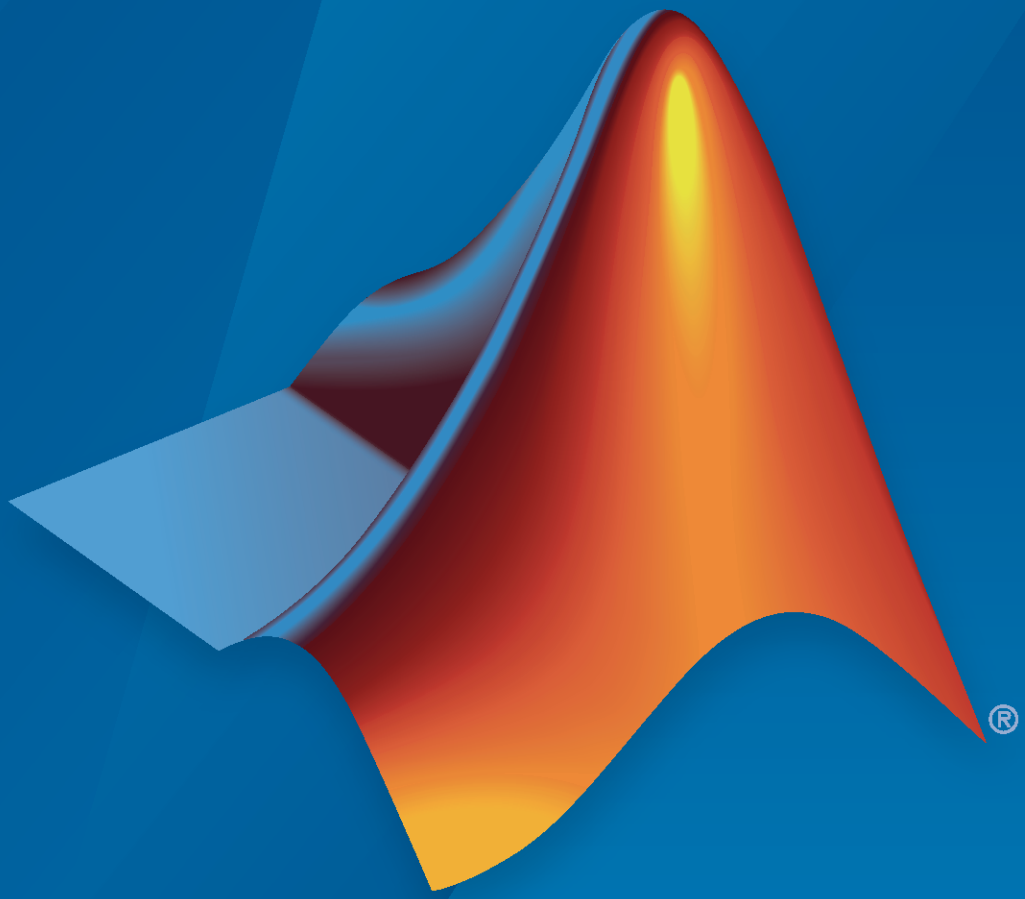
Sensor Fusion and Tracking Toolbox™ Release Notes



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

R2023a

R2022b

R2022a

R2021b

R2021a

R2020b

R2020a

R2019b

R2019a

R2018b

R2023a

Track targets with detection class fusion using trackerJPDA System object

With the `trackerJPDA` System object™, you can track targets using detections that contain target classification information and output tracks that contain track classification information. Using class fusion can reduce track switches and increase estimation accuracy.

To enable class fusion, set the `ClassFusionMethod` property of the object to `"Bayes"`. Additionally, use these two properties to adjust the class fusion algorithm:

- `InitialClassProbabilities` — A priori class probability of new tracks
- `ClassFusionWeight` — Weight of class fusion likelihood in the mixed likelihood of detection-to-track association

To specify a detection with class information, use the `ObjectClassParameters` property of the `objectDetection` object.

Smooth track estimates with joint integrated probabilistic data association (JIPDA) smoother

With the `smootherJIPDA` object, you can now perform offline tracking, also known as fixed-interval track smoothing, using a JIPDA smoothing algorithm.

Online tracking uses sensor data from the initial time to the estimation time to provide an estimate at the estimation time. Offline tracking uses all the sensor data including those beyond the estimation time. By using measurements from succeeding time steps and estimating joint association probabilities between forward and backward tracks, the smoother can estimate data associations more accurately and resolve ambiguities more efficiently than a JIPDA tracker.

For more details, see these examples:

- “Introduction to JIPDA Smoothing”
- “Understand and Analyze JIPDA Smoother Algorithm”

Evaluate tracking performance using CLEAR MOT metrics

Using the `trackCLEARMetrics` object, you can evaluate the Classification of Events, Activities, and Relationships (CLEAR) Multi-Object Tracking (MOT) metrics by comparing tracks with ground truth. Other than CLEAR, the object also outputs Mostly-Tracked, Partially-Tracked, and Mostly-Lost MOT metrics. You can choose the similarity method, used to compare tracks with truth, as intersection over union, Euclidean distance, or a custom method.

For more details, see the “Implement Simple Online and Realtime Tracking” example.

Specify waypointTrajectory using ground speed or velocity input and new properties

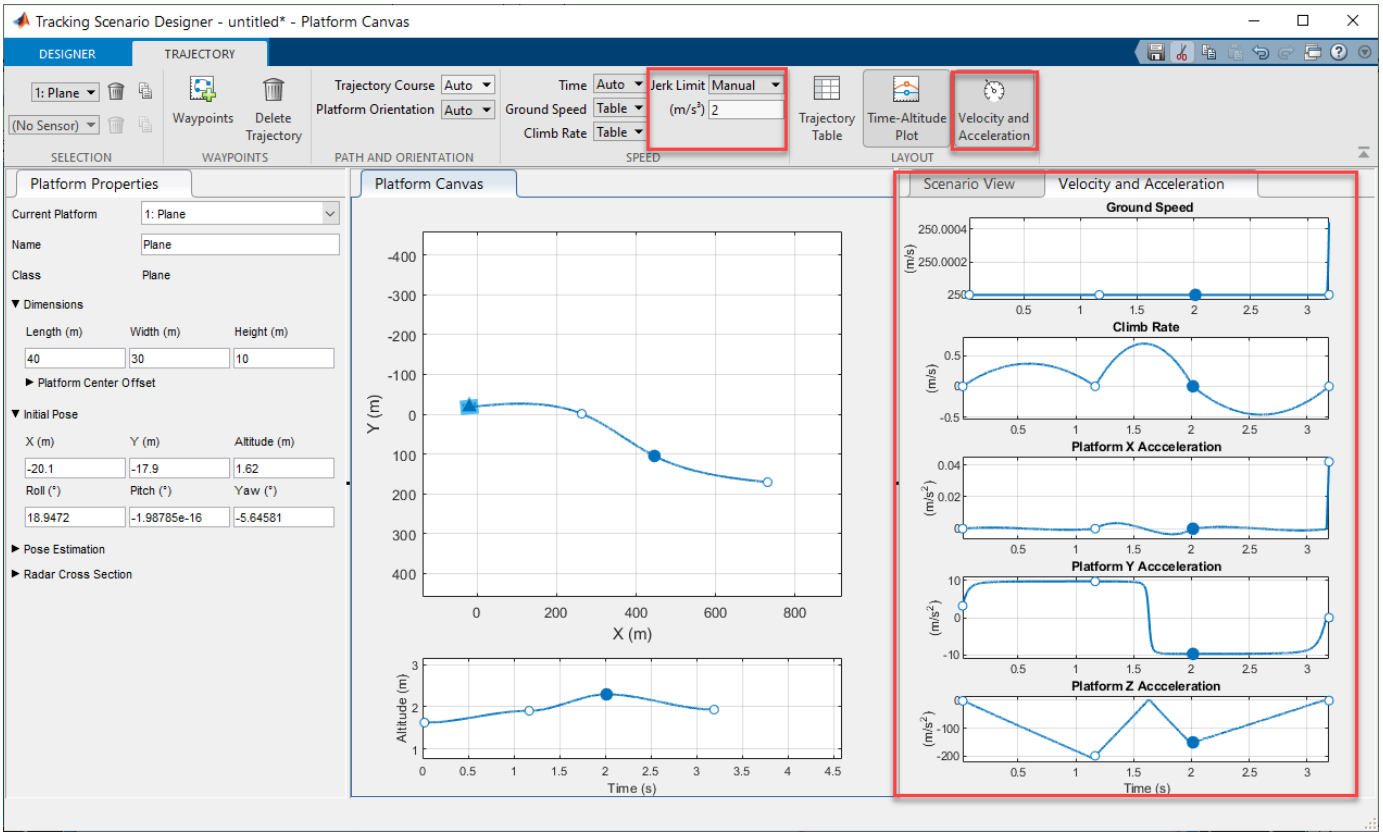
When creating a `waypointTrajectory` object, if you specify the velocity or ground speed input, the time-of-arrival input is no longer required. When you do not specify the time-of-arrival input, you can use these new properties:

