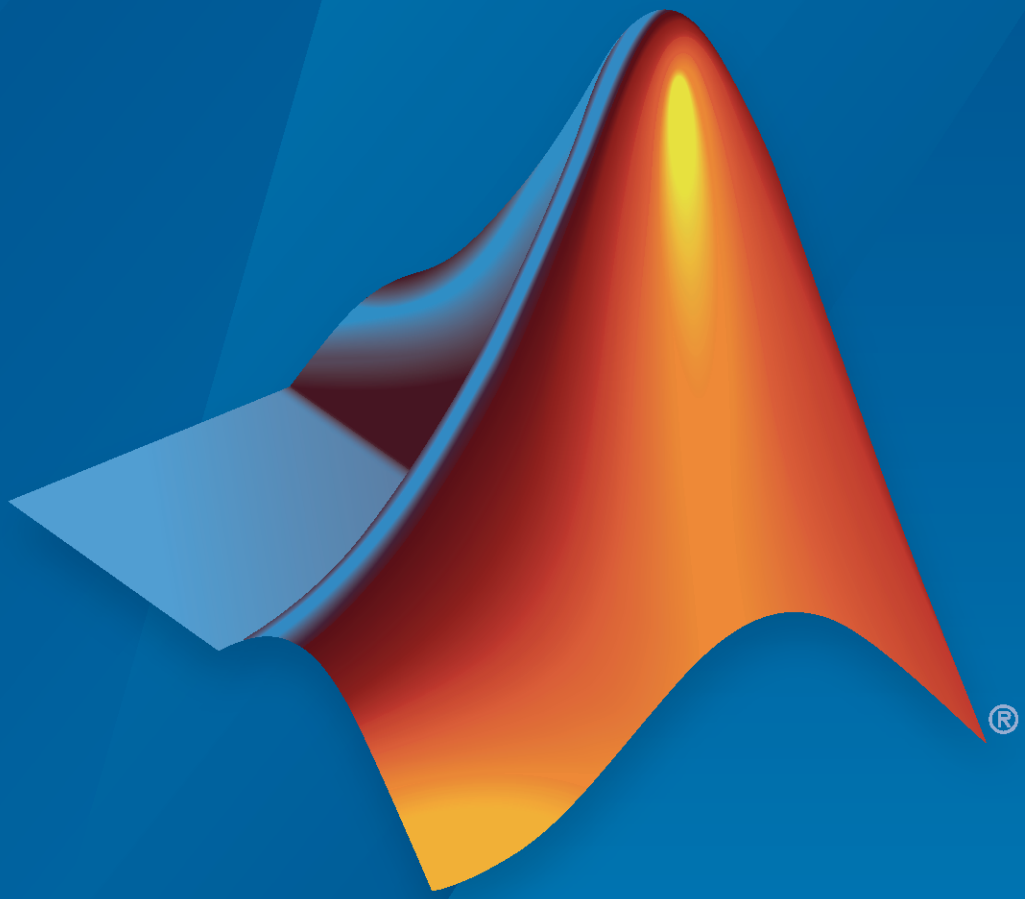
Sensor Fusion and Tracking Toolbox™ Release Notes



MATLAB®

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™ Release Notes

Contents

R2021b

R2021a

R2020b

R2020a

R2019b

R2019a

R2021b

Version: 2.2

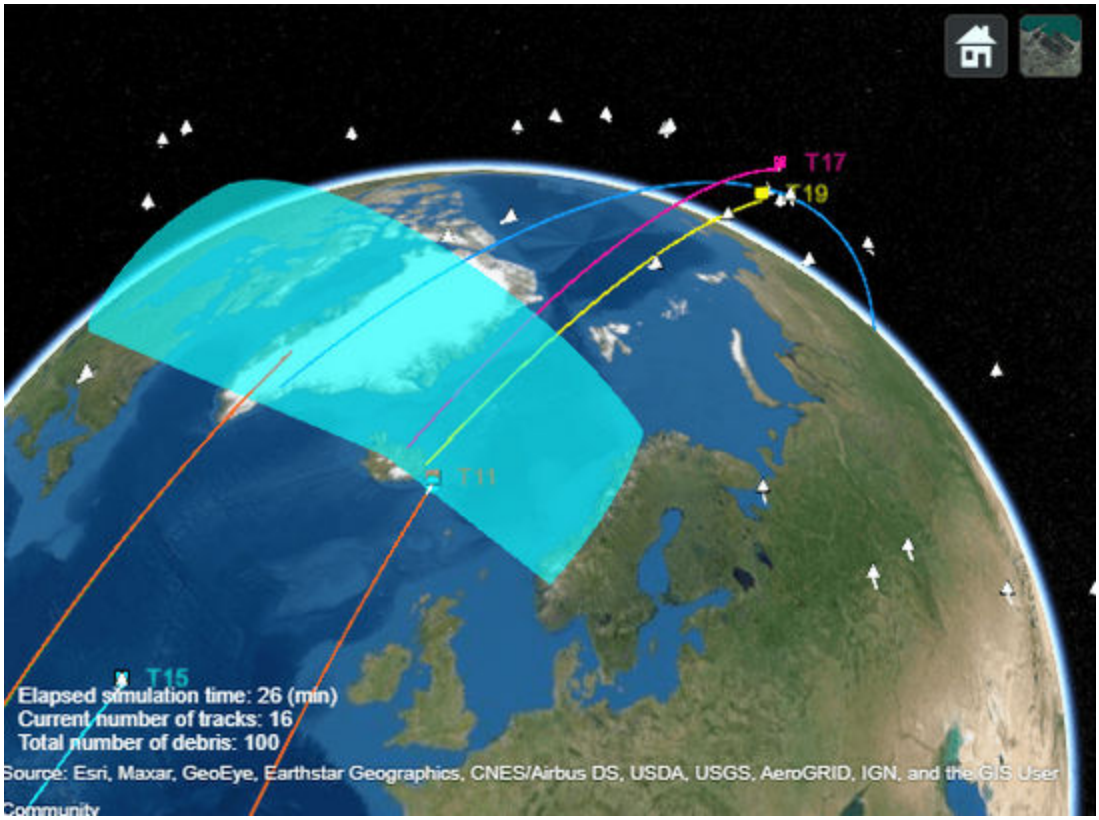
New Features

Bug Fixes

Visualize tracking scenario in virtual globe using trackingGlobeViewer

Use the `trackingGlobeViewer` object to create a virtual globe for tracking scenario visualization. On the globe, you can plot platforms, trajectories, sensor coverages, detections, and tracks. You can specify target trajectories and platform positions using latitude, longitude, and altitude (LLA) coordinates in the global Earth-centered-Earth-fixed (ECEF) frame or using Cartesian coordinates in the local north-east-down (NED) or east-north-up (ENU) frames.

See “Track Space Debris Using a Keplerian Motion Model” for an example of using the `trackingGlobeViewer` object.



