Sensor Fusion and Tracking Toolbox™

User's Guide

MATLAB®&SIMULINK®

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

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

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

Contents

Featured Examples

5

Tracking Scenarios

Tracking Simulation Overview

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

1 Use the tracking scenario generator to create ground truth for all moving and stationary radar platforms and all target platforms (planes, ships, cars, drones). The `trackingScenario` class models the motion of all platforms in a global coordinate system called scenario coordinates. These objects can represent ships, ground vehicles, airframes, or any object that the radar detects. See "Orientation, Position, and Coordinate" for a discussion of coordinate systems.

2 Optionally, simulate an inertial navigation system (INS) that provides radar sensor platform position, velocity, and orientation relative to scenario coordinates.

3 Create models for each radar sensor with specifications and parameters using the `monostaticRadarSensor`, `radarSensor`, or `radarEmitter` objects. Using target platform pose and profile information, generate synthetic radar detections for each radar-target combination. Methods belonging to `trackingScenario` retrieve the pose and profile of any target platform. The `trackingScenario` generator does not have knowledge of scenario coordinates. It knows the relative positions of the target platforms with respect to the body platform of the radar. Therefore, the detector can only generate detections relative to the radar location and orientation.

   If there is an INS attached to a radar platform, then the radar can transform detections to the scenario coordinate system. The INS allows multiple radars to report detections in a common coordinate system.

4 Process radar detections with a multi-object tracker to associate detections to existing tracks or create tracks. Multi-object tracks include `trackerGNN`, `trackerTOMHT`, `trackerJPDA` and `trackerPHD`. If there is no INS, the tracker can only generate tracks specific to one radar. If an INS is present, the tracker can create tracks using measurements from all radars.

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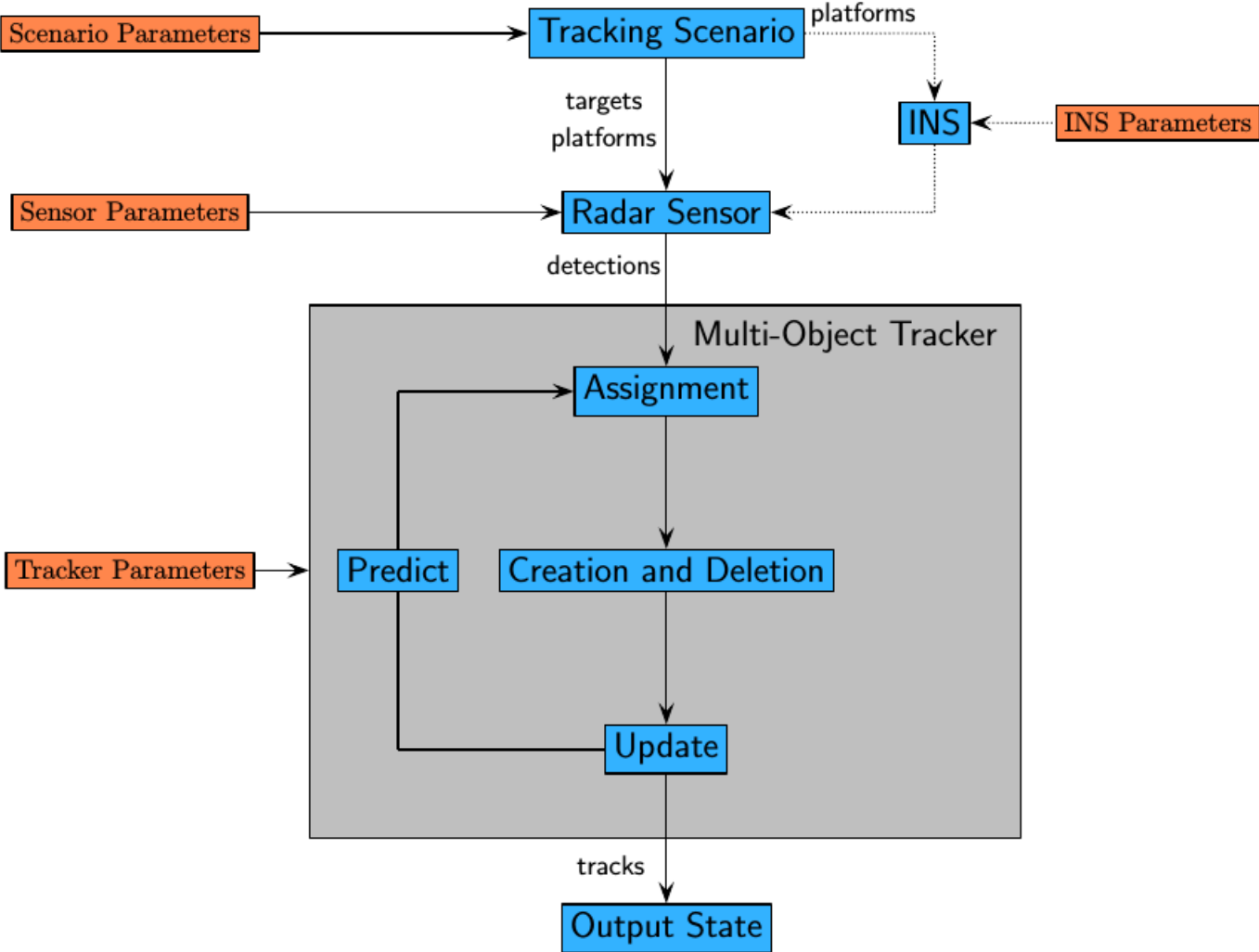
Creating a Tracking Scenario