- `JerkLimit` — Longitudinal limit of trajectory jerk. Jerk is the derivative of the translational acceleration. If you specify a finite value for the jerk limit, `waypointTrajectory` produces a horizontal trapezoidal acceleration profile based on `JerkLimit`.
- `InitialTime` — Time before trajectory starts. If you specify a nonzero value, `waypointTrajectory` delays the start of the trajectory by the initial time.
- `WaitTime`— Wait time at each waypoint. If you specify a nonzero value for a waypoint, `waypointTrajectory` waits at the waypoint.

For more details, see the “Define Trajectory Using Positions and Ground Speed” example.

Visualize trajectory profiles and enable jerk limit in Tracking Scenario Designer

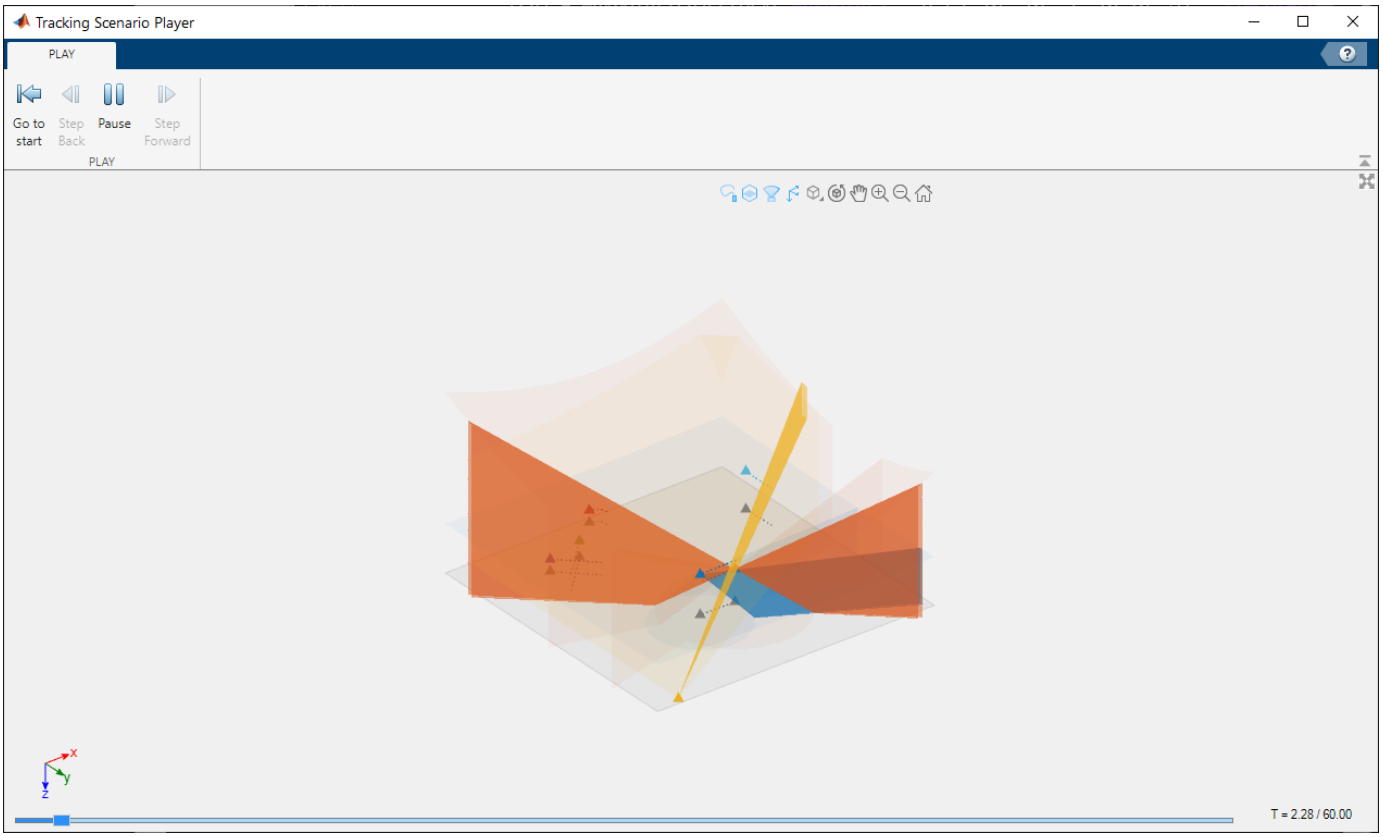
When specifying platform trajectories in the **Tracking Scenario Designer** app, you can now visualize the velocity and acceleration profiles. You can also customize the jerk limit by setting the **Jerk Limit** parameter to `Manual` and specifying the exact jerk limit.



Play trackingScenario and trackingScenarioRecording in Tracking Scenario Player

You can now play a `trackingScenario` object in the **Tracking Scenario Player** app by using its `play` object function.

You an also play a `trackingScenarioRecording` object in the **Tracking Scenario Player** app by using its `play` object function.



Tune trackingIMM filter using trackingFilterTuner

You can now tune a `trackingIMM` filter object by using the `trackingFilterTuner` object. The tunable properties of a `trackingIMM` object are:

- `TransitionProbabilities`
- `ModelProbabilities`
- All tunable properties of filter objects contained in the `TrackingFilters` property

To support tuning, these two object functions are enabled for `trackingIMM`:

- `tunableProperties` — Get tunable properties of filter
- `setTunedProperties` — Set properties to tuned values

Previously, you could use the `trackingFilterTuner` object to tune the `trackingKF`, `trackingEKF`, `trackingCKF`, and `trackingUKF` objects.

