemag` –– Correct states using magnetometer data
- `fusegps` –– Correct states using GPS data

Generally, accelerometer and gyroscope data is acquired at a higher rate than magnetometer and GPS data. You can use nested loops to call `predict`, `fusemag`, and `fusegps` at different rates.

```
accelData   = [0 0 9.8];
gyroData    = [0 0 0];
magData     = [19.535 -5.109 47.930];
magCov      = 0;
position    = [0 0 0];
positionCov = 0;
velocity    = rand(1,3);
```

```
velocityCov = 0;

predictDataSampleRate = 100;
fuseDataSampleRate = 2;
predictSamplesPerFuse = predictDataSampleRate/fuseDataSampleRate;

duration = 5;

for i = 1:duration*predictDataSampleRate

    for j = 1:predictSamplesPerFuse

        predict(FUSE,accelData,gyroData);

    end

    fusegps(FUSE,position,positionCov,velocity,velocityCov);
    fusemag(FUSE,magData,magCov);

end
```

At any time, you can call `pose` to return the current position and orientation estimates:

```
[position, orientation] = pose(FUSE)
```

```
position =

   1.0e-15 *

   -0.0000   -0.0555    0.1110


orientation =

  quaternion


     0.65342 + 0.56507i + 0.31113j + 0.39615k
```

For a complete example workflow using `MARGGPSFuser`, see “IMU and GPS Fusion for Inertial Navigation”.

Fuse Accelerometer, Gyroscope, and GPS with Nonholonomic Constraints

Fusing accelerometer, gyroscope, and GPS data with nonholonomic constraints is a common configuration for ground vehicle pose estimation. To fuse accelerometer, gyroscope, and GPS data, create a `NHConstrainedIMUGPSFuser` object using the `insfilter` function:

```
FUSE = insfilter('NonholonomicHeading',true,'Magnetometer',false)
```

```
FUSE = 

  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
```

`NHConstrainedIMUGPSFuser` uses an extended Kalman filter with the following functions:

- `predict` –– Update states using accelerometer and gyroscope data
- `fusegps` –– Correct states using GPS data

Generally, accelerometer and gyroscope data is acquired at a higher rate than GPS data. You can use nested loops to call `predict` and `fusegps` at different rates.

```
accelData   = [0 0 9.8];
gyroData    = [0 0 0];
position    = [0 0 0];
```

```
positionCov = 0;
velocity    = rand(1,3);
velocityCov = 0;

predictDataSampleRate = 100;
fuseDataSampleRate = 2;
predictSamplesPerFuse = predictDataSampleRate/fuseDataSampleRate;

duration = 5;


for i = 1:duration*predictDataSampleRate

    for j = 1:predictSamplesPerFuse

        predict(FUSE,accelData,gyroData);

    end

    fusegps(FUSE,position,positionCov,velocity,velocityCov);

end
```

At any time, you can call `pose` to return the current position and orientation estimates:

```
[position, orientation] = pose(FUSE)


position =

   1.0e-15 *

   -0.0000   -0.0555    0.1110


orientation =

  quaternion


     0.65342 + 0.56507i + 0.31113j + 0.39615k
```

For a complete example workflow using `NHConstrainedIMUGPSFuser`, see “Estimate Position and Orientation of a Ground Vehicle”.

See Also

`ahrsfilter` | `ecompass` | `imuSensor` | `imufilter` | `insfilter`

More About

- “IMU and GPS Fusion for Inertial Navigation”
- “Estimate Position and Orientation of a Ground Vehicle”
- “Estimate Orientation and Height Using IMU, Magnetometer, and Altimeter”