You can define a tracking simulation by using the `trackingScenario` object. By default, the object creates an empty scenario. You can then populate the scenario with platforms by calling the `platform` method as many times as needed. A platform is an object (moving or stationary), which can either be a sensor, a target, or any other entity. A platform can be modeled as a point or a cuboid by specifying the `Dimensions` property of `Platform`. After creating a platform, you can specify the motion of the platform by using its `Trajectory` property. To configure a trajectory, you can use `waypointTrajectory`, which allows you to specify the 3-D waypoints that the platform follows and the associated arrival time for each waypoint. Alternately, you can use `kinematicTrajectory`, which allows you to specify the 3-D acceleration and angular velocity of the platform with initial pose and translational velocity. You can also specify the orientation of a platform using the `Orientation` property of `kinematicTrajectory` or `waypointTrajectory`.

Run the simulation by calling the `advance` method on the `trackingScenario` object in a loop, or by calling the `record` method to run the simulation all at once. You can set the simulation update interval using the `UpdateRate` property in the `trackingScenario` object. You can set the properties of a platform or leave them to their default value. You can set them all except for `PlatformID`. The complete list of `Platform` properties is shown here.

Platform Properties

`PlatformID`	Scenario-defined platform ID.
`ClassID`	User-specified platform classification ID.
`Dimensions`	3-D dimensions of a cuboid that approximates the size of a platform and offset of the origin of the platform body frame from the center of the cuboid. The default value of `Dimensions` has all fields equal to zero, which corresponds to a point model.
`Trajectory`	Platform motion, specified by `kinematicTrajecotry` or `waypointTrajectory`.
`Signatures`	Platform signatures, specified as a cell array of `irSignature`, `rcsSignature`, and `tsSignature` objects. A signature represents the reflection or emission pattern of a platform.
`PoseEstimator`	A pose estimator, specified as a pose-estimator object such as `insSensor` (default).
`Emitter`	Emitters mounted on platform, specified as a cell array of emitter objects, such as `radarEmitter` or `sonarEmitter`.
`Sensors`	Sensors mounted on platform, specified as a cell array of sensor objects such as `irSensor` or `sonarSensor`.

At any time during the simulation, you can retrieve the current values of platform properties using the `platformPoses` and `platformProfiles` methods of the `trackingScenario` object. Both the `platformPoses` and `platformProfiles` methods return properties of all platforms with respect to the scenario's NED frame. You can also use the `pose` method of the `Platform` to return the

properties of one specific platform. In addition, the `Platform.targetPoses` method, while similar, returns properties of other platforms with respect to a specified platform.

Create Tracking Scenario with Two Platforms

Construct a tracking scenario with two platforms that follow different trajectories.

```
sc = trackingScenario('UpdateRate',100.0,'StopTime',1.2);
```

Create two platforms.

```
platfm1 = platform(sc);
platfm2 = platform(sc);
```

Platform 1 follows a circular path of radius 10 m for one second. This is accomplished by placing waypoints in a circular shape, ensuring that the first and last waypoint are the same.

```
wpts1 = [0 10 0; 10 0 0; 0 -10 0; -10 0 0; 0 10 0];
time1 = [0; 0.25; .5; .75; 1.0];
platfm1.Trajectory = waypointTrajectory(wpts1, time1);
```

Platform 2 follows a straight path for one second.

```
wpts2 = [-8 -8 0; 10 10 0];
time2 = [0; 1.0];
platfm2.Trajectory = waypointTrajectory(wpts2,time2);
```

Verify the number of platforms in the scenario.

```
disp(sc.Platforms)
```

```
    {1×1 fusion.scenario.Platform}    {1×1 fusion.scenario.Platform}
```

Run the simulation and plot the current position of each platform. Use an animated line to plot the position of each platform.

```
figure
grid
axis equal
axis([-12 12 -12 12])
line1 = animatedline('DisplayName','Trajectory 1','Color','b','Marker','.');
line2 = animatedline('DisplayName','Trajectory 2','Color','r','Marker','.');
title('Trajectories')
p1 = pose(platfm1);
p2 = pose(platfm2);
addpoints(line1,p1.Position(1),p1.Position(2));
addpoints(line2,p2.Position(2),p2.Position(2));

while advance(sc)
    p1 = pose(platfm1);
    p2 = pose(platfm2);
    addpoints(line1,p1.Position(1),p1.Position(2));
    addpoints(line2,p2.Position(2),p2.Position(2));
    pause(0.1)
end
```