For more details, see the “Automatically Tune Filter to Track Maneuvering Targets” example.

Generate Tuning Data for Tuning Tracking Filter

You can now use the `tuningData` object function of `trackingFilterTuner` to create tuning data that contains detection logs and truth tables based on `trackingScenarioRecording` objects. To tune a tracking filter with the created tuning data, use the `tune` object function.

Tune tracking filter with new solver and display in trackingFilterTuner

When tuning a tracking filter by using the `trackingFilterTuner` object, you can now specify the solver as `"active-set"`. The previously supported solvers, `"fmincon"`, `"patternsearch"`, and `"particleswarch"`, require an Optimization Toolbox™ license or a Global Optimization Toolbox license.

Additionally, you can now control the tuning iteration display by specifying the new `Display` property as:

- `"Text"` — An iterative textual display in the MATLAB command window
- `"None"` — No display during tuning
- `"Plot"` — Plot the tuning cost as a function of iteration
- `"SolverOptions"` — Control the display by using the `SolverOptions` property

Previously, you could use only the `SolverOptions` property to control the tuning display.

Evaluate untuned cost and retrieve tuned parameters in trackingFilterTuner

Use the `parameterCost` object function of `trackingFilterTuner` to obtain the cost before tuning. You can optionally specify tuned parameters, detection log, and truth log for evaluating the tuning cost.

Use the `tunedParameters` object function of `trackingFilterTuner` to retrieve the latest tuned parameters after the tuning process terminates.

Smooth state estimates using insEKF

You can now obtain smoothed state estimates as the second output from the `estimateStates` object function of the `insEKF` object. The function uses the Rauch-Tung-Striebel (RTS) smoothing algorithm to smooth state estimates.

Estimate orientation using Complementary Filter Simulink block

You can now use the Complementary Filter Simulink® block to estimate orientation based on accelerometer, gyroscope, and magnetometer sensor data.

Obtain component-level information from trackerPHD

You can now obtain component level information from the `trackerPHD` System object by using its `getPHDFilter` object function. The function returns a copy of the `ggiwphd` or `gmphd` object used in the `trackerPHD` object.

New examples

These new examples are now available:

- “Object-Level Fusion of Lidar and Camera Data for Vehicle Tracking”
- “Automatically Tune Filter to Track Maneuvering Targets”
- “Visual Tracking of Occluded and Unresolved Objects”
- “Import Camera-Based Datasets in MOT Challenge Format for Object Tracking”
- “Implement Simple Online and Realtime Tracking”
- “Autonomous Underwater Vehicle Pose Estimation Using Inertial Sensors and Doppler Velocity Log”
- “Introduction to JIPDA Smoothing”
- “Understand and Analyze JIPDA Smoother Algorithm”
- “Define Trajectory Using Positions and Ground Speed”

Functionality being removed or changed

Default solver of trackingFilterTuner changed
Behavior change

As of R2023a, the default value of the `Solver` property of `trackingFilterTuner` is `"active-set"`. Previously, the default value of the `Solver` property was `"fmincon"`, which requires an Optimization Toolbox license.

getTrackPositions and getTrackVelocities take empty cell or track structure as input
Behavior change

As of R2023a, the `getTrackPositions` and `getTrackVelocities` functions can take an empty cell or an empty track structure as input. See the reference pages for details on the output formats when using an empty input. Previously, providing an empty input resulted in the two functions throwing an error.

R2022b

Version: 2.4

New Features

Bug Fixes

Track targets with detection class fusion using trackerGNN System object

With the `trackerGNN` System object, you can track targets using detections that contain target classification information and output tracks that contain track classification information. In addition to providing target classification information, using class fusion also reduces track switches and increases estimation accuracy.

Specify class confusion matrix for `objectDetection` object

Using the new `ObjectClassParameters` property of the `objectDetection` object, you can specify detection class statistics in the form of a confusion matrix.

Fuse detection classification information using `trackerGNN` System object

Using the `trackerGNN` System object, you can fuse detection classification information by setting the `ClassFusionMethod` property of the object to `"Bayes"`. This enables the tracker to use the detection classification information to assign tracks based on the Bayesian Product Class Fusion algorithm.

Additionally, you can use these two properties to adjust the class fusion algorithm:

- `InitialClassProbabilities` — A priori class probability of new tracks.
- `ClassFusionWeight` — Weight of class fusion cost in the overall assignment cost.

Represent track class probability in `objectTrack` object

The `objectTrack` object has a new property, `ObjectClassProbabilities`, which represents the probabilities that the tracked target belongs to specific classes.

For more details on how to use class fusion, see the Introduction to Class Fusion and Classification-Aided Tracking example.

Tune tracking filters for improved estimation performance

Using the `trackingFilterTuner` object, you can now tune these tracking filters using measurements and truth data. The tuning process adjusts the tunable properties of the filter to minimize the cost between measurements and truth data.

Tracking Filter	Tunable Property
`trackingKF`	• `ProcessNoise` — Process noise matrix. By default, the tuner tunes this property. • `StateCovariance` — Initial state estimate error covariance matrix. By default, the tuner does not tune this property.
`trackingEKF`	
`trackingCKF`	

Tracking Filter	Tunable Property
`trackingUKF`	• `ProcessNoise` — Process noise matrix. By default, the tuner tunes this property. • `StateCovariance` — Initial state estimate error covariance matrix. By default, the tuner does not tune this property. • `Alpha` — Sigma point spread around state. By default, the tuner tunes this property. • `Beta` — Distribution of sigma points. By default, the tuner does not tune this property. • `Kappa` — Secondary scaling factor for generating sigma points. By default, the tuner does not tune this property.

To tune a tracking filter, create a `trackingFilterTuner` object, specify its `FilterInitializationFcn` property to generate the tracking filter, and use the `tune` object function.

The `tune` object function outputs the tuned properties as a structure. You can use the `setTunedProperties` object function to apply the tuned properties to the filter. Alternately, you can output an initialization function of the tuned filter by using the `exportToFunction` object function. You can also plot the filter estimation errors after tuning using the `plotFilterErrors` object function.

To configure how the tuner tunes the filter, use these properties of the `trackingFilterTuner` object:

- `Cost`
- `Solver`
- `UseMex`
- `UseParallel`
- `SolverOptions`

To change specific properties to tune or specific elements in a property to tune from the default, follow these steps:

1. Obtain the default tunable properties of a tracking filter object by using its `tunableProperties` function, which returns a `tunableFilterProperties` object.
2. Use the `setPropertyTunability` object function of the `tunableFitlerProperties` object to specify the properties or property elements you want to tune.
3. Set the `TunablePropertiesSource` property of the `trackingFilterTuner` object to `"Custom"`.
4. Specify the `CustomTunableProperties` property of the `trackingFilterTuner` object as the configured `tunableFitlerProperties` object.

For more details, see the Automatically Tune Tracking Filter for Multi-Object Tracker example.