6

Tracking Scenario

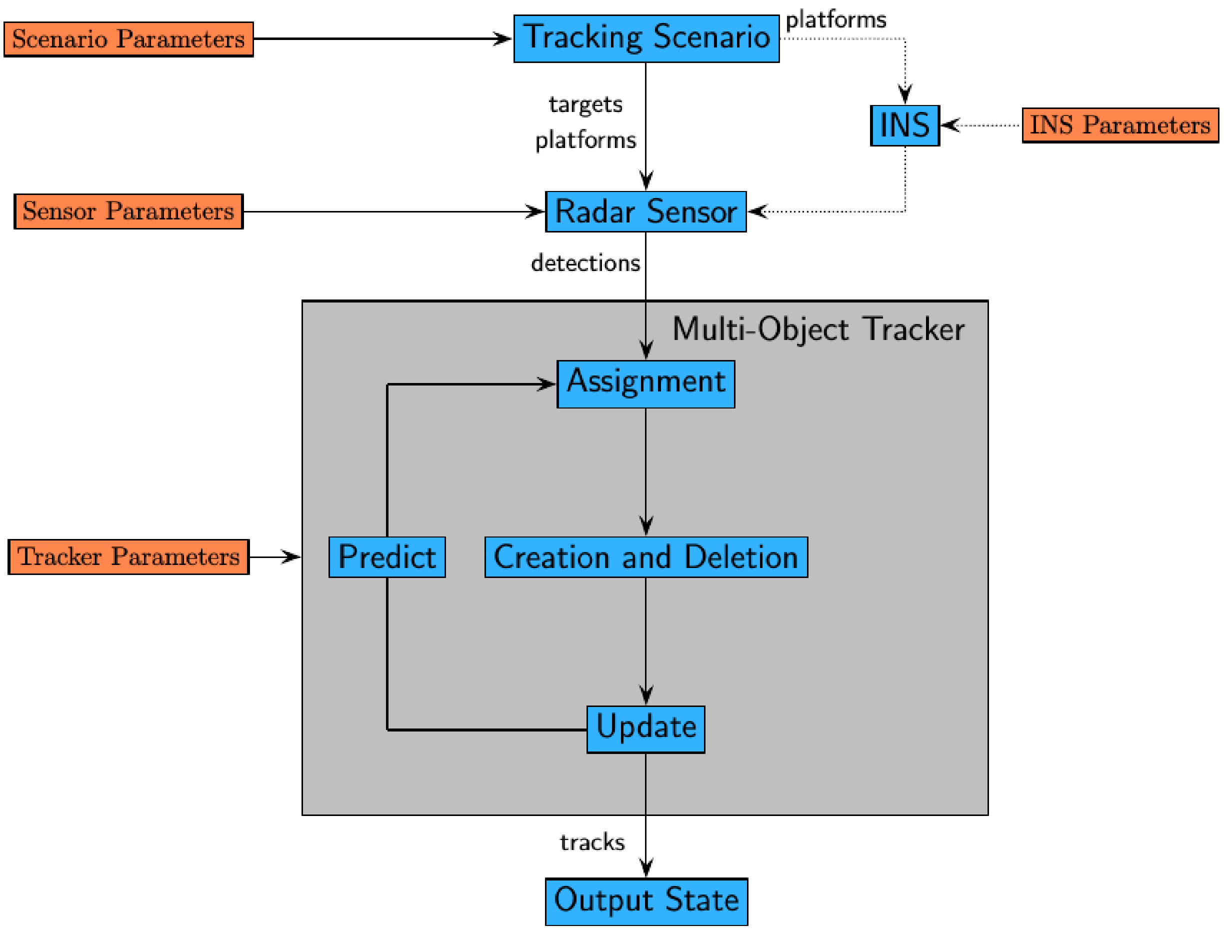
Tracking Simulation Overview

You can build a complete tracking simulation using the functions and objects supplied in this toolbox. The workflow for sensor fusion and tracking simulation consists of three (and optionally four) components. These components are

1. Use the tracking scenario generator to create ground truth for all moving and stationary radar platforms and all target platforms (planes, ships, cars, drones). The `trackingScenario` class models the motion of all platforms in a global coordinate system called scenario coordinates. These objects can represent ships, ground vehicles, airframes, or any object that the radar detects. See "Orientation, Position, and Coordinate Systems" on page 4-2 for a discussion of coordinate systems.
2. Optionally, simulate an inertial navigation system (INS) that provides radar sensor platform position, velocity, and orientation relative to scenario coordinates.
3. Create models for each radar sensor with specifications and parameters using the `monostaticRadarSensor`, `radarSensor`, or `radarEmitter` objects. Using target platform pose and profile information, generate synthetic radar detections for each radar-target combination. Methods belonging to `trackingScenario` retrieve the pose and profile of any target platform. The `trackingScenario` generator does not have knowledge of scenario coordinates. It knows the relative positions of the target platforms with respect to the body platform of the radar. Therefore, the detector can only generate detections relative to the radar location and orientation.

   If there is an INS attached to a radar platform, then the radar can transform detections to the scenario coordinate system. The INS allows multiple radars to report detections in a common coordinate system.
4. Process radar detections with a multi-object tracker to associate detections to existing tracks or create tracks. Multi-object tracks include `trackerGNN` and `trackerTOMHT`. If there is no INS, the tracker can only generate tracks specific to one radar. If an INS is present, the tracker can create tracks using measurements from all radars.

The flow diagram shows the progression of information in a tracking simulation.


Scenario Parameters
Tracking Scenario
platforms
targets
platforms
INS
INS Parameters
Sensor Parameters
Radar Sensor
detections
Multi-Object Tracker
Assignment
Tracker Parameters
Predict
Creation and Deletion
Update
tracks
Output State

Creating Tracking Scenario

You can define a tracking simulation by using the `trackingScenario` object. This object creates an empty scenario which you can then populate with platforms by calling the `platform` method as many times as needed. A platform is an object (moving or stationary) which can either be a sensor, a target, or other entity. After creating a platform, you can use the `platform` `path` method to create paths that specify how the platform moves. You construct paths by specifying three-dimensional waypoints and the arrival time of the platform at the waypoint. `path` creates a three-dimensional piecewise-clothoid curve that the platform follows. Then, run the simulation by calling the `advance` method in a loop, or by calling the record method to run the simulation all at once. You can set the simulation update interval using the `SampleTime` property.

When you create a platform, you set its properties or leave them to their default value. You can set them all except for `PlatformID`. The orientation of a platform can be specified in three equivalent ways. The `Roll`, `Pitch`, and `Yaw`, the `OrientationQuaternion`, and `OrientationMatrix` can all be used to specify the orientation of the platform. They must all be consistent. If you change one, the others are automatically changed. If you specify more than one when creating a platform, the last value entered is used. Platform properties and their default values are listed here.

Platform Properties

`PlatformID`	Scenario-defined platform ID.
`Position`	Position of platform, specified as a real-valued 1-by-3 vector representing the *(x,y,z)* coordinates of the platform referenced to the scenario coordinate system. The default is `[0,0,0]`. Units are in meters.
`Velocity`	Velocity of platform, specified as a real-valued 1-by-3 vector representing the *(vx,vy,vz)* velocity components referenced to the scenario coordinate system. The default is `[0,0,0]`. Units are in m/s.
`Roll`	Roll helps define the orientation of the platform body with respect to the scenario coordinate system. Roll is the clockwise rotation about the x body axes. If you do not enter a value for roll, roll has the default value of zero. You specify roll in degrees.
`Pitch`	Pitch helps define the orientation of the platform body with respect to the scenario coordinate system. Pitch is the clockwise rotation about the y body axes. If you do not enter a value for pitch, pitch has the default value of zero. You specify pitch in degrees.
`Yaw`	Yaw helps define the orientation of the platform body with respect to the scenario coordinate system. Yaw is the clockwise rotation about the z body axes. If you do not enter a value for yaw, yaw has the default value of zero. You specify yaw in degrees.
`OrientationQuaternion`	The orientation of the body expressed in quaternions.