Handle out-of-sequence measurement (OOSM) using retrodiction

You can use the retrodiction algorithm, provided in the `trackingKF` and `trackingEKF` filter objects, to process the OOSM and improve state estimate accuracy. To enable retrodiction in the filter, specify the `MaxNumOOSMSteps` property as a positive integer. After updating the filter regularly, by calling the `predict` and `correct` object functions, you can use the `retrodict` and `retroCorrect` object functions to incorporate the OOSM into the state estimate.

You can also enable the retrodiction capability in the `trackerGNN` System object by first specifying its `OOSMHandling` property as `"Retrodiction"`, and then specifying its `MaxNumOOSMSteps` property as a positive integer. For details on how the tracker processes an OOSM, see the description of the OOSMHandling property.

Similarly, you can enable the retrodiction capability in the Global Nearest Neighbor Multi Object Tracker Simulink block by specifying its **Out-of-sequence measurements handling** parameter as

`Retrodiction` and specifying its **Maximum number of OOSM steps** parameter as a positive integer.

For more details, see these topics and examples:

- “Introduction to Out-of-Sequence Measurement Handling”
- “Handle Out-of-Sequence Measurements with Filter Retrodiction”
- “Handle Out-of-Sequence Measurements in Multisensor Tracking Systems”
- “Event-Based Sensor Fusion and Tracking with Retrodiction”

Import tracking scenario using Tracking Scenario Reader Simulink block

Use the Tracking Scenario Reader Simulink block to import a `trackingScenario` object or a **Tracking Scenario Designer** app session file into Simulink. By default, the block outputs the information of platforms in the scenario and the simulation time. Optionally, you can configure the block to output detections, point clouds, emissions, sensor and emitter configurations, and sensor coverages.

For more details, see the “Track Point Targets in Dense Clutter Using GM-PHD Tracker in Simulink” example.

Track objects using Grid-Based Multi Object Tracker Simulink block

Use the Grid-Based Multi Object Tracker Simulink block to track objects using a grid-based occupancy evidence approach. By specifying the **Enable dynamic grid map visualization** parameter, you can choose to visualize the dynamic evidential grid map maintained by the block.

For more details, see the “Grid-based Tracking in Urban Environments Using Multiple Lidars in Simulink” example.

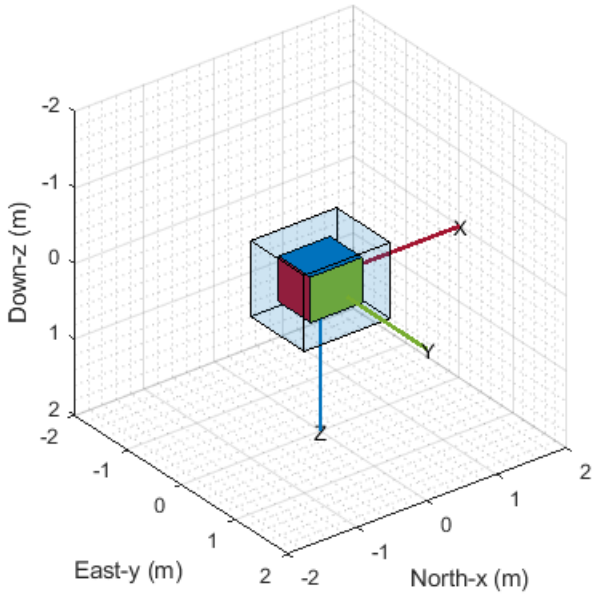
Model and Simulate GPS sensor using GPS Simulink block

Use the GPS Simulink block to simulate a GPS sensor and generate noise-corrupted GPS measurements based on position and velocity inputs.

Visualize rigid body position and orientation using poseplot

Use the `poseplot` function to visualize the 3-D platform pose (position and orientation). To customize the appearance of the pose plot, see PosePatch Properties.

See “Estimate Orientation Through Inertial Sensor Fusion” for an example of using the `poseplot` function.



INS Simulink block provides more parameters to specify its characteristics

The INS Simulink block provides six new parameters for modeling an inertial navigation system sensor:

- **Mounting location** — Location of sensor on platform
- **Use acceleration and angular velocity** — Enable both input and output ports of **Acceleration** and **AngularVelocity**
- **Acceleration accuracy** — Standard deviation of acceleration noise
- **Angular velocity accuracy** — Standard deviation of angular velocity noise
- **Enable HasGNSSFix port** — Enable **HasGNSSFix** input port
- **Position error factor** — Drift rate of position without GNSS fix

Perturb imuSensor properties

You can now use the `perturbations` function to define perturbations on the `imuSensor` System object for its accelerometer, gyroscope, and magnetometer components. You can then use the `perturb` function to apply the perturbations defined on the `imuSensor` object.

Perturb object properties using truncated normal distribution

You can now define the perturbation distribution of a property as a truncated normal distribution using the `perturbations` function. With offset values bounded by a finite interval, the truncated normal distribution is suitable for perturbing a property whose valid values are confined in a finite interval.

Partition detections using DBSCAN algorithm

Using the `partitionDetections` function, you can now partition detections using the density-based spatial clustering of applications with noise (DBSCAN) algorithm in addition to the preexisting distance-based partitioning algorithm. To use the DBSCAN algorithm, specify the `Algorithm` name-value argument of the function as `"DBSCAN"`.

Merge detections into clustered detections using mergeDetections

Use the `mergeDetections` function to merge detections that share the same cluster index into clustered detections.

Generate more memory-efficient C/C++ code from trackers and tracking filters

These objects and Simulink blocks now support strict single-precision code generation and non-dynamic memory allocation code generation:

- `trackerGNN`
- `trackerJPDA`
- `trackingEKF`
- `trackingUKF`
- `trackingCKF`
- `trackingIMM`
- Global Nearest Neighbor Multi Object Tracker
- Joint Probabilistic Data Association Multi Object Tracker

For details, see “Generate Code with Strict Single-Precision and Non-Dynamic Memory Allocation from Sensor Fusion and Tracking Toolbox”. You can also refer to the **Extended Capabilities** section on each object or block reference page for its code generation limitations.

New examples

The following new examples are available:

- “Lidar and Radar Fusion in an Urban Air Mobility Scenario”
- “Object Tracking and Motion Planning Using Frenet Reference Path”
- “Extended Object Tracking of Highway Vehicles with Radar and Camera in Simulink”
- “Extended Target Tracking with Multipath Radar Reflections in Simulink”
- “Grid-based Tracking in Urban Environments Using Multiple Lidars in Simulink”
- “Handle Out-of-Sequence Measurements with Filter Retrodiction”
- “Handle Out-of-Sequence Measurements in Multisensor Tracking Systems”
- “Smooth Trajectory Estimation of trackingIMM Filter”
- “Event-Based Sensor Fusion and Tracking with Retrodiction”

- “Reconstruct Ground Truth Trajectory from Sampled Data Using Filtering, Smoothing, and Interpolation”

R2021a

Design and evaluate tracking systems in Simulink

Track objects using PHD tracker in Simulink

Use the Probability Hypothesis Density (PHD) Tracker block in Simulink® to track point targets and extended objects.