Generate radar detections using Fusion Radar Sensor block

You can now generate radar detections by using the Fusion Radar Sensor block in Simulink. The inputs to the block are target poses in the reference frame of the sensor mounting platform and optionally the simulation time. You can configure the block parameters to output unclustered detections, clustered detections, and tracks.

For more details, see these examples:

- Air Traffic Control in Simulink
- Extended Object Tracking with Radar for Marine Surveillance in Simulink

Transform target poses from scenario frame to platform frame in Simulink

You can use the Scenario To Platform block to transform target platform poses expressed in the scenario frame to target platform poses expressed in the reference frame of a specific platform.

For more details, see the Extended Object Tracking with Radar for Marine Surveillance in Simulink example.

Create tracking sensor configuration from sensor, platform, and tracking scenario

You can now create `trackingSensorConfiguration` objects, which are required by the `trackerPHD` and `trackerGridRFS` System objects, from a sensor, platform, and tracking scenario. Specifically, you can:

- Generate a `trackingSensorConfiguration` object directly based on one of these sensor objects:
  - `monostaticLidarSensor`
  - `irSensor`
  - `sonarSensor`
  - `lidarPointCloudGenerator` (Automated Driving Toolbox)
  - `visionDetectionGenerator` (Automated Driving Toolbox)

  Previously, you could directly generate a `trackingSensorConfiguration` object from only the `fusionRadarSensor`, `radarDataGenerator` (Radar Toolbox), or `drivingRadarDataGenerator` (Automated Driving Toolbox) object.
- Generate a `trackingSensorConfiguration` object based on one of these Simulink blocks:
  - Fusion Radar Sensor
  - Driving Radar Data Generator (Automated Driving Toolbox)
  - Radar Data Generator (Radar Toolbox)
  - Vision Detection Generator (Automated Driving Toolbox)
  - Lidar Point Cloud Generator (Automated Driving Toolbox)
- Generate `trackingSensorConfigurarion` objects for all sensors mounted on a `Platform` object.

- Specify a platform pose input representing the coordinate transformation from the scenario to the sensor frame when you create the `trackingSensorConfigurarion` object.
- Generate `trackingSensorConfigurarion` objects for all sensors in a `trackingScenario` object.

Generate more memory-efficient C/C++ code from probability hypothesis density (PHD) tracker

You can now generate single-precision and non-dynamic memory allocation C/C++ code from the `trackerPHD` System object and the Probability Hypothesis Density Tracker block.

- The `trackingSensorConfiguration` object, which you can use to specify sensor configurations for a PHD tracker, now has a new property, `MaxNumDetections`. Use the new property to set the maximum number of detections by a specific sensor.
- These filter initialization functions now accept a data type argument, which you can specify as `"single"` or `"double"`.

  - `initcvggiwphd`
  - `initcaggiwphd`
  - `initctggiwphd`
  - `initcvgmphd`
  - `initcagmphd`
  - `initctgmphd`
  - `initctrectgmphd`

  If you specify a single-precision filter initialization function and use single-precision detection inputs, the PHD tracker executes single-precision code.
- Specify the maximum number of PHD components maintained in the `trackerPHD` System object using the new `MaxNumComponents` property. Similarly, the Probability Hypothesis Density Tracker block has a new parameter, **Maximum number of components**.

Generate strict single-precision and non-dynamic memory allocation C/C++ code from partitionDetections and mergeDetections

You can now generate strict single-precision and non-dynamic memory allocation C/C++ code from the `partitionDetections` and `mergeDetections` functions. You can use these functions to cluster multiple detections from the same object and track extended objects using conventional trackers, such as the `trackerGNN` System object.

Plot ground surfaces using theater plot

You can plot ground surfaces contained in a `trackingScenario` object by following these steps:

- Use the `surfacePlotter` object function to create a `SurfacePlotter` object associated with a `theaterPlot` object.
- Use the `surfacePlotterData` function to get the plotter data based on the `SurfaceManager` object saved in the `SurfaceManager` property of the `trackingScenario` object.

- Use the `plotSurface` object function to plot the surfaces based on the surface plotter and the plotter data.

Copy, reset, and create motion and sensor models for insEKF object

You can use these functions to copy, reset, and create motion and sensor models for the `insEKF` object in inertial sensor fusion design.

- Use the `insCreateMotionModelTemplate` function to create a motion model template based on the `positioning.insMotionModel` abstract class.
- Use the `insCreateSensorModelTemplate` function to create a sensor model template based on the `positioning.insSensorModel` abstract class.
- Use the `copy` object function to create a copy of an `insEKF` object. The function copies all the properties of the `insEKF` object including motion and sensor models used in the filter.
- Use the `reset` object function to reset the `State` and `StateCovariance` properties of the `insEKF` object to their default values.
- If you define a custom motion model using the `positioning.insMotionModel` abstract class, you can optionally implement a `copy` method for copying non-public properties.
- If you define a custom sensor model using the `positioning.insSensorModel` abstract class, you can optionally implement a `copy` method for copying non-public properties.

Specify wait and reverse motion for waypoint trajectory

You can now specify wait and reverse motion using the `waypointTrajectory` System object.

- To let the trajectory wait at a specific waypoint, simply repeat the waypoint coordinate in two consecutive rows when specifying the `Waypoints` property.
- To render reverse motion, separate positive (forward) and negative (backward) groundspeed values by a zero value in the `GroundSpeed` property.

Confirm track directly in multi-object trackers

You can now directly confirm a track by using the `confirmTrack` object function of the `trackerGNN` and `trackerJPDA` System objects. You can also directly confirm a branch by using the `confirmBranch` object function of the `trackerTOMHT` System object.

Obtain position, velocity, and covariance from tracks using motion model name input

By using the `getTrackPositions` and `getTrackVelocities` functions, you can now obtain positions, velocities, and associated covariances of tracks by specifying the motion model name as an input. For example,

```
[positions,covariances] = getTrackPositions(tracks,"constvel")
```

returns positions and position covariances in tracks based on the constant-velocity model in the `constvel` function.

Previously, you could use only the position selector or velocity selector input to obtain the position and velocity states. For example,

```
positionSelector = [1 0 0 0 0 0 0 0 0;
                    0 0 0 1 0 0 0 0 0;
                    0 0 0 0 0 0 1 0 0];
[positions,covariances] = getTrackPositions(tracks,positionSelector)
```

extracts positions and associated covariances from `tracks` using a position selector.

Specify measurement means and covariances in mergeDetections merging function

In the `mergeDetections` function, when you specify the `MergingFcn` name-value argument, you can now also specify measurement means and covariances as inputs to the merging function. Previously, you could use only `objectDetection` objects or its equivalent structures as inputs. For more details, see the description of the `MergingFcn` input argument.

New examples

These new examples are now available:

- Introduction to Class Fusion and Classification-Aided Tracking
- Automatically Tune Tracking Filter for Multi-Object Tracker
- Angle and Position Measurement Fusion for Marine Surveillance
- Asynchronous Angle-only Tracking with GM-PHD Tracker
- Gesture Recognition Using Inertial Measurement Units
- Air Traffic Control in Simulink
- Analyze Track and Detection Association Using Analysis Info

R2022a

Fuse inertial sensor data using insEKF-based flexible fusion framework

Use an EKF-based flexible fusion framework to fuse measurement data generated from accelerometer, gyroscope, magnetometer, GPS, and other sensors to estimate the platform state, such as position, velocity, acceleration, orientation, and angular velocity.