`OrientationMatrix`	3-by-3 matrix specifying the platforms body coordinate vector with respect to scenario coordinates. The columns represent the directions of the carried x, y, and z coordinates of the body.
`AngularVelocity`	Angular velocity of platform, specified as a real-valued 1-by-3 vector specifying the x,y, and z components of the angular velocity referenced to scenario coordinates. Units are in degrees per second.
`RCSPattern`	The radar cross-section of a platform defines the response of the platform to radar signals. The values are specified at discrete azimuth and elevation angles. These angles are referenced to the platform body coordinates.
`RCSAzimuthAngles`	Azimuth angles of the entries in `RCSPattern`.
`RCSAzimuthAngles`	Elevation angles of the entries in `RCSPattern`.
`ClassID`	User-specified platform classification ID.

The input to the `path` method is the set of name-value parameters listed here.

Path Parameters

`Waypoints`	An *M*-by-3 matrix containing the x, y, and z positions in scenario coordinates through which the platform must pass. Units are in meters.
`TimeOfArrival`	An *M*-element vector that specifies the time at which the target crosses the corresponding waypoint. Units are in seconds.
`Velocities`	An *M*-by-3 matrix containing the x, y, and z components of the velocity in scenario coordinates at the corresponding waypoint. Units are in m/s.
`Groundspeed`	An *M*-element vector that specifies the ground speed at the corresponding waypoint. Units are in m/s.
`ClimbRate`	An *M*-element vector that specifies the rate of climb (in the z dimension). Units are in m/s.
`Course`	An *M*-element vector that specifies to the direction in which the target is moving in the horizontal plane. Course is specified as an angle measured from the *x*-axis to the *y*-axis in the *xy*-plane.
`Orientation`	An *M*-element vector of quaternions or a 3-by-3-by-*M* matrix of rotation matrices at each waypoint. These quantities correspond to the rotational orientation of the platform at that waypoint. If unspecified, the platform yaw and pitch angles are aligned in the direction of travel. Then, the `Banking` parameter roll determines the roll angle.

Banking	Specified as either 'flat' or 'frictionless'. • If 'flat', the platform roll angle is zero. • If 'frictionless', the roll angle of the platform is chosen to balance the accelerations due to the trajectory path and the acceleration due to gravity. The banking Banking property cannot be used together with Orientation. If you do not specify Banking or Orientation, the platform roll angle is zero.

At any time during the simulation, you can retrieve the current values of platform properties using the `platformPoses` and `platformProfiles` methods of the `trackingScenario` class. You can also use the `targetPoses` method of the Platform class. Both the `platformPoses` and `platformProfiles` methods return properties of all platforms with respect to scenario coordinates. The `targetPoses` method, while similar, returns properties of other platforms with respect to a specified platform.

Radar Detections

Simulate Radar Detections

The monostaticRadarSensor object simulates the detection of targets by a scanning radar. You can use the object to model many properties of real radar sensors. For example, you can

- simulate real detections with added random noise
- generate false alarms
- simulate mechanically scanned antennas and electronically scanned phased arrays
- specify angular, range, and range-rate resolution and limits

The radar sensor is assumed to be mounted on a platform and carried by the platform as it maneuvers. A platform can carry multiple sensors. When you create a sensor, you specify sensor positions and orientations with respect to the body coordinate system of a platform. Each call to monostaticRadarSensor creates a sensor. The output of monostaticRadarSensor generates the detection input to the multi-object tracker, gnnTracker, or any simple one-object tracker such as trackingKF, trackingEKF, trackingUKF, and trackingCKF.

The radar platform does not maintain any information about the radar sensors that are mounted on it. (The sensor itself contains its position and orientation with respect to the platform on which it is mounted but not which platform). You must create the association between radar sensors and platforms. A way to do this association is to put the platform and its associated sensors into a cell array. When you call a particular sensor, pass in the platform-centric target pose and target profile information. The sensor converts this information to sensor-centric poses. Target poses are outputs of trackingScenario methods.

Create Radar Sensor

You can create a radar sensor using the monostaticRadarSensor object. Set the radar properties using name-value pairs and then execute the simulator. For example,

```
radar1 = monostaticRadarSensor( ...
    'UpdateRate',updaterate, ...          % Hz
    'ReferenceRange', 111.0e3, ...        % m
    'ReferenceRCS', 0.0, ...              % dBsm
    'HasMechanicalScan',true, ...
    'MaxMechanicalScanRate',scanrate, ...  % deg/s
    'HasElectronicScan',false, ...
```

```
    'FieldOfView',fov, ...                  % [az;el] deg
    'HasElevation',false, ...
    'HasRangeRate',false, ...
    'AzimuthResolution',1.4, ...            % deg
    'RangeResolution', 135.0)                  % m
dets = radar1(targets,simtime);
```

Convenience Syntaxes

There are several syntaxes of `monostaticRadarSensor` that make it easier to specify the properties of commonly implemented radar scan modes. These syntaxes set combinations of these properties: `ScanMode`, `FieldOfView`, `MaxMechanicalScanRate`, `MechanicalScanLimits`, and `ElectronicScanLimits`.