For more details, see the Track Point Targets in Dense Clutter Using GM-PHD Tracker in Simulink example.

Perform track-level fusion in Simulink

Use the Track-To-Track Fuser block in Simulink to fuse tracks generated from trackers and tracking sensors.

For more details, see the Track-Level Fusion of Radar and Lidar Data in Simulink example.

Calculate tracking metrics in Simulink

Use the Optimal Subpattern Assignment Metric block in Simulink to calculate the optimal subpattern assignment metric (OSPA) between tracks and truths. The metric is comprised of the localization error, cardinality error, and labeling error components.

Use the Generalized Optimal Subpattern Assignment Metric block in Simulink to calculate the generalized optimal subpattern assignment (GOSPA) metric between tracks and truths. The metric is comprised of the switching error, localization error, missed target error, and false track error components.

For more details, see the Track-Level Fusion of Radar and Lidar Data in Simulink example.

Concatenate detections in Simulink

Use the Detection Concatenation block in Simulink to concatenate detection buses originating from multiple sources into a single bus of detections that can be passed to a tracker block.

Concatenate tracks in Simulink

Use the Track Concatenation block in Simulink to concatenate track buses originating from multiple sources into a single bus of tracks that can be passed to a Track-To-Track Fuser block.

Construct tracking architectures and simulate tracking systems

Use the `trackingArchitecture` System object™ to systematically model the architecture of a tracking system that consists of trackers and track fusers. You can simulate the constructed tracking system using the created `trackingArchitecture` object.

For more details, see the Define and Test Tracking Architectures for System-of-Systems example.

Smooth tracking filters

Use the `smooth` function to backward smooth tracking filters including `trackingABF`, `trackingKF`, `trackingEKF`, `trackingUKF`, `trackingMSCEKF`, `trackingCKF`, and `trackingIMM` objects.

To enable the `smooth` function on a filter, set the `EnableSmoothing` property of the filter to `true`, and optionally set the value for the `MaxNumSmoothingSteps` property.

Model Automatic Dependent Surveillance-Broadcast (ADS-B) transponder and receiver

Use the `adsbTransponder` System object to model an ADS-B transponder and generate ADS-B messages.

Use the `adsbReceiver` System object to receive ADS-B messages and generate tracks.

Comprehensive support for tuning inertial sensor filters

You can tune the parameters of these inertial sensor filter objects using the associated `tune` object function:

- `insfilterNonholonomic`
- `ahrs10filter`
- `insfilterMARG`
- `insfilterErrorState`

Previously, the `imufilter`, `ahrsfilter`, and `insfilterAsync` objects had associated `tune` functions.

The `tunerconfig` object, which configures the tuning process, has three new properties:

- `Filter` — Class name of the fusion filter.
- `FunctionTolerance` — Minimum change in cost to continue tuning.
- `OutputFcn` — Output function to show tuning results. For example, you can use the `tunerPlotPose` function to visualize the truth data and state estimates after tuning.

Generate synthetic radar detections using fusionRadarSensor

Use the `fusionRadarSensor` System object to statistically model a radar sensor and generate synthetic data. This object enables you to model the radar detection mode as monostatic, bistatic, or electronic support measure (ESM). You can generate detections, clustered detections, and tracks using the object.

For more details, see these examples:

- Introduction to Tracking Scenario and Simulating Sensor Detections
- Multiplatform Radar Detection Fusion
- Extended Object Tracking With Radar For Marine Surveillance

Compatibility Considerations

This System object replaces the `monostaticRadarSensor` and `radarSensor`, unless you require C/C++ code generation. Currently, the `fusionRadarSensor` System object does not support C/C++

code generation. For more details, see “radarSensor and monostaticRadarSensor System objects are not recommended” on page 2-6.

Support for K-best joint events in trackerJPDA

The `trackerJPDA` System object now supports K-best joint probability data association events in detection and track association, which calculate the K most probable events instead of exhausting all possible events. This new option can reduce computation and memory costs when you use `trackerJPDA`, especially for tracking closely spaced objects.

To enable K-best joint events, specify the `MaxNumEvents` property of `trackerJPDA` as a positive integer.

Access dynamicEvidentialGridMap from trackerGridRFS

You can choose to output a `dynamicEvidentialGridMap` object when running a `trackerGridRFS` System object. Using the object functions of `dynamicEvidentialGridMap`, you can directly access and monitor the grid map maintained in the tracker, which enables you to tune the tracker more efficiently and predict the motion of the target using the grid map.

You can also use the `predictMapToTime` object function of the `trackerGridRFS` object to predict the dynamic evidential map maintained in the tracker.

For more details, see the Motion Planning in Urban Environments Using Dynamic Occupancy Grid Map example.

Allow out-of-sequence measurements (OOSM) in trackers and corresponding Simulink blocks

You can choose to neglect out-of-sequence measurements (OOSM) when using the trackers and Simulink blocks listed in this table. The table also describes how to enable or disable this option.

Tracker System Objects or Simulink Blocks	OOSM Management
`trackerGNN`	Set the `OOSMHandling` property of the tracker to one of these options: • `'Terminate'` — The tracker stops running when it encounters an out-of-sequence measurement. • `'Neglect'` — The tracker ignores any out-of-sequence measurements and continues to run.
`trackerJPDA`	
`trackerTOMHT`	
Global Nearest Neighbor Multi Object Tracker	Set the **Out-of-sequence measurements handling** parameter of the block to one of these options: • `Terminate` — The block stops running when it encounters an out-of-sequence measurement. • `Neglect` — The block ignores any out-of-sequence measurements and continues to run.
Joint Probabilistic Data Association Multi Object Tracker	
Track-Oriented Multi-Hypothesis Tracker	

Transform between geodetic coordinates and local Cartesian coordinates

Use these functions to transform between geodetic coordinates and local north-east-down (NED) or east-north-up (ENU) coordinates.

- `enu2lla` — Transform local ENU coordinates to geodetic coordinates.
- `ned2lla` — Transform local NED coordinates to geodetic coordinates.
- `lla2enu` — Transform geodetic coordinates to local ENU coordinates.
- `lla2ned` — Transform geodetic coordinates to local NED coordinates.