Using the `insEKF` object, you can build an inertial sensor estimate framework that integrates with various sensor models and motion models. To configure the parameters of the filter, use an `insOptions` object. The `insEKF` object provides object functions for various estimation purposes:

- Use the `predict` and `fuse` (or `correct`) object functions to fuse sensor data for sequential state estimation.
- Use the `estimateStates` object function to batch process cumulative sensor data and estimate the platform state.
- Use the `tune` object function to tune the filter parameters for better estimation performance.
- Use other object functions to access the filter state and other components in the filter.

Sensor Fusion and Tracking Toolbox provides these sensor models that you can integrate with the `insEKF` object to enable the processing of sensor measurements:

- `insAccelerometer`
- `insGyroscope`
- `insMagnetometer`
- `insGPS`

You can also define your own sensor model by inheriting from the `positioning.insSensorModel` interface class.

Sensor Fusion and Tracking Toolbox also provides two motion models that you can use in the `insEKF` object to propagate the filter state:

- `insMotionOrientation` — Motion model for 3-D orientation estimation, assuming a constant angular velocity.
- `insMotionPose` — Motion model for 3-D pose (position and orientation) estimation, assuming a constant angular velocity and a constant linear acceleration.

You can also define your own motion model by inheriting from the `positioning.insMotionModel` interface class.

For more details, see these topics and examples:

- Fuse Inertial Sensor Data Using `insEKF`-Based Flexible Fusion Framework
- Design Fusion Filter for Custom Sensors
- Estimate Orientation Using GPS-Derived Yaw Measurements

Manage memory footprint of global nearest neighbor (GNN) multi-object tracker and joint probabilistic data association (JPDA) tracker

For the `trackerGNN` System object, you can solve the assignment problem using smaller data sets or clusters by specifying the `AssignmentClustering` property as `"on"`. With assignment clustering

enabled, the tracker separates out unassignable tracks or detections into new clusters and potentially reduces the memory footprint and computational costs of the tracker. Similarly, you can enable assignment clustering for the Global Nearest Neighbor Multi Object Tracker Simulink block by specifying the **Cluster tracks and detections for assignment** parameter as `on`.

For the `trackerGNN` and `trackerJPDA` System objects, specify the `EnableMemoryManagement` property as `true` to enable these properties for managing the memory footprint of the tracker:

- `MaxNumDetectionsPerSensor`
- `MaxNumDetectionsPerCluster`
- `MaxNumTracksPerCluster`
- `ClusterViolationHandling`

Similarly, for the Global Nearest Neighbor Multi Object Tracker block and the Joint Probabilistic Data Association Multi Object Tracker block, select the **Enable memory management** parameter to enable these parameters for managing the footprint of the tracker:

- **Maximum number of detections per sensor**
- **Maximum number of detections per cluster**
- **Maximum number of tracks per cluster**
- **Handle run-time violation of cluster size**

Managing memory footprint is important for many code generation and hardware deployment applications. For more details, see the Processor-in-the-Loop Verification of JPDA Tracker for Automotive Applications example.

Export tracking architecture, tracker, and track fuser to Simulink

Use the `exportToSimulink` object function to export a `trackingArchitecture` System object, consisting of trackers and track fusers, to a Subsystem block in a Simulink model. The function automatically uses the Track Concatenation block and the Detection Concatenation block to handle track and detection concatenation, respectively, in the architecture. The function also uses the Delay (Simulink) block when the architecture contains algebraic loops.

You can also use the `exportToSimulink` object function to export these tracker or track fuser objects to their corresponding blocks in Simulink.

Tracker or Fuser Object	Corresponding Simulink Block
`trackerGNN`	Global Nearest Neighbor Multi Object Tracker
`trackerJPDA`	Joint Probabilistic Data Association Multi Object Tracker
`trackerTOMHT`	Track-Oriented Multi-Hypothesis Tracker
`trackerPHD`	Probability Hypothesis Density (PHD) Tracker
`trackFuser`	Track-To-Track Fuser

For more details, see these examples:

- Export `trackingArchitecture` to Simulink

- Define and Test Tracking Architectures for System-of-Systems in Simulink

Correct filter estimates using multiple out-of-sequence measurements (OOSMs) with joint probabilistic data association (JPDA) algorithm

You can use the `retroCorrectJPDA` object function to correct the state estimates of the `trackingKF`, `trackingEKF`, and `trackingIMM` filter objects using multiple OOSMs, based on the joint probabilistic data association algorithm.

Handle multiple out-of-sequence detections using JPDA tracker

You can use the `trackerJPDA` System object or the Joint Probabilistic Data Association Multi Object Tracker block to handle multiple out-of-sequence detections.

To enable the retrodiction capability of the `trackerJPDA` System object, specify its `OOSMHandling` property as `"Retrodiction"`, and then specify its `MaxNumOOSMSteps` property as a positive integer. You must specify the `FilterInitializationFcn` property to return one of these tracking filters:

- `trackingKF`
- `trackingEKF`
- `trackingIMM`

Similarly, you can enable the retrodiction capability of the Joint Probabilistic Data Association Multi Object Tracker block by specifying its **Out-of-sequence measurements handling** parameter as `Retrodiction` and specifying the **Maximum number of OOSM steps** as a positive integer. You must specify the **Filter initialization function** parameter to return one of these tracking filters:

- `trackingKF`
- `trackingEKF`
- `trackingIMM`

For more details, see the Handle Out-of-Sequence Measurements in Multisensor Tracking Systems example.

Retrodict out-of-sequence measurements (OOSMs) using trackingIMM filter object

You can now use the retrodiction algorithm provided in the `trackingIMM` filter object to process OOSMs and improve state estimate accuracy, in addition to the previously supported `trackingKF` and `trackingEKF` filter objects. After updating the filter regularly, by calling the `predict` and `correct` object functions, you can use the `retrodict` and `retroCorrect` object functions to incorporate an OOSM into the state estimate.

You can also enable retrodiction when using the `trackingIMM` filter with a `trackerGNN` or `trackerJPDA` System object. First, specify the `OOSMHandling` property of the tracker object as `"Retrodiction"`. Then specify the `FilterInitializationFcn` property such that the tracker returns a `trackingIMM` object. You can also adjust the maximum number of OOSM steps using the `MaxNumOOSMSteps` property of the tracker.

Similarly, you can enable retrodiction when using the `trackingIMM` filter with a Global Nearest Neighbor Multi Object Tracker block or a Joint Probabilistic Data Association Multi Object Tracker

block by specifying the **Out-of-sequence measurements handling**, **Filter initialization function**, and **Maximum number of OOSM steps** parameters.

Simulate out-of-sequence detections using objectDetectionDelay System object

You can use the `objectDetectionDelay` System object to apply a time delay to `objectDetection` objects and generate out-of-sequence detections. The `objectDetectionDelay` object acts as a buffer, into which you can save delayed detections and from which you can retrieve out-of-sequence detections that are available at the current time.