- `sensor = monostaticRadarSensor('Rotator')` creates a `monostaticRadarSensor` object that mechanically scans 360° in azimuth. Setting `HasElevation` to `true` points the radar antenna towards the center of the elevation field of view.
- `sensor = monostaticRadarSensor('Sector')` creates a `monostaticRadarSensor` object that mechanically scans a 90° azimuth sector. 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.
- `sensor = monostaticRadarSensor('Raster')` returns a `monostaticRadarSensor` object that mechanically scans a raster pattern spanning 90° in azimuth and 10° in elevation upwards from the horizon. You can change the `ScanMode` property to `'Electronic'` to perform an electronic raster scan in the same volume.
- `sensor = monostaticRadarSensor('No scanning')` returns a `monostaticRadarSensor` object that stares along the radar antenna boresight direction. No mechanical or electronic scanning is performed.

You can set other radar properties when you use these syntaxes. For example,

```
sensor = monostaticRadarSensor('Raster','ScanMode','Electronic')
```

Radar Sensor Parameters

The properties specific to the `monostaticRadarSensor` object are listed here. For more detailed information, type

```
help monostaticRadarSensor
```

at the command line.

Sensor location parameters.

Sensor Location

SensorIndex	A unique identifier for each sensor.
UpdateRate	Rate at which sensor updates are generated, specified as a positive scalar. The reciprocal of this property must be an integer multiple of the simulation time interval. Updates requested between sensor update intervals do not return detections.
MountingLocation	Sensor (x,y,z) defining the offset of the sensor origin from the origin of its platform. The default value positions the sensor origin at the platform origin.
Yaw	Angle specifying the rotation around the platform z-axis to align the platform coordinate system with the sensor coordinate system. Positive yaw angles correspond to a clockwise rotation when looking along the positive direction of the z-axis of the platform coordinate system. Rotations are applied using the ZYX convention.
Pitch	Angle specifying the rotation around the platform y-axis to align the platform coordinate system with the sensor coordinate system. Positive pitch angles correspond to a clockwise rotation when looking along the positive direction of the y-axis of the platform coordinate system. Rotations are applied using the ZYX convention.

`Roll`	Angle specifying the rotation around the platform x-axis to align the platform coordinate system with the sensor coordinate system. Positive pitch angles correspond to a clockwise rotation when looking along the positive direction of the x-axis of the platform coordinate system. Rotations are applied using the ZYX convention.
`DetectionCoordinates`	Specifies the coordinate system for detections reported in the “Detections” output `struct`. The coordinate system can be one of: • `'Scenario'` –- detections are reported in the scenario coordinate frame in rectangular coordinates. This option can only be selected when the sensor `HasINS` property is set to `true`. • `'Body'` –- detections are reported in the body frame of the sensor platform in rectangular coordinates. • `'Sensor rectangular'` –- detections are reported in the radar sensor coordinate frame in rectangular coordinates aligned with the sensor frame axes. • `'Sensor spherical'` –- detections are reported in the radar sensor coordinate frame in spherical coordinates based on the sensor frame axes.

Sensitivity parameters.

Sensitivity Parameters

DetectionProbability	Probability of detecting a target with radar cross section, ReferenceRCS, at the range of ReferenceRange.
FalseAlarmRate	The probability of a false detection within each resolution cell of the radar. Resolution cells are determined from the AzimuthResolution and RangeResolution properties and when enabled the ElevationResolution and RangeRateResolution properties.
ReferenceRange	Range at which a target with radar cross section, ReferenceRCS, is detected with the probability specified in DetectionProbability.
ReferenceRCS	The target radar cross section (RCS) in dB at which the target is detected at the range specified by ReferenceRange with a detection probability specified by DetectionProbability.

Sensor resolution and bias parameters.

Resolution Parameters

`AzimuthResolution`	The radar azimuthal resolution defines the minimum separation in azimuth angle at which the radar can distinguish two targets.
`ElevationResolution`	The radar elevation resolution defines the minimum separation in elevation angle at which the radar can distinguish two targets. This property only applies when the `HasElevation` property is set to `true`.
`RangeResolution`	The radar range resolution defines the minimum separation in range at which the radar can distinguish two targets.
`RangeRateResolution`	The radar range rate resolution defines the minimum separation in range rate at which the radar can distinguish two targets. This property only applies when the `HasRangeRate` property is set to `true`.
`AzimuthBiasFraction`	This property defines the azimuthal bias component of the radar as a fraction of the radar azimuthal resolution specified by the `AzimuthResolution` property. This property sets a lower bound on the azimuthal accuracy of the radar.
`ElevationBiasFraction`	This property defines the elevation bias component of the radar as a fraction of the radar elevation resolution specified by the `ElevationResolution` property. This property sets a lower bound on the elevation accuracy of the radar. This property only applies when the `HasElevation` property is set to `true`.

RangeBiasFraction	This property defines the range bias component of the radar as a fraction of the radar range resolution specified by the RangeResolution property. This property sets a lower bound on the range accuracy of the radar.
RangeRateBiasFraction	This property defines the range rate bias component of the radar as a fraction of the radar range resolution specified by the RangeRateResolution property. This property sets a lower bound on the range rate accuracy of the radar. This property only applies when you set the HasRangeRate property to true.

Enabling parameters.