Trajectories
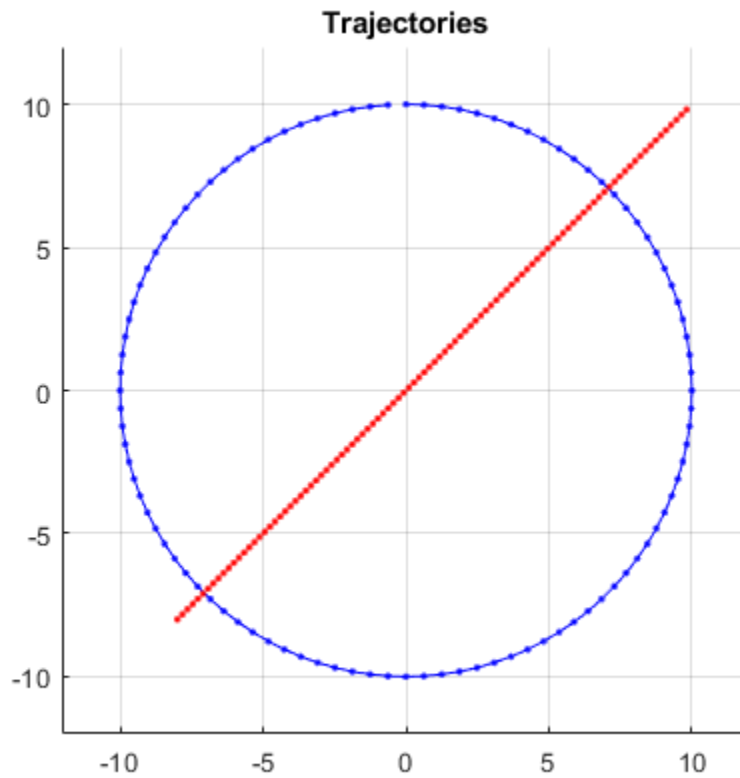
Plot the waypoints for both platforms.

```
hold on
plot(wpts1(:,1),wpts1(:,2),' ob')
text(wpts1(:,1),wpts1(:,2),"t = " + string(time1),'HorizontalAlignment','left','VerticalAlignment
plot(wpts2(:,1),wpts2(:,2),' or')
text(wpts2(:,1),wpts2(:,2),"t = " + string(time2),'HorizontalAlignment','left','VerticalAlignment
hold off
```

Trajectories
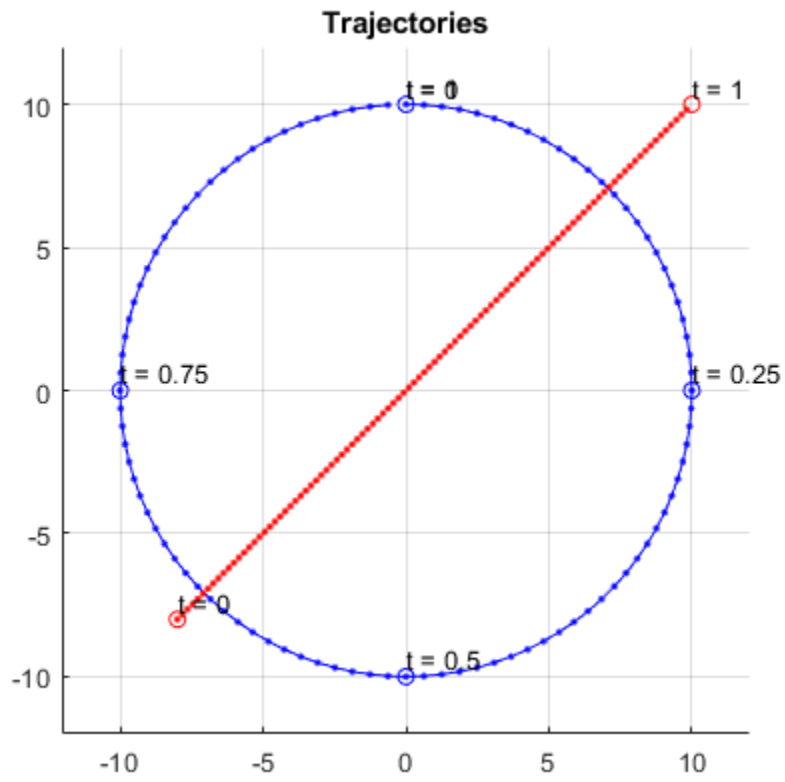
Radar Detections

The radar detectors `monostaticRadarSensor` and `radarSensor` generate measurements from target poses.

Simulate Radar Detections

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

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

The radar sensor is assumed to be mounted on a platform and carried by the platform as it maneuvers. A platform can carry multiple sensors. When you create a sensor, you specify sensor positions and orientations with respect to the body coordinate system of a platform. Each call to `monostaticRadarSensor` creates a sensor. The output of `monostaticRadarSensor` generates the detection that can be used as input to multi-object trackers, such as `trackerGNN`, or any tracking filters, such as `trackingKF`.

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

Create Radar Sensor

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

```
radar1 = monostaticRadarSensor( ...
    'SensorIndex',1,...
    'UpdateRate',10, ...            % Hz
    'ReferenceRange', 111.0e3, ...          % m
    'ReferenceRCS', 0.0, ...              % dBsm
    'MaxMechanicalScanRate',5, ...   % deg/s
    'FieldOfView',[70,10], ...                % [az;el] deg
    'HasElevation',false, ...
    'HasRangeRate',false, ...
    'AzimuthResolution',1.4, ...          % deg
    'RangeResolution', 135.0)              % m
```

Convenience Syntaxes

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

- `sensor = monostaticRadarSensor('Rotator')` creates a `monostaticRadarSensor` object that mechanically scans 360° in azimuth. Setting `HasElevation` to `true` points the radar antenna towards the center of the elevation field of view.