You can specify the time delay applied to each `objectDetection` in these ways, depending on how you specify the `DelaySource` property:

- `"Property"` — Specify the time delay applied to the detections as a constant, uniform, or normal distribution using the `DelayDistribution` property.
- `"Input"` — Specify the time delay as an input to the `objectDetectionDelay` object.

For more details, see the How to Simulate Out-of-Sequence Measurements example.

Model ground surface and terrain occlusion in trackingScenario object

You can now model ground surface and terrain occlusion in the `trackingScenario` object.

To add a ground surface to a tracking scenario, use the `groundSurface` object function, which adds a `GroundSurface` object to the scenario object. The `GroundSurface` object enables you to specify the terrain, boundary, and reference height of the ground surface. You can also use the `height` object function to obtain the terrain height at a specified position, as well as use the `occlusion` function to determine if the line-of-sight between two points is occluded by the surface. You can add multiple `GroundSurface` objects to the scenario, as long as they do not overlap with each other.

By default, the tracking scenario object enables terrain occlusion when you use the `detect` object function of the scenario, or the `detect` object function of a `Platform` object in the scenario. Also, if the `IsEarthCentered` property of the tracking scenario is set to `true`, by default the scenario models the surface of the Earth using the WGS84 model and enables detection occlusion caused by the surface of the Earth.

You can use the `SurfaceManager` object, saved in the `SurfaceManager` property of the tracking scenario, to manage the use of the ground surfaces in the scenario.

- To disable terrain occlusion, specify the `UseOcclusion` property of the `SurfaceManager` object as `false`.
- You can determine if the line-of-sight between two positions in the scenario is occluded by surfaces by using the `occlusion` object function of the `SurfaceManager` object.
- You can obtain the height of the terrain in the scenario by using the `height` object function of the `SurfaceManager` object.

For more details, see the Simulate and Track Targets with Terrain Occlusions example.

Wrap measurements in tracking filters to prevent filter divergence

You can enable measurement wrapping for these tracking filter objects:

- `trackingEKF`
- `trackingUKF`
- `trackingCKF`
- `trackingIMM`
- `trackingGSF`
- `trackingMSCEKF`

First, specify the `MeasurementWrapping` property as `true`, and then specify the `MeasurementFcn` property as a measurement function with two outputs: the measurement and the measurement wrapping bound. With this setup, the filter wraps the measurement residuals according to the measurement bounds, which helps prevent the filter from diverging due to incorrect measurement residual values.

These measurement functions have predefined wrapping bounds:

- `cvmeas`
- `cameas`
- `ctmeas`
- `cvmeasmsc`
- `singermeas`

For these functions, the wrapping bounds are [–180, 180] degrees for azimuth angle measurements and [–90, 90] degrees for elevation angle measurements. Other measurements are unwrapped.

You can also customize a wrapping-enabled measurement function by returning the wrapping bounds as the second output of the measurement function.

For more details, see the Track Objects with Wrapping Azimuth Angles and Ambiguous Range and Range Rate Measurements example.

Evaluate cumulative tracking performance using OSPA(2) metric

You can now use the optimal subpattern assignment squared metric ($OSPA^{(2)}$) to evaluate cumulative tracking performance for a duration of time. To enable the $OSPA^{(2)}$ metric in the `trackOSPAMetric` System object, specify its `Metric` property as `"OSPA(2)"`. The object provides four new properties to specify the length of the sliding window and adjust various weights in the metric:

- `WindowLength`
- `WindowSumOrder`
- `WindowWeights`
- `WindowWeightExponent`

Similarly, you can enable the $OSPA^{(2)}$ metric in the Optimal Subpattern Assignment Metric Simulink block by specifying the **Metric** parameter as `OSPA(2)`. The block provides four new parameters to specify the length of the sliding window and adjust various weights in the metric:

- **Window length**
- **Window sum order**
- **Window weights**

- **Window weight exponent**

To evaluate instantaneous tracking performance using the traditional OSPA metric in the `trackOSPAMetric` object, specify the `Metric` property as `"OSPA"`. Similarly, in the Optimal Subpattern Assignment Metric block, specify the **Metric** parameter as `OSPA`.

Specify tracking sensor configuration using configuration output from fusionRadarSensor System object

You can now specify the sensor configuration by using the configuration output from the `fusionRadarSensor` System object.

- The configuration structure output from the `fusionRadarSensor` System object now contains two additional fields:
  - `RangeLimits` — The lower and upper limits of the detection range of the sensor, returned as a two-element real-valued vector in meters.
  - `RangeRateLimits` — The lower and upper limits of the detection range-rate of the sensor, returned as a two-element real-valued vector in m/s.
- You can now directly create a `trackingSensorConfiguration` object using a `fusionRadarSensor` object, instead of specifying name-value arguments for the object properties. For example:

  ```
  sensor = fusionRadarSensor(1);
  config = trackingSensorConfiguration(sensor)
  ```
- You can now use the configuration structure output from the `fusionRadarSensor` object to update the sensor configuration directly during each call of a `trackerPHD` or `trackerGridRFS` object. Previously, you needed to specify the sensor configuration input as `trackingSensorConfiguration` objects.

Time Scope MATLAB object now supports additional signal statistics

Starting in R2022a, the `timescope` object supports these additional signal statistics:

- Standard deviation
- Variance
- Mean square

For more details, see Configure Time Scope MATLAB Object.

Configure timescope measurements programmatically

These properties configure measurement data programmatically for the `timescope` object:

- `MeasurementChannel`
- `BilevelMeasurements`
- `CursorMeasurements`
- `PeakFinder`
- `SignalStatistics`

- `Trigger`

Using these properties, you can now interact with the scope from the command line. Any changes that you make to the properties in the command line change the corresponding values in the UI. Any changes that you make in the UI change the corresponding property values in the command line.

Functionality being removed or changed

Behavior changes of trackingKF object
Behavior change

As of R2022a, the `trackingKF` filter object has these behavior changes:

- The object now accepts and uses the process noise specified using the `ProcessNoise` property. Previously, the object ignored the process noise specified in the `ProcessNoise` property.
- If you set the `MotionModel` property to a predefined state transition model, such as `"1D Constant Velocity"`, you can no longer specify the control model for the filter. To use a control model, specify the `MotionModel` property as `"Custom"`.
- To specify a control model, you must:

  - Specify the `MotionModel` property as `"Custom"` and use a customized motion model.
  - Specify the control model when creating the filter.

  Also, you cannot change the size of the control model matrix.
- You can no longer change the size of the measurement model matrix specified in the `MeasurementModel` property.
- The dimensions of the process matrix set through the `ProcessNoise` property now differ between a predefined motion model and a customized motion model.

  - If the specified motion model is a predefined motion model, specify the `ProcessNoise` property as a *D*-by-*D* matrix, where *D* is the dimension of the motion. For example, $D = 2$ for the `"2D Constant Velocity"` motion model.
  - If the specified motion model is a custom motion model, specify the `ProcessNoise` property as an *N*-by-*N* matrix, where *N* is the dimension of the state. For example, $N = 6$ if you customize a 3-D motion model in which the state is (x, v_x, y, v_y, z, v_z).
- The orientation of the state now matches that of the state vector that you specify when creating the filter. For example, if you set the initial state in the filter as a row vector, the filter displays the filter state as a row vector and outputs the state as a row vector when using the `predict` or `correct` object functions. Previously, the filter displayed and output the filter state as a column vector regardless of the initial state.
- You can generate efficient C/C++ code without dynamic memory allocation for `trackingKF`.