Enabling Parameters

HasElevation	This property allows the radar sensor to scan in elevation and estimate elevation from target detections.
HasRangeRate	This property allows the radar sensor to estimate range rate.
HasFalseAlarms	This property allows the radar sensor to generate false alarm detection reports.
HasRangeAmbiguities	When true, the radar does not resolve range ambiguities. When a radar sensor cannot resolve range ambiguities, targets at ranges beyond the MaxUnambiguousRange property value are wrapped into the interval [0 MaxUnambiguousRange]. When false, targets are reported at their unwrapped range.
HasRangeRateAmbiguites	When true, the radar does not resolve range rate ambiguities. When a radar sensor cannot resolve range rate ambiguities, targets at range rates above the MaxUnambiguousRadialSpeed property value are wrapped into the interval [0 MaxUnambiguousRadialSpeed]. When false, targets are reported at their unwrapped range rates. This property only applies when the HasRangeRate property is set to true.

`HasNoise`	Specifies if noise is added to the sensor measurements. Set this property to `true` to report measurements with noise. Set this property to `false` to report measurements without noise. The reported measurement noise covariance matrix contained in the output `objectDetection` struct is always computed regardless of the setting of this property.
`HasINS`	Set this property to true to enable an optional input argument to pass the current estimate of the sensor platform pose to the sensor. This pose information is added to the `MeasurementParameters` field of the reported detections. Then, the tracking and fusion algorithms can estimate the state of the target detections in scenario coordinates.

Scan parameters.

Scan Parameters

`ScanMode`	This property specifies the scan mode used by the radar as one of: • `'No scanning'` -- the radar does not scan. The radar beam points along the antenna boresight. • `'Mechanical'`-- the radar mechanically scans between the azimuth and elevation limits specified by the `MechanicalScanLimits` property. • `'Electronic'`-- the radar electronically scans between the azimuth and elevation limits specified by the `ElectronicScanLimits` property. • `'Mechanical and electronic'` -- the radar mechanically scans the antenna boresight between the mechanical scan limits and electronically scans beams relative to the antenna boresight between the electronic scan limits. The total field of regard scanned in this mode is the combination of the mechanical and electronic scan limits. In all scan modes except `'No scanning'`, the scan proceeds at angular intervals specified by the radar field of view specified in `FieldOfView`.

`MaxMechanicalScanRate`	This property sets the magnitude of the maximum mechanical scan rate of the radar. When `HasElevation` is `true`, the scan rate is a vector consisting of separate azimuthal and elevation scan rates. When `HasElevation` is `false`, the scan rate is a scalar representing the azimuthal scan rate. The radar sets its scan rate to step the radar mechanical angle by the radar field of regard. When the required scan rate exceeds the maximum scan rate, the maximum scan rate is used.
`MechanicalScanLimits`	This property specifies the mechanical scan limits of the radar with respect to its mounted orientation. When `HasElevation` is `true`, the limits are specified by minimum and maximum azimuth and by minimum and maximum elevation. When `HasElevation` is `false`, limits are specified by minimum and maximum azimuth. Azimuthal scan limits cannot span more than 360 degrees and elevation scan limits must lie in the closed interval `[-90 90]`.
`ElectronicScanLimits`	This property specifies the electronic scan limits of the radar with respect to the current mechanical angle. When `HasElevation` is `true`, the limits are specified by minimum and maximum azimuth and by minimum and maximum elevation. When `HasElevation` is `false`, limits are specified by minimum and maximum azimuth. Both azimuthal and elevation scan limits must lie in the closed interval `[-90 90]`.

`FieldOfView`	This property specifies the sensor azimuthal and elevation fields of view. The field of view defines the total angular extent observed by the sensor during a sensor update. The field of view must lie in the interval *(0,180]*. Targets outside of the sensor angular field of view during a sensor update are not detected.

Range and range rate parameters.

Range and Range Rate Parameters

MaxUnambiguousRange	This property specifies the range at which the radar can unambiguously resolve the range of a target. Targets detected at ranges beyond the unambiguous range are wrapped into the range interval [0 MaxUnambiguousRange]. This property only applies to true target detections when you set HasRangeAmbiguities property to true. This property also defines the maximum range at which false alarms are generated. This property only applies to false target detections when you set HasFalseAlarms property to true.
MaxUnambiguousRadialSpeed	This property specifies the maximum magnitude value of the radial speed at which the radar can unambiguously resolve the range rate of a target. Targets detected at range rates whose magnitude is greater than the maximum unambiguous radial speed are wrapped into the range rate interval [-MaxUnambiguousRadialSpeed MaxUnambiguousRadialSpeed]. This property only applies to true target detections when you set both the HasRangeRate and HasRangeRateAmbiguities properties to true. This property also defines the range rate interval over which false target detections are generated. This property only applies to false target detections when you set both the HasFalseAlarms and HasRangeRate properties to true.

Detector Input

Each sensor created by `monostaticRadarSensor` accepts as input an array of target structures. This structure serves as the interface between the `trackingScenario` and the sensors. You create the target `struct` from target poses and profile information produced by `trackingScenario` or equivalent software.

The 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 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)`.

Field	Description
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].

You can create a target pose structure by merging information from the platform information output from the targetProfiles method of trackingScenario and target pose information output from the targetPoses method on the platform carrying the sensors. You can merge them by extracting for each PlatformID in the target poses array, the profile information in platform profiles array for the same PlatformID.

The platform targetPoses method returns this structure for each target other than the platform.

Target Poses

platformID
ClassID
Position
Velocity
Yaw
Pitch
Roll
AngularVelocity

The platformProfiles method returns this structure for all platforms in the scenario.