Use geodetic coordinates as inputs for gpsSensor

You can use geodetic coordinates as inputs for a `gpsSensor` System object. To enable this option, specify the `PositionInputFormat` property of the `gpsSensor` object as `'Geodetic'`.

insSensor provides more properties to specify its characteristics

The `insSensor` System object provides six new properties to model an inertial navigation system sensor:

- `MountingLocation` — Location of sensor on platform
- `AccelerationAccuracy` — Standard deviation of acceleration noise
- `AngularVelocityAccuracy` — Standard deviation of angular velocity noise
- `TimeInput` — Enable or disable input of simulation time
- `HasGNSSFix` — Enable or disable GNSS fix
- `PositionErrorFactor` — Drift rate of position without GNSS fix

trackingScenario provides new properties to control and monitor scenario simulation

You can use the new `InitialAdvance` property of the `trackingScenario` object to specify the initial timestamp when running the scenario simulation. Specify one of the following values:

- `'Zero'` — The scenario simulation starts at time `0` in the first call to the `advance` object function.
- `'UpdateInterval'` — The scenario simulation starts at time $1/F$, where F is the value of a nonzero `UpdateRate` property. If you specify the `UpdateRate` property as `0`, the scenario ignores the `InitialAdvance` property and starts at time `0`.

You can use the new read-only property `SimulationStatus` to monitor the status of the scenario simulation. In the course of a scenario simulation, the `SimulationStatus` property changes from `NotStarted` to `InProgress` to `Completed`.

Compatibility Considerations

Using the `IsRunning` property to monitor simulation status is no longer recommended. Use the `SimulationStatus` property instead.

Variable-sized input support for timescope object

The `timescope` object allows you to visualize scalar or variable-sized input signals. If the signal is variable-sized, the number of channels (columns) must remain constant.

New examples

The following new examples are available:

- Custom Tuning of Fusion Filters
- Define and Test Tracking Architectures for System-of-Systems
- Detect and Track LEO Satellite Constellation with Ground Radars
- Adaptive Tracking of Maneuvering Targets with Managed Radar
- Track Point Targets in Dense Clutter Using GM-PHD Tracker in Simulink
- Track-Level Fusion of Radar and Lidar Data in Simulink
- Highway Vehicle Tracking with Multipath Radar Reflections
- Motion Planning in Urban Environments Using Dynamic Occupancy Grid Map
- Use `theaterPlot` to Visualize Tracking Scenario
- Model Platform Motion Using Trajectory Objects

Functionality being removed or changed

radarSensor and monostaticRadarSensor System objects are not recommended
Still runs

The `radarSensor` and `monostaticRadarSensor` System objects are not recommended, unless you require C/C++ code generation. Instead, use the `fusionRadarSensor` System object. The `fusionRadarSensor` object provides additional properties for modeling radar sensors, including the ability to generate tracks and clustered detections. Currently, `fusionRadarSensor` doe not support code generation.

There are no current plans to remove the `radarSensor` and `monostaticRadarSensor` System objects. MATLAB® code that use these features will continue to run. In addition to the new `fusionRadarSensor` object, you can still import a scenario containing `monostaticRadarSensor` objects into the **Tracking Scenario Designer** app. Also, when you export a scenario to MATLAB code, the app exports the sensors as `fusionRadarSensor` objects.

IsRunning property of trackingScenario object is not recommended
Still runs

The `IsRunning` property of the `trackingScenario` object is not recommended. Instead, use the `SimulationStatus` property. There are no current plans to remove the `IsRunning` property.

R2020b

Version: 2.0

New Features

Bug Fixes

Create Earth-centered waypoint trajectory

Use the `geoTrajectory` System object to create an Earth-centered waypoint trajectory. To define a `geoTrajectory` object in a tracking scenario, set the value of the `IsEarthCentered` property of the `trackingScenario` object to `true`.

For more details, see the Simulate and Track En-Route Aircraft in Earth-Centered Scenarios example.

Track objects using grid-based RFS tracker

Use the grid-based random finite set (RFS) tracker, the `trackerGridRFS` System object, to track objects using a grid-based occupancy evidence approach. To visualize the dynamic occupancy grid map of the tracking scene, use the `showDynamicMap` function.

For more details, see the Grid-based Tracking in Urban Environments Using Multiple Lidars example.

Generate synthetic point cloud data using simulated lidar sensor

Use the `monostaticLidarSensor` System object to model a lidar sensor and generate synthetic point cloud data from platforms in a tracking scenario. The `monostaticLidarSensor` System object obtains data from mesh representations of platforms within the scenario.

- To define a mesh representation other than the default cuboid mesh for a platform, use the `extendedObjectMesh` object. The toolbox also provides a predefined mesh object, `tracking.scenario.airplaneMesh`.
- To obtain the meshes of platforms in a tracking scenario, use the `targetMeshes` function.
- To obtain point cloud data from all the `monostaticLidarSensor` objects mounted on a platform, use the `lidarDetect` function of the platform. To obtain point cloud data from all the `monostaticLidarSensor` objects in a tracking scenario, use the `lidarDetect` function of the scenario.

For more details, see the Extended Object Tracking with Lidar for Airport Ground Surveillance example.

Improve inertial sensor fusion performance using filter tuner

Use the `tune` function and the `tunerconfig` object to adjust properties of the `imufilter`, `ahrsfilter`, and `insfilterAsync` objects for performance improvement.

For more details, see the Automatic Tuning of the insfilterAsync Filter example.

Perturb tracking scenarios, sensors, and trajectories for Monte Carlo simulation

Use the `perturbations` and `perturb` functions to perturb tracking scenarios, sensors, and trajectories. You can run Monte Carlo simulations on the perturbed objects using the `monteCarloRun` function.

For more details, see the Simulate, Detect, and Track Anomalies in a Landing Approach example.

Import trackingScenario object into Tracking Scenario Designer app

Import a `trackingScenario` object into the **Tracking Scenario Designer** app to visualize and redesign the imported tracking scenario. For the limitations of importing a `trackingScenario` object, see the Programmatic Use section.

Model and simulate INS sensor in Simulink

Use the INS block to model and simulate an INS sensor and generate INS measurements in Simulink.

Model and simulate Singer acceleration motion

Use the `singer` function to model and simulate Singer acceleration motion. You can also use the `initsingerekf`, `singerjac`, `singermeas`, `singermeasjac`, and `singerProcessNoise` functions to construct a Singer model-based extended Kalman filter as a `trackingEKF` object.

For more details, see the Track Multiple Lane Boundaries with a Global Nearest Neighbor Tracker example.

Time Scope object: Bilevel measurements, triggers, and compiler support

The `timescope` object now includes support for:

- Bilevel measurements – Measure transitions, overshoots, undershoots, and cycles.
- Triggers – Set triggers to sync repeating signals and pause the display when events occur.
- MATLAB Compiler™ support – Use the `mcc` function to compile code for deployment.