- `sensor = monostaticRadarSensor('Sector')` creates a `monostaticRadarSensor` object that mechanically scans a 90° azimuth sector. Setting `HasElevation` to `true`, points the radar antenna towards the center of the elevation field of view. You can change the `ScanMode` to `'Electronic'` to electronically scan the same azimuth sector. In this case, the antenna is not mechanically tilted in an electronic sector scan. Instead, beams are stacked electronically to process the entire elevation spanned by the scan limits in a single dwell.

- `sensor = monostaticRadarSensor('Raster')` returns a `monostaticRadarSensor` object that mechanically scans a raster pattern spanning 90° in azimuth and 10° in elevation upwards from the horizon. You can change the `ScanMode` property to `'Electronic'` to perform an electronic raster scan in the same volume.

- `sensor = monostaticRadarSensor('No scanning')` returns a `monostaticRadarSensor` object that stares along the radar antenna boresight direction. No mechanical or electronic scanning is performed.

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

```
sensor = monostaticRadarSensor(1,'Raster','ScanMode','Electronic')
```

Radar Sensor Parameters

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

```
help monostaticRadarSensor
```

at the command line.

Sensor location parameters.

Sensor Location

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

DetectionCoordinates	Specifies the coordinate system for detections reported in the "Detections" on page 2-15 output `struct`. The coordinate system can be one of: • `'Scenario'` –- detections are reported in the scenario coordinate frame in rectangular coordinates. This option can only be selected when the sensor `HasINS` property is set to `true`. • `'Body'` –- detections are reported in the body frame of the sensor platform in rectangular coordinates. • `'Sensor rectangular'` –- detections are reported in the radar sensor coordinate frame in rectangular coordinates aligned with the sensor frame axes. • `'Sensor spherical'` –- detections are reported in the radar sensor coordinate frame in spherical coordinates based on the sensor frame axes.

Sensitivity parameters.

Sensitivity Parameters

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

Sensor resolution and bias parameters.

Resolution Parameters

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

Enabling parameters.

Enabling Parameters

HasElevation	This property allows the radar sensor to scan in elevation and estimate elevation from target detections.
HasRangeRate	This property allows the radar sensor to estimate range rate.
HasFalseAlarms	This property allows the radar sensor to generate false alarm detection reports.
HasRangeAmbiguities	When true, the radar does not resolve range ambiguities. When a radar sensor cannot resolve range ambiguities, targets at ranges beyond the MaxUnambiguousRange property value are wrapped into the interval [0 MaxUnambiguousRange]. When false, targets are reported at their unwrapped range.
HasRangeRateAmbiguites	When true, the radar does not resolve range rate ambiguities. When a radar sensor cannot resolve range rate ambiguities, targets at range rates above the MaxUnambiguousRadialSpeed property value are wrapped into the interval [0 MaxUnambiguousRadialSpeed]. When false, targets are reported at their unwrapped range rates. This property only applies when the HasRangeRate property is set to true.
HasNoise	Specifies if noise is added to the sensor measurements. Set this property to true to report measurements with noise. Set this property to false to report measurements without noise. The reported measurement noise covariance matrix contained in the output objectDetection struct is always computed regardless of the setting of this property.
HasOcclusion	Enable occlusion from extended objects, specified as true or false. Set this property to true to model occlusion from extended objects. Note that both extended objects and point targets can be occluded by extended objects, but a point target cannot occlude another point target or an extended object. Set this property to false to disable occlusion of extended objects.
HasINS	Set this property to true to enable an optional input argument to pass the current estimate of the sensor platform pose to the sensor. This pose information is added to the MeasurementParameters field of the reported detections. Then, the tracking and fusion algorithms can estimate the state of the target detections in scenario coordinates.

Scan parameters.

Scan Parameters

ScanMode	This property specifies the scan mode used by the radar as one of: • 'No scanning' -- the radar does not scan. The radar beam points along the antenna boresight. • 'Mechanical' -- the radar mechanically scans between the azimuth and elevation limits specified by the MechanicalScanLimits property. • 'Electronic' -- the radar electronically scans between the azimuth and elevation limits specified by the ElectronicScanLimits property. • 'Mechanical and electronic' -- the radar mechanically scans the antenna boresight between the mechanical scan limits and electronically scans beams relative to the antenna boresight between the electronic scan limits. The total field of regard scanned in this mode is the combination of the mechanical and electronic scan limits. In all scan modes except 'No scanning', the scan proceeds at angular intervals specified by the radar field of view specified in FieldOfView.
MaxMechanicalScanRate	This property sets the magnitude of the maximum mechanical scan rate of the radar. When HasElevation is true, the scan rate is a vector consisting of separate azimuthal and elevation scan rates. When HasElevation is false, the scan rate is a scalar representing the azimuthal scan rate. The radar sets its scan rate to step the radar mechanical angle by the radar field of regard. When the required scan rate exceeds the maximum scan rate, the maximum scan rate is used.
MechanicalScanLimits	This property specifies the mechanical scan limits of the radar with respect to its mounted orientation. When HasElevation is true, the limits are specified by minimum and maximum azimuth and by minimum and maximum elevation. When HasElevation is false, limits are specified by minimum and maximum azimuth. Azimuthal scan limits cannot span more than 360 degrees and elevation scan limits must lie in the closed interval [-90 90].