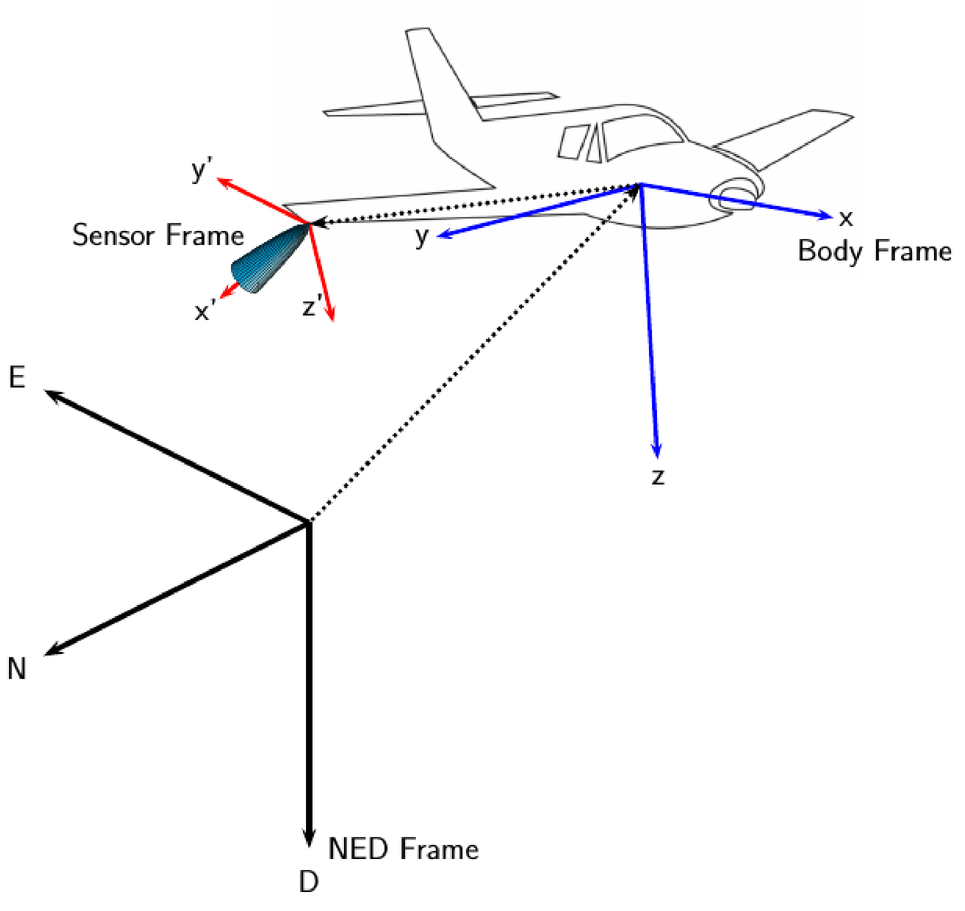
Platform Profiles

`PlatformID`
`ClassID`
`RCSPattern`
`RCSAzimuthAngles`
`RCSElevationAngles`

Radar Sensor Coordinate Systems

Detections consist of measurements of positions and velocities of targets and their covariance matrices. Detections are constructed with respect to sensor coordinates but can be output in one of several coordinates. Multiple coordinate frames are used to represent the positions and orientations of the various platforms and sensors in a scenario.

In a radar simulation, there is always a top-level global coordinate system which is usually the North-East-Down (NED) Cartesian coordinate system defined by a tangent plane at any point on the surface of the Earth. The `trackingScenario` object models the motion of platforms in the global coordinate system. When you create a platform, you specify its location and orientation relative to the global frame. These quantities define the body axes of the platform. Each radar sensor is mounted on the body of a platform. When you create a sensor, you specify its location and orientation with respect to the platform body coordinates. These quantities define the sensor axes. The body and radar axes can change over time, however, global axes do not change.



Additional coordinate frames can be required. For example, often tracks are not maintained in NED (or ENU) coordinates, as this coordinate frame changes based on the latitude and longitude where it is defined. For scenarios that cover large areas (over 100 kilometers in each dimension), earth-centered earth-fixed (ECEF) can be a more appropriate global frame to use.

A radar sensor generates measurements in spherical coordinates relative to its sensor frame. However, the locations of the objects in the radar scenario are maintained in a top-level frame. A radar sensor is mounted on a platform and will, by default, only be aware of its position and orientation relative to the platform on which it is mounted. In other words, the radar expects all target objects to be reported relative to the platform body axes. The radar reports the required transformations (position and orientation) to relate the reported detections to the platform body axes. These transformations are used by consumers of the radar detections (e.g. trackers) to maintain tracks in the platform body axes. Maintaining tracks in the platform body axes enables the fusion of measurement or track information across multiple sensors mounted on the same platform.

If the platform is equipped with an inertial navigation system (INS) sensor, then the location and orientation of the platform relative to the top-level frame can be determined. This INS information can be used by the radar to reference all detections to scenario coordinates.

INS

When you specify `HasINS` as true, you must pass in an INS `struct` into the `step` method. This structure consists of the position, velocity, and orientation of the platform in scenario coordinates. These parameters let you express target poses in scenario coordinates by setting the `DetectionCoordinates` property.

Detections

Radar sensor detections are returned as a cell array of `objectDetection` objects. A detection contains these properties.

objectDetection Structure

Field	Definition
Time	Measurement time
Measurement	Measurements
MeasurementNoise	Measurement noise covariance matrix
SensorIndex	Unique ID of the sensor
ObjectClassID	Object classification
MeasurementParameters	Parameters used by initialization functions of any nonlinear Kalman tracking filters
ObjectAttributes	Additional information passed to tracker

Measurement and MeasurementNoise are reported in the coordinate system specified by the DetectionCoordinates property of the monostaticRadarSensor are reported in sensor Cartesian coordinates.