New examples

This release contains several new examples:

- Simulate and Track En-Route Aircraft in Earth-Centered Scenarios
- Grid-based Tracking in Urban Environments Using Multiple Lidars
- Extended Object Tracking with Lidar for Airport Ground Surveillance
- Detect, Classify, and Track Vehicles Using Lidar
- Simulate, Detect, and Track Anomalies in a Landing Approach
- Track Multiple Lane Boundaries with a Global Nearest Neighbor Tracker
- Automatic Tuning of the insfilterAsync Filter
- Generate Code for a Track Fuser with Heterogeneous Source Tracks

R2020a

Version: 1.3

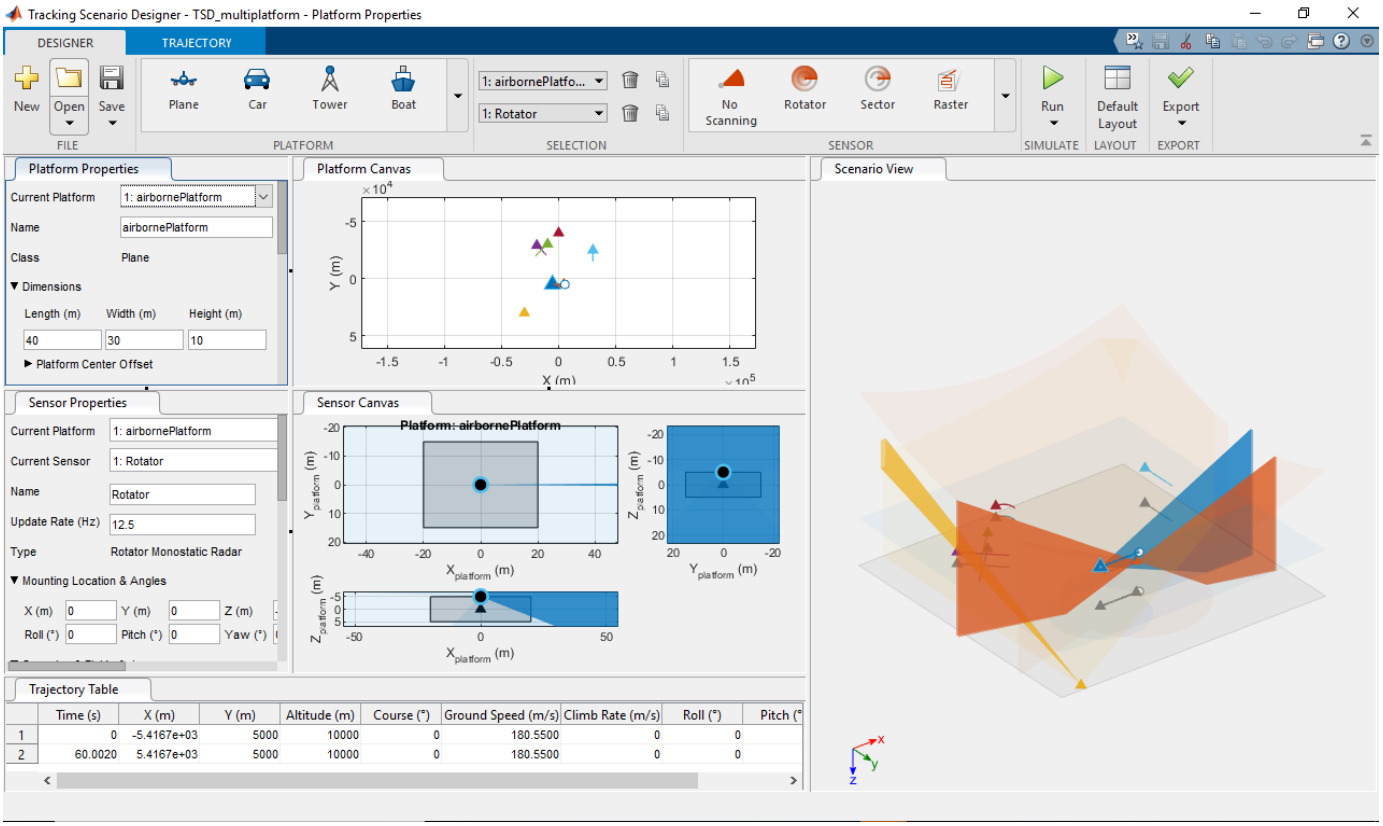
New Features

Bug Fixes

Tracking Scenario Designer App: Interactively design tracking scenarios

Use the Tracking Scenario Designer app to interactively design a tracking scenario composed of platforms, sensors, and trajectories. You can also output scripts for the designed tracking scenario.

For more details, see the Design and Simulate Tracking Scenario with Tracking Scenario Designer example.



Design and run Monte Carlo simulations

Use `trackingScenario`, `trackingScenarioRecording`, and `monteCarloRun` to design and run Monte Carlo simulations for tracking applications.

Visualize sensor coverage area

Use `coveragePlotter` and `plotCoverage` along with `theaterPlot` to plot and visualize beam and coverage area of sensors in a tracking scenario.

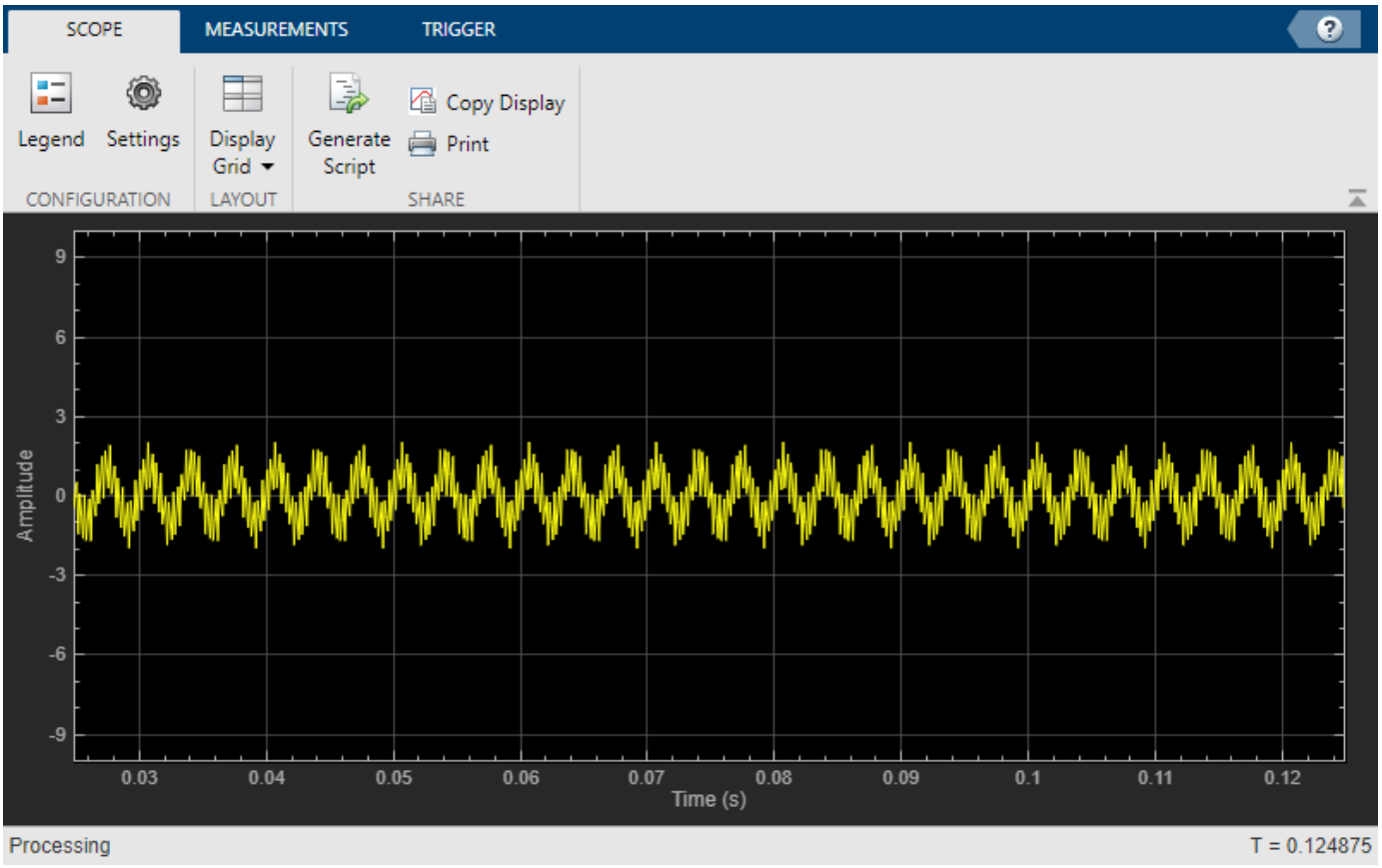
New time scope object: Visualize signals in the time domain