New examples

This release contains these new examples:

- Processor-in-the-Loop Verification of JPDA Tracker for Automotive Applications
- Object Tracking Using Time Difference of Arrival (TDOA)
- How to Simulate Out-of-Sequence Measurements

- Track Objects with Wrapping Azimuth Angles and Ambiguous Range and Range Rate Measurements
- Simulate and Track Targets with Terrain Occlusions
- Tuning Kalman Filter to Improve State Estimation
- Design Fusion Filter for Custom Sensors
- Estimate Orientation Using GPS-Derived Yaw Measurements
- Ground Vehicle Pose Estimation for Tightly Coupled IMU and GNSS
- Export `trackingArchitecture` to Simulink
- Define and Test Tracking Architectures for System-of-Systems in Simulink
- Extended Object Tracking with Radar for Marine Surveillance in Simulink

R2021b

Version: 2.2

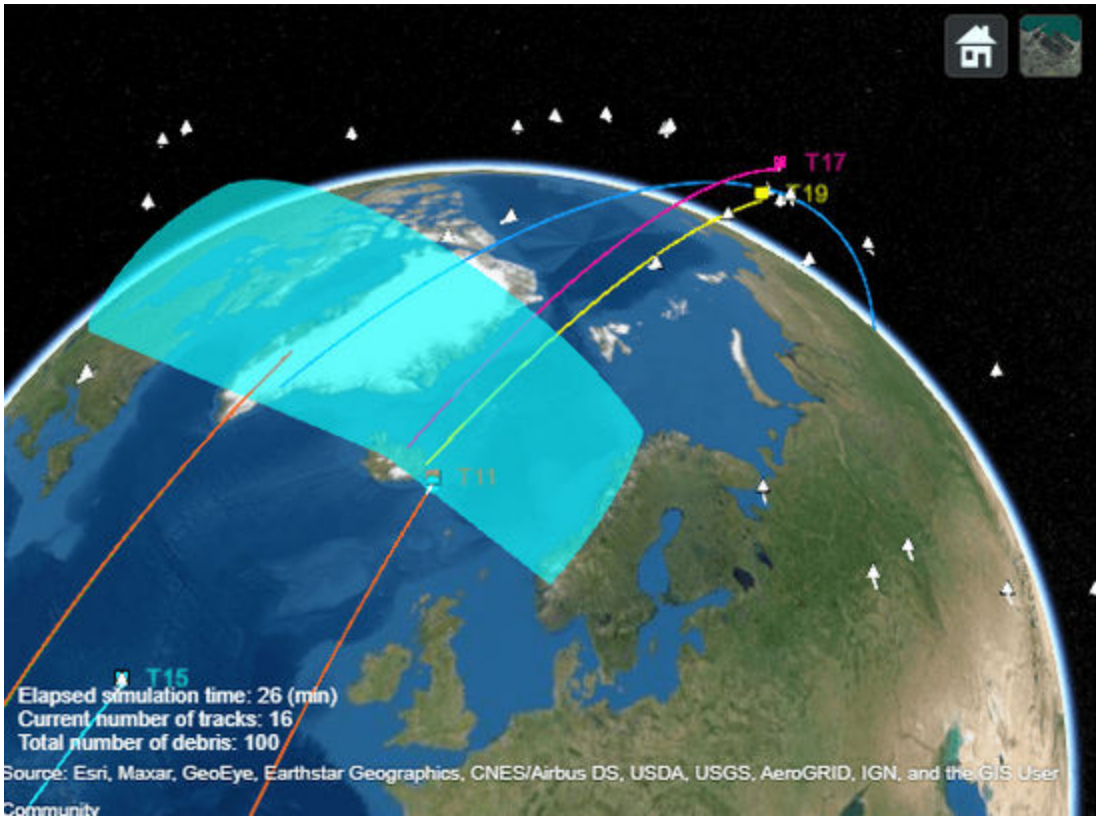
New Features

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.

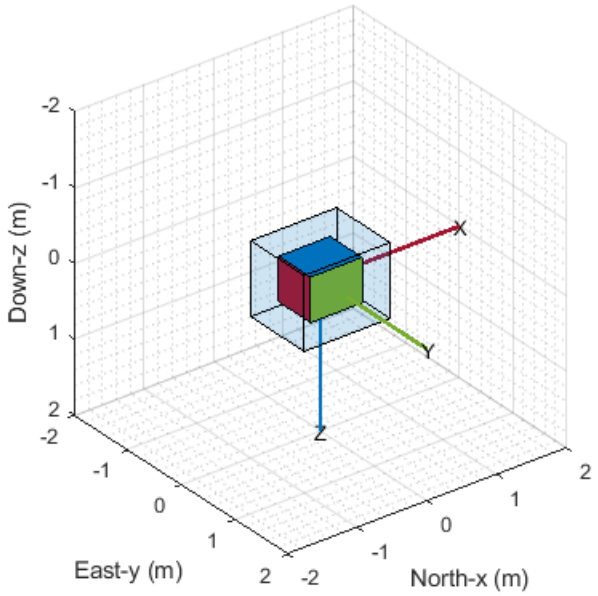
Model and Simulate GPS sensor using GPS Simulink block

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 5-6.