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>

The `MeasurementParameters` field consists of an array of `structs` describing a sequence of coordinate transformations from a child frame to a parent frame or the inverse transformations (see “Frame Rotation” on page 4-3). 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.

Each `struct` takes the form:

MeasurementParameters

Parameter	Definition
`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 frame(k) from the origin of frame(k+1) represented as a 3-by-1 vector.
`OriginVelocity`	Velocity offset of the origin of frame(k) from the origin of frame(k+1) represented as a 3-by-1 vector.
`Orientation`	A 3-by-3 real-valued orthonormal frame rotation matrix which rotates the axes of frame(k+1) into alignment with the axes of frame(k).
`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's coordinate frame to the parent's coordinate frame.
`HasElevation`	A logical scalar indicating if the frame has three-dimensional position. Only set to false for the first `struct` when detections are reported in spherical coordinates and `HasElevation` is `false`, otherwise it is `true`.
`HasVelocity`	A logical scalar indicating if the reported detections include velocity measurements. `true` when `HasRangeRate` is enabled, otherwise `false`.

ObjectAttributes

Attribute	Definition
`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.

8

Multi-Object Tracking

Tracking is the process of estimating the state of motion of an object based on measurements taken off the object. For an object moving in space, the state usually consists of position, velocity, and any other state parameters of objects at any given time. A state is the necessary information needed to predict future states of the system given the specified equations of motion. The estimates are derived from observations on the objects and are updated as new observations are taken. Observations are made using one or more sensors. Observations can only be used to update a track if it is likely that the observation is that of the object having that track. Observations need to be either associated with an existing track or used to create a new track. When several tracks are present, there are several ways observations are associated with one and only one track. The chosen track is based on the "closest" track to the observation.

Tracking and Tracking Filters

Multi-Object Tracking

You can use the multi-sensor, multi-target trackers, `trackerGNN`, and `trackerTOMHT`, to track many targets simultaneously. Tracks are initiated and updated using sensor detections of targets. There are several steps in the execution of the tracker when new detections are made.

- The tracker tries to assign a detection to an existing track.
- The tracker creates a track for each detection it cannot assign. When starting the tracker, all detections are used to create tracks.
- The tracker evaluates the status of each track. For new tracks, the status is judged to be tentative until enough detections are made to confirm the track. For existing tracks, newly assigned detections are used by the tracking filter to update the track state. When a track has no new added detections, it is held in a coasted state until new detections are assigned to it. If no new detections are added after a specified number of updates, the track is deleted.

on the command line.

When tracking multiple objects, there are several things to consider

- Decide which type of tracking filter to use. The choice of tracking filter depends on the expected dynamics of the object you want to track. The toolbox provides five Kalman filters for this purpose: the Linear Kalman filter, `trackingKF`, the Extended Kalman filter, `trackingEKF`, the Unscented Kalman filter, `trackingUKF`, the Cubature Kalman filter, `trackingCKF`, and the Interacting Multiple Model (IMM) Kalman filter, `trackingIMM`. The linear Kalman filter is used when the dynamics of the object follow a linear model and the measurements are linear functions of the state vector. The extended, unscented, and cubature Kalman filters are used when the dynamics are nonlinear or the measurement model is nonlinear or both.

  The toolbox provides multi-object trackers, `trackerGNN`. and `trackerTOMHT`, which run the tracking filters and manage tracks. You can set the type of filter using an initialization function such as `initcvkf` which creates a constant-velocity linear Kalman filter from a single detection report. For any nonlinear Kalman filter, the initialization lets you specify a state transition function and a measurement function. For the extended Kalman filter, you can specify an optional state transition function

Jacobian and an optional measurement function Jacobian. For example, `initcaekf` creates a constant-acceleration extended Kalman filter.

- Choose which track assignment function to use. The assignment function determines whether a new detection belongs to an existing track or not. The toolbox contains three assignment algorithms, all of which use a cost matrix. Each column is assigned to a row in a way that minimizes the total cost. The algorithms are described in the help for each of these functions.

  - `assignmunkres` uses a Munkres assignment algorithm to find an optimal solution to the global nearest neighbor (GNN) assignment problem.
  - `assignauction` uses a forward/reverse auction assignment algorithm to find a suboptimal solution to the GNN assignment problem.
  - `assignjv` uses a Jonker-Volgenant assignment algorithm to find another type of optimal solution to the GNN assignment problem.
  - Other assignment algorithms include `assignkbest`, `assignkbestsd`, `assignsd`, and `assignTOMHT`.
- For the `trackerTOMHT` tracker, you can specify the conditions under which a track is confirmed or deleted by setting the `TrackLogic` property. Two algorithms are supported: `'History'` and `'Score'`.

  - `'History'` -- track confirmation and deletion are based on the number of times the track has been assigned to a detection in the last several tracker updates.
  - `'Score'` -- track confirmation and deletion are based on a log-likelihood computation. 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 false.

Multi-Object Tracker Properties

trackerGNN Properties

The `trackerGNN` object is a multi-sensor, multi-object tracker that uses global nearest neighbor association. Each detection can be assigned to only one track (single-hypothesis tracker) which can also be a new track that the detection initiates. At each step of the simulation, the tracker updates the track state. You can specify the behavior of the tracker by setting the following properties.

trackerGNN Properties

FilterInitializationFcn	A handle to a function that initializes a tracking filter based on a single detection. This function is called when a detection cannot be assigned to an existing track. For example, initcaekf creates an extended Kalman filter for an accelerating target. All tracks are initialized with the same type of filter.
Assignment	The name of the assignment algorithm. The tracker provides three built-in algorithms: 'Munkres', 'Jonker-Volgenant', and 'Auction' algorithms. You can also create your own custom assignment algorithm by specifying 'Custom'.
CustomAssignmentFcn	The name of the custom assignment algorithm function. This property is available on when the Assignment property is set to 'Custom'.
AssignmentThreshold	Specify the threshold that controls the assignment of a detection to a track. Detections can only be assigned to a track if their normalized distance from the track is less than the assignment threshold. Each tracking filter has a different method of computing the normalized distance. Increase the threshold if there are detections that can be assigned to tracks but are not. Decrease the threshold if there are detections that are erroneously assigned to tracks.