Use the `timescope` object to visualize real- and complex-valued floating-point and fixed-point signals in the time domain.

The Time Scope window has two toolstrip tabs:

Scopes Tab

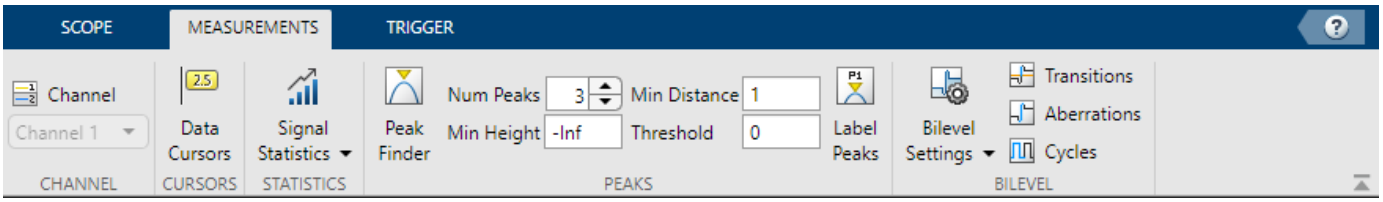
On the **Scopes** tab, you can control the layout and configuration settings, and set the display settings of the Time Scope. You can also generate script to recreate your time scope with the same settings. When doing so, an editor window opens with the code required to recreate your `timescope` object.



Measurements Tab

In the **Measurements** tab, all measurements are made for a specified channel.

- **Data Cursors** — Display the screen cursors.
- **Signal Statistics** — Display the various statistics of the selected signal, such as maximum/minimum values, peak-to-peak values, mean, median, and RMS.
- **Peak Finder** — Display peak values for the selected signal.



Evaluate tracking performance using GOSPA metric

Use `trackGOSPAMetric` to evaluate the performance of a tracking system against truths based on the global optimal subpattern assignment metric.

For more details, see the Introduction to Tracking Metrics example.

Collect emissions and detections from platforms in tracking scenario

Use `emit`, `propagate`, and `detect` to collect emissions and detections from platforms in a `trackingScenario`.

Access residuals and residual covariance of insfilters and ahrs10filter

You can access the residuals and residual covariance information of insfilters (`insfilterMARG`, `insfilterAsync`, `insfilterErrorState`, and `insfilterNonholonomic`) and `ahrs10filter` through their object functions such as `fusegps`, `fusegyro`, `residual`, and `residualgps`.

For more details, see the Detect Noise in Sensor Readings with Residual Filtering example.

Track objects using TOMHT tracker Simulink block

Use the Track-Oriented Multi-Hypothesis Tracker Simulink block to track objects.

Model inertial measurement unit using IMU Simulink block

Use the IMU Simulink block to model an inertial measurement unit (IMU) composed of accelerometer, gyroscope, and magnetometer sensors.

For more details, see the IMU Sensor Fusion with Simulink example.

Estimate device orientation using AHRS Simulink block

Use the AHRS Simulink block to estimate the orientation of a device from its accelerometer, magnetometer, and gyroscope sensor readings.

For more details, see the IMU Sensor Fusion with Simulink example.

Calculate angular velocity from quaternions

Use `angvel` to calculate angular velocity from an array of quaternions.

Transform position and velocity between two frames to motion quantities in a third frame

Use `transformMotion` to transform position and velocity between two coordinate frames to motion quantities in a third coordinate frame.

For more details, see the Generate Off-centered IMU Readings examples.

Import Parameters to imuSensor

Use the `loadparams` object function to import parameters in JSON files to the `imuSensor` System object.

New examples

This release contains several new examples:

- Track-Level Fusion of Radar and Lidar Data
- Track Point Targets in Dense Clutter Using GM-PHD Tracker
- Track Space Debris Using a Keplerian Motion Model
- Introduction to Tracking Metrics
- Tuning a Multi-Object Tracker
- Detect Noise in Sensor Readings with Residual Filtering
- Generate Off-centered IMU Readings
- IMU Sensor Fusion with Simulink

R2019b

Perform track-level fusion using a track fuser

Use `trackFuser` to fuse tracks generated by tracking sensors or trackers and architect decentralized tracking systems.

For more details, see the Track-to-Track Fusion for Automotive Safety Applications example.

Track objects using a Gaussian mixture PHD tracker

Use `trackerPHD` with a `gmphd` filter to track point objects and extended objects with designated shapes. With `gmphd`, you can also use rectangular object models (such as `ctrect` and `ctrectmeas`) to track objects of rectangular shape.

For more details, see the Extended Object Tracking and Performance Metrics Evaluation example.

Evaluate tracking performance using the OSPA metric

Use `trackOSPAMetric` to evaluate the performance of a tracking system against truth based on the optimal subpattern assignment metric.

For more details, see the Extended Object Tracking and Performance Metrics Evaluation example.

Estimate orientation using a complementary filter

You can use `complementaryFilter` to estimate orientation based on accelerometer, gyroscope, and magnetometer sensor data.

For more details, see the Estimate Orientation with a Complementary Filter and IMU Data example.

Track objects using tracker Simulink blocks

You can use the GNN tracker and JPDA tracker Simulink blocks to track objects.