ElectronicScanLimits	This property specifies the electronic scan limits of the radar with respect to the current mechanical angle. When `HasElevation` is `true`, the limits are specified by minimum and maximum azimuth and by minimum and maximum elevation. When `HasElevation` is `false`, limits are specified by minimum and maximum azimuth. Both azimuthal and elevation scan limits must lie in the closed interval `[-90 90]`.
FieldOfView	This property specifies the sensor azimuthal and elevation fields of view. The field of view defines the total angular extent observed by the sensor during a sensor update. The field of view must lie in the interval *(0,180]*. Targets outside of the sensor angular field of view during a sensor update are not detected.

Range and range rate parameters.

Range and Range Rate Parameters

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

Detector Input

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

The structure contains these fields.

Field	Description
PlatformID	Unique identifier for the platform, specified as a scalar positive integer. This is a required field with no default value.

Field	Description
ClassID	User-defined integer used to classify the type of target, specified as a nonnegative integer. Zero is reserved for unclassified platform types and is the default value.
Position	Position of target in platform coordinates, specified as a real-valued, 1-by-3 vector. This is a required field with no default value. Units are in meters.
Velocity	Velocity of target in platform coordinates, specified as a real-valued, 1-by-3 vector. Units are in meters per second. The default is [0 0 0].
Acceleration	Acceleration of target in platform coordinates specified as a 1-by-3 row vector. Units are in meters per second-squared. The default is [0 0 0].
Orientation	Orientation of the target with respect to platform coordinates, specified as a scalar quaternion or a 3-by-3 rotation matrix. Orientation defines the frame rotation from the platform coordinate system to the current target body coordinate system. Units are dimensionless. The default is quaternion(1,0,0,0).
AngularVelocity	Angular velocity of target in platform coordinates, specified as a real-valued, 1-by-3 vector. The magnitude of the vector defines the angular speed. The direction defines the axis of clockwise rotation. Units are in degrees per second. The default is [0 0 0].

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

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

Target Poses

platformID
ClassID
Position
Velocity
Yaw
Pitch
Roll
AngularVelocity

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

Platform Profiles

PlatformID
ClassID
RCSPattern
RCSAzimuthAngles
RCSElevationAngles

Radar Sensor Coordinate Systems

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

In a radar simulation, there is always a top-level global coordinate system which is usually the North-East-Down (NED) Cartesian coordinate system defined by a tangent plane at any point on the surface of the Earth. The `trackingScenario` object models the motion of platforms in the global coordinate system. When you create a platform, you specify its location and orientation relative to the global frame. These quantities define the body axes of the platform. Each radar sensor is mounted on the body of a platform. When you create a sensor, you specify its location and orientation with respect to the platform body coordinates. These quantities define the sensor axes. The body and radar axes can

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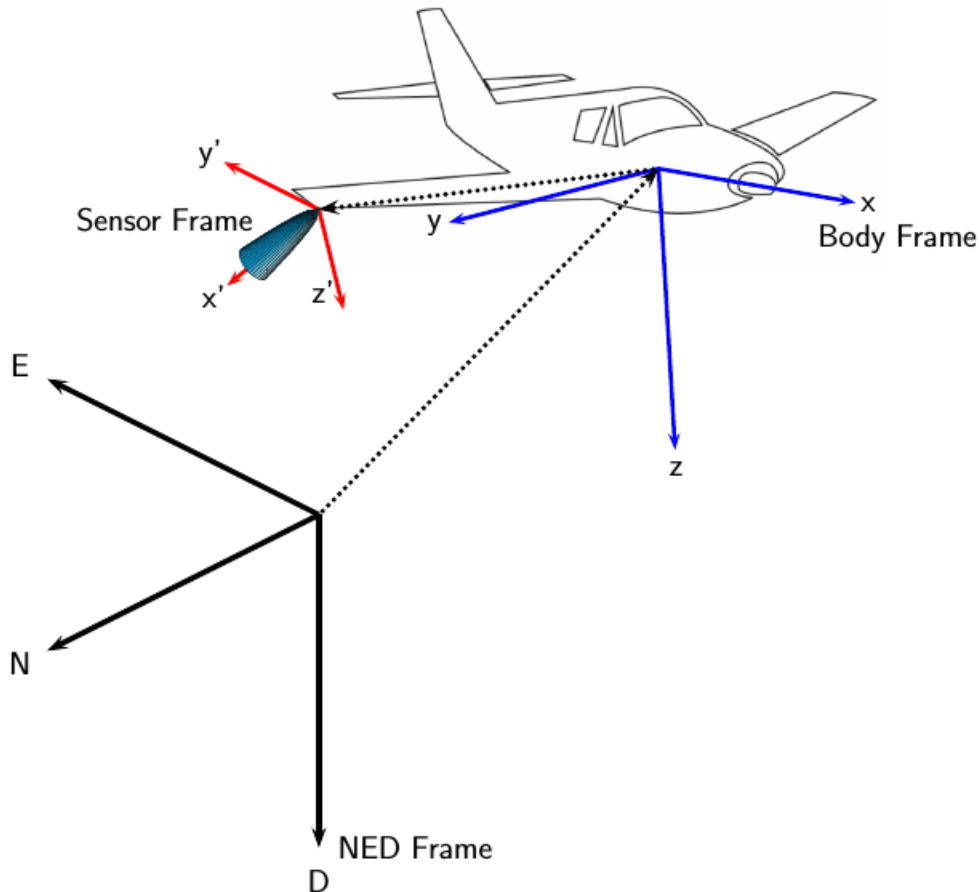
Additional coordinate frames can be required. For example, often tracks are not maintained in NED (or ENU) coordinates, as this coordinate frame changes based on the latitude and longitude where it is defined. For scenarios that cover large areas (over 100 kilometers in each dimension), earth-centered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