`TrackLogic`	Specify the track confirmation logic --`'History'` or `'Score'`. For descriptions of these options, type
	
	`help trackHistoryLogic`
	
	or
	
	`help trackScoreLogic`
	
	at the command line.

`ConfirmationThreshold`	Specify the threshold for track confirmation. The threshold depends on the setting for `TrackLogic` • `'History'` -- specify the confirmation threshold as `[M N]`. If the track is detected at least `M` times in the last `N` updates, the track is confirmed. • `'Score'` --- specify the confirmation threshold as a single number. If the score is greater than or equal to the threshold, this track is confirmed. .

DeletionThreshold	Specify the threshold for track deletion. The threshold depends on the setting of TrackLogic • 'History' -- specify the deletion threshold as a pair of integers [P R]. A track is deleted if it is not assigned to a track at least P times in the last R updates. • 'Score' --- specify the deletion threshold as a single number. The track is deleted if its score decreases by at least this threshold from its maximum track score. .
DetectionProbability	Specify the probability of detection as a number in the range *(0,1)*. The probability of detection is used to calculate the track score when initializing and updating a track. This property is used only when TrackLogic is set to 'Score'.
FalseAlarmRate	Specify the rate of false detection as a number in the range *(0,1)*. The false alarm rate is used to calculate the track score when initializing and updating a track. This property is used only when TrackLogic is set to 'Score'.
Beta	Specify the rate of new tracks per unit volume as a positive number. This property is used only when TrackLogic is set to 'Score'. The rate of new tracks is used in calculating the track score during track initialization. This property is used only when TrackLogic is set to 'Score'.

`Volume`	Specify the volume of the sensor measurement bin as a positive scalar. For example, a radar sensor that produces a 4-D measurement of azimuth, elevation, range, and range-rate creates a 4-D volume. The volume is a product of the radar angular beamwidth, the range bin width, and the range-rate bin width. The volume is used in calculating the track score when initializing and updating a track. This property is used only when `TrackLogic` is set to `'Score'`.
`MaxNumTracks`	Specify the maximum number of tracks the tracker can maintain.
`MaxNumSensors`	Specify the maximum number of sensors sending detections to the tracker as a positive integer. This number must be greater than or equal to the largest `SensorIndex` value used in the `objectDetection` input to the `step` method. This property determines how many sets of `ObjectAttributes` each track can have.
`HasDetectableTrackIDsInput`	Set this property to `true` if you want to provide a list of detectable track IDs as input to the `step` method. This list contains all tracks that the sensors expect to detect and, optionally, the probability of detection for each track ID.
`HasCostMatrixInput`	Set this property to `true` if you want to provide an assignment cost matrix as input to the `step` method.

trackerGNN Input

The input to the `trackerGNN` consists of a list of detections, the update time, cost matrix, and other data. Detections are specified as a cell array of `objectDetection` objects (see “Detections”). The input arguments are listed here.

trackerGNN Input

`tracker`	A `trackerGNN` object.
`detections`	Cell array of `objectDetection` objects (see "Detections").
`time`	Time to which all the tracks are to be updated and predicted. The time at this execution step must be greater than the value in the previous call.
`costmatrix`	Cost matrix for assigning detections to tracks. A real *T*-by-*D* matrix, where *T* is the number of tracks listed in the `allTracks` argument returned from the previous call to `step`. *D* is the number of detections that are input in the current call. A larger cost matrix entry means a lower likelihood of assignment.
`detectableTrackIDs`	IDs of tracks that the sensors expect to detect, specified as an *M*-by-1 or *M*-by-2 matrix. The first column consists of track IDs, as reported in the `TrackID` field of the tracker output. The second column is optional and allows you to add the detection probability for each track.

trackerGNN Output

The output of the tracker can consist of up to three `struct` arrays with track state information. You can retrieve just the confirmed tracks, the confirmed and tentative tracks, or these tracks plus a combined list of all tracks.

```
confirmedTracks = step(...)
```

```
[confirmedTracks, tentativeTracks] = step(...)
```

```
[confirmedTracks, tentativeTracks, allTracks] = step(...)
```

The fields contained in the `struct` are:

trackerGNN Output `struct`

`TrackID`	Unique integer that identifies the track.
`UpdateTime`	Time to which the track is updated.
`Age`	Number of updates since track initialization.
`State`	State vector at update time.
`StateCovariance`	State covariance matrix at update time.
`IsConfirmed`	True if the track is confirmed.
`TrackLogic`	The track logic used in confirming the track – `'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 larger of *N* specified in the confirmation threshold property, `ConfirmationThreshold`, and *R* specified in the deletion threshold property, `DeletionThreshold`. • For `'Score'` track logic, a 1-by-2 numerical array in the form: `[currentScore, maxScore]`.
`IsCoasted`	True if the track has been updated without a detection. In this case, tracks are predicted to the current time.
`ObjectClassID`	An integer value representing the target classification. Zero is reserved for an "unknown" class.
`ObjectAttributes`	A cell array of cells. Each cell captures the object attributes reported by the corresponding sensor.