For more details on how to use these two blocks, see these example:

- Track Vehicles Using Lidar Data in Simulink
- Track Closely Spaced Targets Under Ambiguity in Simulink
- Track Simulated Vehicles Using GNN and JPDA Trackers in Simulink

Features supporting ENU reference frame

By specifying the `'ReferenceFrame'` argument, you can set the output reference frame for the following functions and objects as the ENU (east-north-up) frame. The default reference frame for these functions and objects is the NED (north-east-down) frame.

Features Supporting ENU	Description
`imuSensor`	IMU simulation model
`gpsSensor`	GPS receiver simulation model
`altimeterSensor`	Altimeter simulation model

Features Supporting ENU	Description
`ecompass`	Orientation from magnetometer and accelerometer readings
`imufilter`	Orientation from accelerometer and gyroscope readings
`ahrsfilter`	Orientation from accelerometer, gyroscope, and magnetometer readings
`ahrs10filter`	Height and orientation from MARG and altimeter readings
`insfilterMARG`	Estimate pose from MARG and GPS data
`insfilterAsync`	Estimate pose from asynchronous MARG and GPS data
`insfilterErrorState`	Estimate pose from IMU, GPS, and monocular visual odometry (MVO) data
`insfilterNonholonomic`	Estimate pose with nonholonomic constraints
`complementaryFilter`	Orientation estimation from a complementary filter

INS filter name and creation syntax changes

The names of these four INS (inertial navigation system) filters have changed.

Old Name	New Name
`MARGGPSFuser`	`insfilterMARG`
`AsyncMARGGPSFuser`	`insfilterAsync`
`ErrorStateIMUGPSFuser`	`insfilterErrorState`
`NHConstrainedIMUGPSFuser`	`insfilterNonholonomic`

Also, the old creation syntaxes, which can create INS filters with new names, will be removed in a future release. The new and recommended creation syntaxes directly create these filters by calling their names.

Old and Discouraged	New and Recommended
`filter = insfilter`	`filter = insfilterMARG`
`filter = insfilter('asyncimu')`	`filter = insfilterAsync`
`filter = insfilter('errorstate')`	`filter = insfilterErrorState`
`filter = insfilter('nonholonomic')`	`filter = insfilterNonholonomic`

New examples

This release contains several new examples:

- Track-to-Track Fusion for Automotive Safety Applications
- Simulate a Tracking Scenario Using an Interactive Application

- Estimate Orientation with a Complementary Filter and IMU Data
- Logged Sensor Data Alignment for Orientation Estimation
- Track Vehicles Using Lidar Data in Simulink
- Track Closely Spaced Targets Under Ambiguity in Simulink
- Track Simulated Vehicles Using GNN and JPDA Trackers in Simulink
- Convert Detections to `objectDetection` Format
- Remove Bias from Angular Velocity Measurement
- Estimating Orientation Using Inertial Sensor Fusion and MPU-9250
- Read and Parse NMEA Data Directly From GPS Receiver

R2019a

Version: 1.1

New Features

Bug Fixes

Track objects using a Joint Probabilistic Data Association (JPDA) tracker

Sensor Fusion and Tracking Toolbox includes `trackerJPDA` as an alternative to the existing `trackerGNN` and `trackerTOMHT`. `trackerJPDA` applies a soft assignment where multiple detections can contribute to each track, and balances the robustness and computational cost between `trackerGNN` and `trackerTOMHT`.

For more details on using `trackerJPDA`, see these examples:

- Track Vehicles Using Lidar: From Point Cloud to Track List
- Tracking Closely Spaced Targets Under Ambiguity

Track extended objects using a Probability Hypothesis Density (PHD) tracker

You can use `trackerPHD` to track extended objects using a Gamma Gaussian Inverse Wishart (GGIW) PHD filter, `ggiwphd`. `trackerPHD` creates a multisensor, multiobject tracker utilizing the multitarget PHD filters to estimate the states of the target.

For more details on using `trackerPHD`, see these examples:

- Marine Surveillance Using a PHD Tracker
- Extended Object Tracking

Simulate radar and IR detections from extended objects

To represent a platform's location as a "spatial extent" instead of a single point, you can use `radarSensor` and `irSensor` to simulate radar and IR detections from extended objects by specifying the `Dimensions` property of `platform`.

For more details on how to simulate radar detections from extended objects, see the `Marine Surveillance` example.

Improve tracker performance for large number of targets

`trackerGNN`, `trackerTOMHT` and `trackerJPDA` enable you to reduce the time required to update the tracker by setting a cost calculation threshold via the `AssignmentThreshold` property. This, along with other performance improvements, reduces the processing time when tracking a large number of targets.

For more details, see these examples:

- How to Efficiently Track Large Numbers of Objects
- Tracking a Flock of Birds

Estimate pose using accelerometer, gyroscope, GPS, and monocular visual odometry data

The `insfilter` can create an error-state Kalman filter suitable for pose (position and orientation) estimation based on accelerometer, gyroscope, GPS, and monocular visual odometry data. To create the error-state Kalman filter, use the `'errorState'` input argument.

Estimate pose using an extended continuous-discrete Kalman filter

The `insfilter` can create a continuous-discrete Kalman filter suitable for pose (position and orientation) estimation based on accelerometer, gyroscope, GPS, and magnetometer input. To create the continuous-discrete Kalman filter, use the `'asyncIMU'` input argument.

For more details, see the Pose Estimation From Asynchronous Sensors example.

Estimate height and orientation using MARG and altimeter data

Use `ahrs10filter` to estimate height and orientation based on altimeter readings and MARG (magnetic, angular rate, gravity) data. Typically, MARG data is derived from magnetometer, gyroscope, and accelerometer readings.

Simulate altimeter sensor readings

Use `altimeterSensor` to simulate altimeter sensor readings based on a ground-truth position.

Model and simulate bistatic radar tracking systems

`radarSensor`, `radarEmitter`, and `radarChannel` support modeling a radar tracking system with bistatic sensors (physically separated transmitter and receiver), including the effects of signal reflections from the target. To create a bistatic radar sensor, set the `DetectionMode` property of `radarSensor` to `'bistatic'`.

For more details, see the Tracking Using Bistatic Range Detections example.

Correct magnetometer readings for soft- and hard-iron effects

Use `magcal` to determine the coefficients needed to correct uncalibrated magnetometer data. You can correct for soft-iron effects, hard-iron effects, or both.

For more details, see the Magnetometer Calibration example.

Determine Allan variance of gyroscope data

Use `allanvar` to determine the Allan variance of gyroscope data. You can use the Allan variance to set noise parameters on your sensor models.

Generate quaternions from uniformly distributed random rotations

Use `randrot` to generate unit quaternions drawn from a uniform distribution of random rotations.

New application examples

This release contains several new application examples:

- Marine Surveillance Using a PHD Tracker shows how to use a PHD tracker to track extended ship targets with radar detections.
- Track Vehicles Using Lidar: From Point Cloud to Track List shows how to use a JPDA tracker to track vehicles with Lidar detections.
- How to Efficiently Track Large Numbers of Objects.
- Tracking a Flock of Birds.
- Tracking Using Bistatic Range Detections.
- Pose Estimation From Asynchronous Sensors.
- Magnetometer Calibration.
- How to Generate C Code for a Tracker.