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

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

INS

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

Detections

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

objectDetection Structure

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

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

Measurement Coordinates

DetectionCoordinates	Measurement and Measurement Noise Coordinates		
`'Scenario'`	**Coordinate Dependence on HasRangeRate**		
`'Body'`	**HasRangeRate**	**Coordinates**	
`'Sensor rectangular'`	true	`[x;y;z;vx;vy;vz]`	
	false	`[x;y;z]`	
`'Sensor spherical'`	**Coordinate Dependence on HasRangeRate and HasElevation**		
	HasRangeRate	**HasElevation**	**Coordinates**
	true	true	`[az;el;rng;rr]`
	true	false	`[az;rng;rr]`
	false	true	`[az;el;rng]`
	false	false	`[az;rng]`

The MeasurementParameters field consists of an array of structs describing a sequence of coordinate transformations from a child frame to a parent frame or the inverse transformations (see "Frame Rotation"). The longest possible sequence of transformations is: Sensor → Platform → Scenario. For example, if the detections are reported in sensor spherical coordinates and HasINS is set to false, then the sequence consists of one transformation from sensor to platform. If HasINS is true, the sequence of transformations consists of two transformations – first to platform coordinates then to scenario coordinates. Trivially, if the detections are reported in platform rectangular coordinates and HasINS is set to false, the transformation consists only of the identity.

Each struct takes the form:

MeasurementParameters

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

ObjectAttributes

Attribute	Definition
`TargetIndex`	Identifier of the platform, `PlatformID`, that generated the detection. For false alarms, this value is negative.
SNR	Detection signal-to-noise ratio in dB.

Introduction to Radar for Object Tracking

Sensor Overview

In a tracking system, sensors are used to generate measurements or detections from targets in an environment. Sensors generally have an aperture by which they intercept the energy that the targets either emit or reflect. Sensors primarily use the intercepted energy to obtain information about the target's state and attributes.

A sensor is an active sensor if the intercepted energy originates from itself, such as a monostatic radar or a monostatic sonar. A sensor is a passive sensor if the intercepted energy originates from an outer source, such as an infra-red (IR) sensor, which receives radiated energy from a target.

Other than receiving targets' energy, the aperture inevitably also collects interfering energy created either by nature (such as background clutter) or by man (such as jamming). Therefore, sensors' detection quality is a complex problem involving many factors such as accuracy, resolution, bias, and false alarms. Also, sensors' detectability that relies on factors such as scanning limits, field of view, and sensor mounting is essential to consider in tracking system design.

In this introduction, the radar (including `radarSensor`, `monostaticRadarSensor`, and `radarEmitter`) is mainly discussed. However, some of the following descriptions also apply to other types of sensors (including `irSensor`, `sonarSensor`, and `sonarEmitter`).

Radar Detection Mode

Radar uses radio wave reflected or emitted from a target to detect the target. Given different transmitter and receiver configurations, a radar can have three detection modes: monostatic, bistatic, and electronic support measures (ESM).

(a) Monostatic   (b) Bistatic   (c) ESM
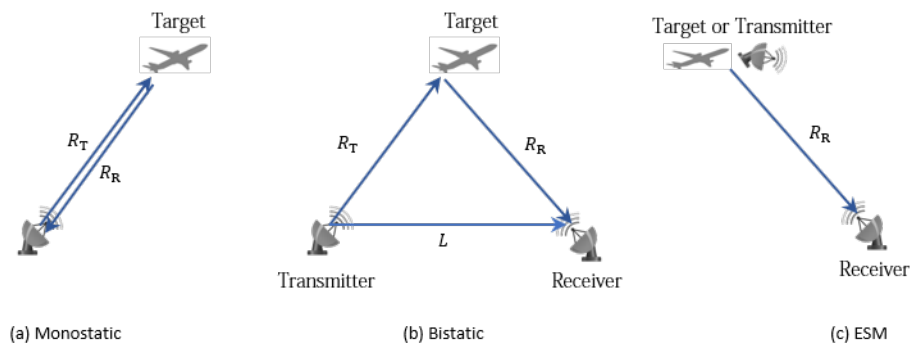
For the monostatic detection mode, the transmitter and the receiver are collocated, as shown in figure (a). In this mode, the range measurement R can be expressed as $R = R_T = R_R$, where R_T and R_R are the ranges from the transmitter to the target and from the target to the receiver, respectively. In this mode, the range measurement is $R = ct/2$, where c is the speed of light and t is the total elapsed time of the signal transmission. Other than the range measurement, a monostatic sensor can also optionally report range rate, azimuth, and elevation measurements of the target.