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

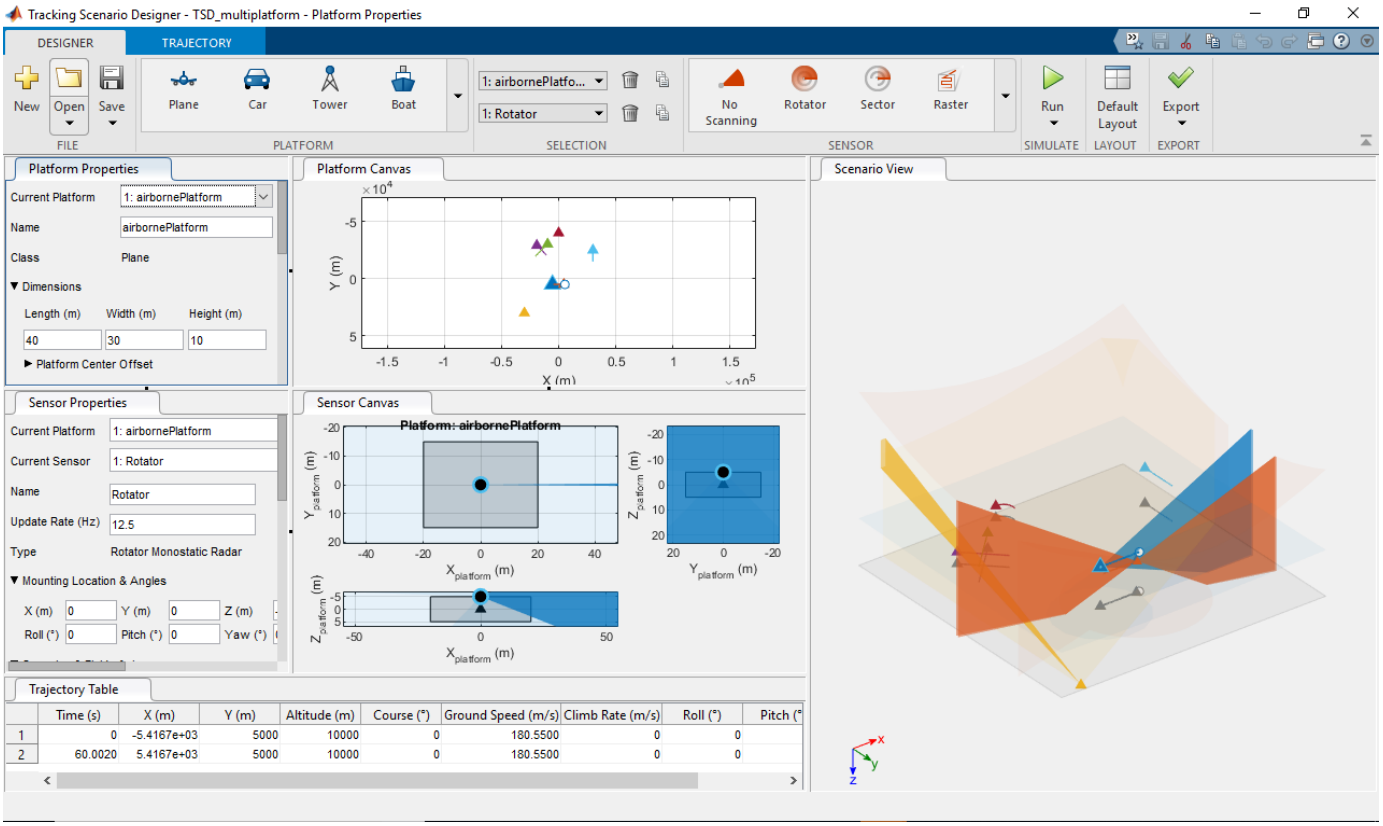
New Features

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.

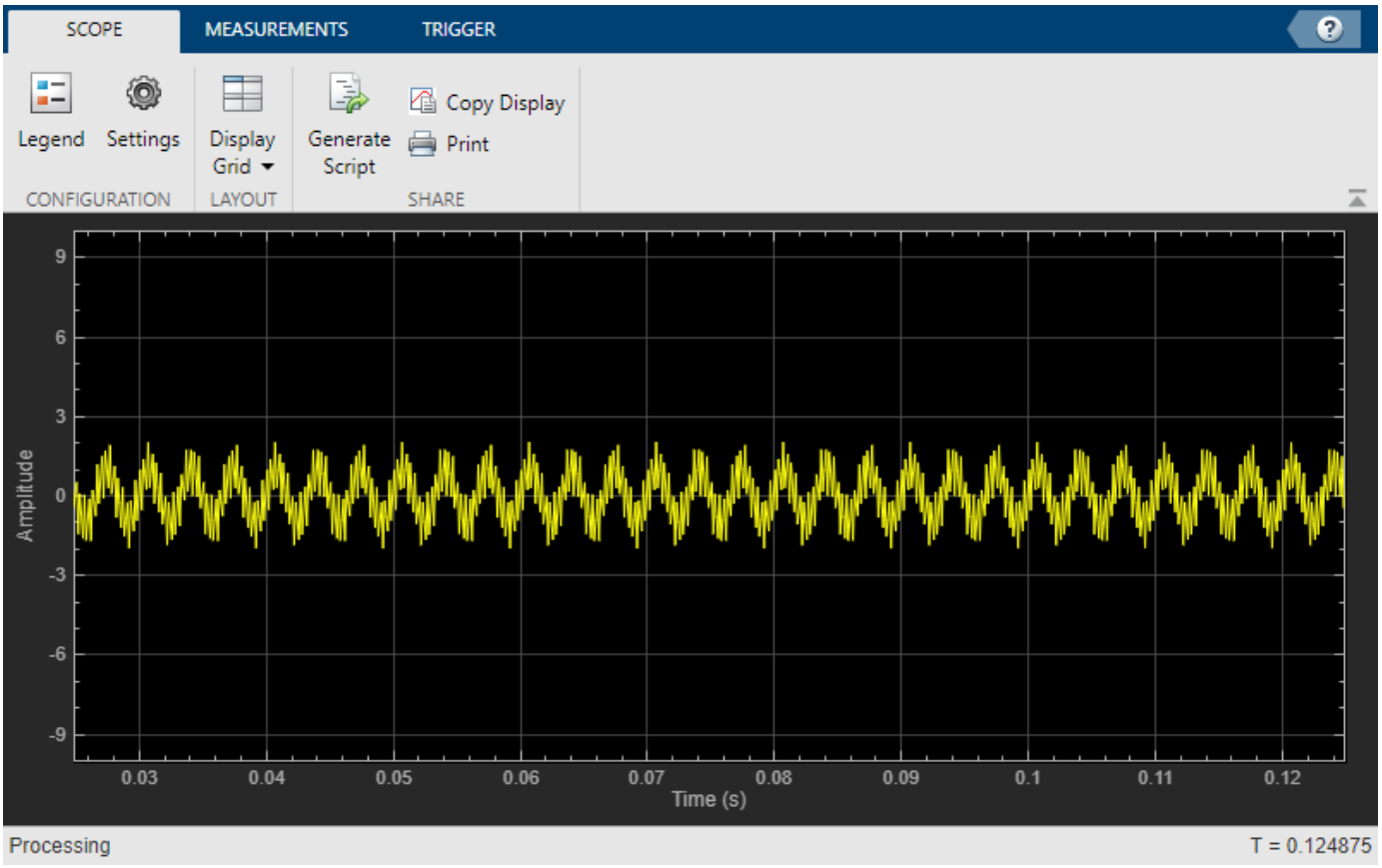
New time scope object: Visualize signals in the time domain

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

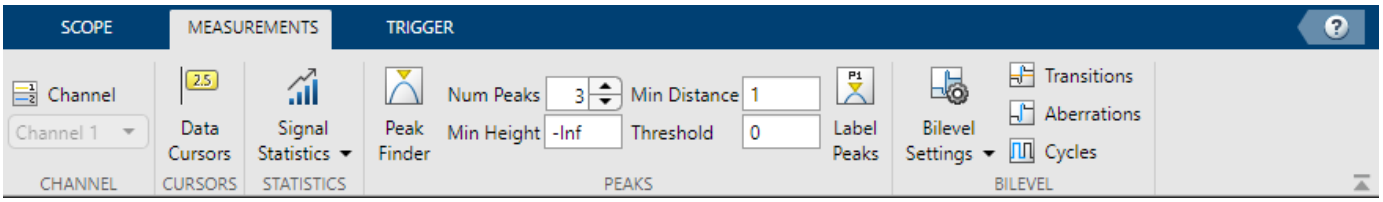
On the **Scopes** tab, you can control the layout and configuration settings, and set the display settings of the Time Scope. You can also generate script to recreate your time scope with the same settings. When doing so, an editor window opens with the code required to recreate your `timescope` object.



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

Version: 1.2

New Features

Bug Fixes

Compatibility Considerations

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