R2018b

Version: 1.0

New Features

Single-Hypothesis and Multi-Hypothesis Multi-Object Trackers

Sensor Fusion and Tracking Toolbox provides multi-object trackers that fuse information from various sensors. Use `trackerGNN` to maintain a single hypothesis about the objects it tracks. Use `trackerTOMHT` to maintain multiple hypotheses about the objects it tracks.

Estimation Filters for Tracking

Sensor Fusion and Tracking Toolbox provides estimation filters that are optimized for specific scenarios, such as linear or nonlinear motion models, linear or nonlinear measurement models, or incomplete observability.

Estimation filters include:

Estimate Filters	Description
`trackingABF`	Alpha-beta filter
`trackingKF`	Linear Kalman filter
`trackingEKF`	Extended Kalman filter
`trackingUKF`	Unscented Kalman filter
`trackingCKF`	Cubature Kalman filter
`trackingPF`	Particle filter
`trackingMSCEKF`	Extended Kalman filter in modified spherical coordinates
`trackingGSF`	Gaussian-sum filter
`trackingIMM`	Interacting multiple model filter

Inertial Sensor Fusion to Estimate Pose

Sensor Fusion and Tracking Toolbox provides algorithms to estimate orientation and position from IMU and GPS data. The algorithms are optimized for different sensor configurations, output requirements, and motion constraints.

Inertial sensor fusion algorithms include:

Inertial Sensor Fusion Algorithm	Description
`ecompass`	Estimate orientation using magnetometer and accelerometer readings.
`imufilter`	Estimate orientation using accelerometer and gyroscope readings
`ahrsfilter`	Estimate orientation using accelerometer, gyroscope, and magnetometer readings
`insfilter`	Estimate position and orientation (pose) using IMU and GPS readings.

Active and Passive Sensor Models

Sensor Fusion and Tracking Toolbox provides active and passive sensor models. You can mimic environmental, channel, and sensor configurations by modifying parameters of the sensor models. For active sensors, you can model the corresponding emitters and channels as separate models.

Sensor models include:

Sensor Model	Description
`imuSensor`	IMU measurements of accelerometer, gyroscope, and magnetometer
`gpsSensor`	GPS position, velocity, groundspeed, and course measurements
`insSensor`	INS/GPS position, velocity, and orientation emulator
`monostaticRadarSensor`	Radar detection generator
`sonarSensor`	Active or passive sonar detection generator
`irSensor`	Infrared (IR) detection generator
`radarSensor`	Radio frequency detection generator

Trajectory and Scenario Generation

Generate ground-truth trajectories to drive sensor models using the `kinematicTrajectory` and `waypointTrajectory` System objects. Simulate tracking of multiple platforms in a 3-D arena using `trackingScenario`.

Visualization and Analytics

Use `theaterPlot` with `trackingScenario` to plot the ground-truth pose, detections, and estimated pose tracks for multi-object scenarios. Get error metrics for tracks using `trackErrorMetrics`. Analyze and compare the performance of multi-object tracking systems using `trackAssignmentMetrics`.

Orientation, Rotations, and Representation Conversions

The `quaternion` data type enables efficient representation of orientation and rotations. Sensor Fusion and Tracking Toolbox provides the following functions for use with the `quaternion` data type:

Rotations	
`rotateframe`	Quaternion frame rotation
`rotatepoint`	Quaternion point rotation

Representation Conversion	
`rotmat`	Convert quaternion to rotation matrix
`rotvec`	Convert quaternion to rotation vector (radians)

Representation Conversion	
`rotvecd`	Convert quaternion to rotation vector (degrees)
`parts`	Extract quaternion parts
`euler`	Convert quaternion to Euler angles (radians)
`eulerd`	Convert quaternion to Euler angles (degrees)
`compact`	Convert quaternion array to N-by-4 matrix

Metrics and Interpolation	
`dist`	Angular distance in radians
`norm`	Quaternion norm
`meanrot`	Quaternion mean rotation
`slerp`	Spherical linear interpolation

Initialization and Convenience Functions	
`ones`	Create quaternion array with real parts set to one and imaginary parts set to zero
`zeros`	Create quaternion array with all parts set to zero
`classUnderlying`	Class of parts within quaternion
`normalize`	Quaternion normalization

Mathematics	
`times, .*`	Element-wise quaternion multiplication
`mtimes, *`	Quaternion multiplication
`prod`	Product of a quaternion array
`minus, -`	Quaternion subtraction
`uminus, -`	Quaternion unary minus
`conj`	Complex conjugate of quaternion
`ldivide, .\`	Element-wise quaternion left division
`rdivide, ./`	Element-wise quaternion right division
`exp`	Exponential of quaternion array
`log`	Natural logarithm of quaternion array
`power, .^`	Element-wise quaternion power

Array Manipulation	
`ctranspose, '`	Complex conjugate transpose of quaternion array
`transpose, .'`	Transpose of quaternion array

Sensor Fusion and Tracking Examples

The release of Sensor Fusion and Tracking Toolbox includes the following examples.

Applications
Air Traffic Control
Multiplatform Radar Detection Fusion
Passive Ranging Using a Single Maneuvering Sensor
Tracking Using Distributed Synchronous Passive Sensors
Search and Track Scheduling for Multifunction Phased Array Radar
Extended Object Tracking
Visual-Inertial Odometry Using Synthetic Data
IMU and GPS Fusion for Inertial Navigation

Multi-Object Trackers
Multiplatform Radar Detection Fusion
Tracking Closely Spaced Targets Under Ambiguity
Tracking Using Distributed Synchronous Passive Sensors
Extended Object Tracking
Introduction to Using the Global Nearest Neighbor Tracker
Introduction to Track Logic

Estimation Filters
Tracking Maneuvering Targets
Tracking with Range-Only Measurements
Passive Ranging Using a Single Maneuvering Sensor

Inertial Sensor Fusion
Estimate Orientation Through Inertial Sensor Fusion
IMU and GPS Fusion for Inertial Navigation
Estimate Position and Orientation of a Ground Vehicle
Estimate Orientation and Height Using IMU, Magnetometer, and Altimeter

Sensor Models
Inertial Sensor Noise Analysis Using Allan Variance
Simulating Passive Radar Sensors and Radar Interferences
Introduction to Simulating IMU Measurements
Introduction to Tracking Scenario and Simulating Radar Detections
Scanning Radar Mode Configuration

Trajectory and Scenario Generation
Introduction to Tracking Scenario and Simulating Radar Detections
Benchmark Trajectories for Multi-Object Tracking
Multiplatform Radar Detection Generation

Quaternion Representation
Rotations, Orientation and Quaternions
Lowpass Filter Orientation Using Quaternion SLERP