For the bistatic detection mode, the transmitter and the receiver are separated by a distance L. As shown in figure (b), the signal is emitted from the transmitter, reflected from the target, and eventually received by the receiver. The bistatic range measurement R_b is defined as $R_b = R_T + R_R -$

L. In the radar sensor, the bistatic range measurement is obtained by $R_b = c\Delta t$, where Δt is the time difference between the receiver receiving the direct signal from the transmitter and receiving the reflected signal from the target. Other than the bistatic range measurement, a bistatic sensor can also optionally report bistatic range rate, azimuth, and elevation measurements of the target. Since the bistatic range and the two bearing angles (azimuth and elevation) do not correspond to the same position vector, they cannot be combined into a position vector and reported in a Cartesian coordinate system. As a result, the measurements of a bistatic sensor can only be reported in a spherical coordinate system without additional information.

For the ESM detection mode, the receiver can only receive a signal reflected from the target or directly emitted from the transmitter, as shown in figure (c). Therefore, the only available measurements are azimuth and elevation of the target or transmitter. These measurements can only be reported in a spherical coordinate system.

Mounting Radar on Platform

To interpret the detection generated by the radar, you need to understand how the radar is mounted on the platform. Usually, the sensor mounting frame (*M*) origin is offset by a displacement from the platform frame origin (*P*). The displacement vector is usually specified by the `MountingLocation` property of various sensor objects such as `radarSensor`. Moreover, the sensor mounting frame usually has an angle displacement from the platform frame. You can specify this angle displacement represented by three rotation angles (in z-y-x sequence) using the `MountingAngles` property. Initially, the radar scanning frame (*S*) is aligned with its mounting frame. However, when the radar starts scanning, the radar can scan around the mounting frame's z- and y-axes. The x-direction of the radar scanning frame is aligned with the radar's current boresight direction.
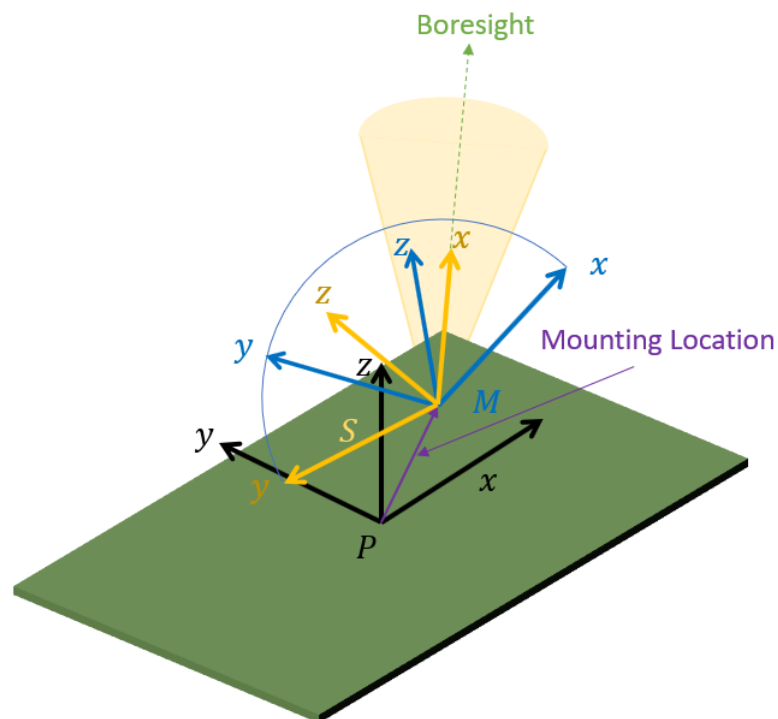
Detection Ability and Quality

In most cases, sensor is working in a scanning mode, in which the sensor beams sweep back and forth (with width equal to field of view) through a space region defined by the sensor scanning limit. The speed of the sweeping is specified by the `UpdateRate` property of the sensor object. The scanning speed can be obtained by the sensor's field of view and update rate. For example, if the sensor's update rate is 20Hz and its field of view is two degrees, then the sensor sweeps through 40 degrees per second.
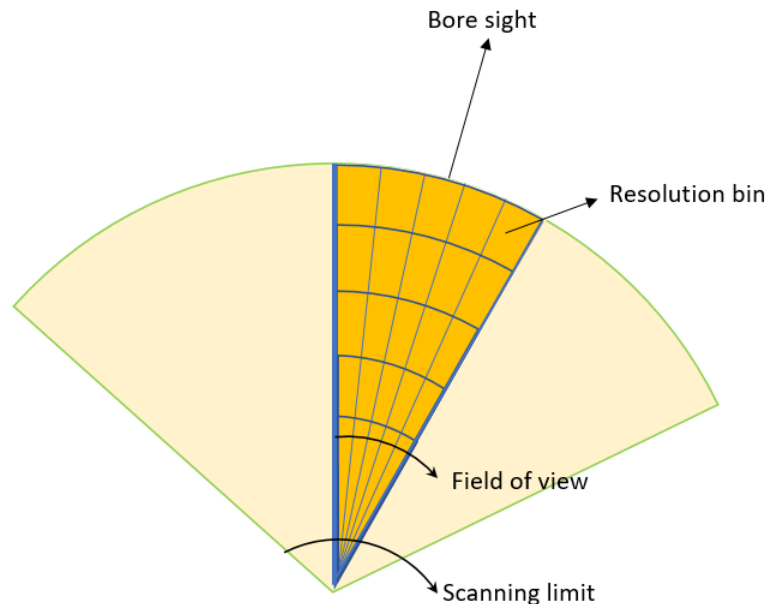
Sensor resolution defines sensor's ability to distinguish between two targets. In 3-D space, the radar's resolution bin is formed by azimuth boundary, elevation boundary, and range boundary. If two targets fall within the same resolution bin, then the radar cannot distinguish them and report them as one target in the detection.

Sensor's accuracy is usually captured by the standard deviation of the measurement error. The accuracy is mainly affected by two factors: the target's SNR and sensor's detection bias. SNR is defined as the ratio of reflected signal power to the noise power in decibels (dB). A ratio higher than 1:1 (greater than 0 dB) indicates more signal than noise. Larger SNR results in small measurement error and higher accuracy. For radar, SNR is usually a function of target's RCS (radar cross section). Sensor's bias is mainly due to imperfect alignment or calibration and often assumed to be a constant value. In each sensor object, you can specify sensor's bias as a fraction of the sensor resolution bin size using properties such as `AzimuthBiasFraction`. The larger the bias is, the more errors are incorporated in the detection.

Sensor can also make wrong assessment of the surveillance region. False alarm rate (p_{FA}) represents the probability that the sensor reports a detection on a resolution bin even though the resolution bin is not occupied by a target. Probability of detection (P_D) represents the probability that the sensor reports a detection on a resolution bin if the resolution bin is actually occupied by a target. Therefore, $1 - P_D$ represents the probability that the target is not detected by a sensor. P_D is usually a function of the target's SNR and the sensor's p_{FA}.

In many cases, the radar has a maximum range and range-rate limit. If the distance between a target and the sensor is above the maximum range, then the sensor truncates the detected range up to the

maximum range. For example, assume the target's range L_t is larger than the maximum range L_{max}, then the reported range of the target is $\text{mod}(L_t, L_{max})$, where `mod` is the remainder after division function in MATLAB. In the radar object, you can disable this limitation by setting the `HasMaxUnabmiguousRange` property as `false`.

Measurement and Detection Format

In terms of tracking systems, there are two basic classes of measurements: kinematic and attribute. Kinematic measurements provide the tracking systems with information about the existence and location of the target. Typical kinematic measurements include range, range rate, azimuth, and elevation. Attribute measurements usually contain targets' identification and characteristics, such as shape and reflectivity. The kinematic measurements for radar are described here.

In general, radar can report kinematic measurements in spherical or Cartesian coordinate frames. For spherical coordinates, the measurements can contain azimuth, elevation, range, and range rate. For Cartesian coordinates, the measurement can contain 2-D or 3-D position and velocity based on the setup. Since each radar detection mode can only output certain type of measurements, the available detection coordinates for each detection mode are:

- For monostatic detection mode, the detection can be reported in the spherical or Cartesian coordinate frames.
- For bistatic detection mode, the detection can only be reported in the spherical coordinate frame and the reported range is the bistatic range to the target.
- For ESM detection mode, the detection can only be reported in the spherical coordinate frame.

In Sensor Fusion and Tracking Toolbox, sensor objects output detections in the format of `objectDetection` object. An `objectDetection` object contains these properties:

Property	Definition
Time	Detection time
Measurement	Object measurements
Measurement Noise	Measure noise covariance matrix
SensorIndex	Unique ID of the sensor
ObjectClassID	Unique ID for object classification
MeasurementParameters	Parameters used to interpret the measurement such as sensor configuration and detection frame information
ObjectAttributes	Additional information of the target such as target ID and target RCS

Note that the `MeasurementParameters` property contains essential information used to interpret the measurements, such as sensor's pose (position, velocity, and orientation) and measurement coordinate frame information at the time of detection. For more details, see "Object Detections" and the "Convert Detections to objectDetection Format" on page 5-469 example.

Inertial Sensor and Sensor Fusion

Choose Inertial Sensor Fusion Filters

The toolbox provides multiple filters to estimate the pose and velocity of platforms by using on-board inertial sensors (including accelerometer, gyroscope, and altimeter), magnetometer, GPS, and visual odometry measurements. Each filter can process certain types of measurements from certain sensors. Each filter also makes assumptions and may have limitations that you should consider carefully before applying it. For example, many filters assume no sustained linear or angular acceleration other than the gravitational acceleration. Therefore, you should avoid using them during strong and constant acceleration, but these filters can perform reasonably well during short linear acceleration bursts. Also, some filters allow piecewise constant linear acceleration and angular velocity since they allow acceleration and angular velocity inputs during the prediction step.

The internal algorithms of these filters also vary greatly. For example, the `ecompass` object uses the TRIAD method to determine the orientation of the platform with very low computation cost. Many filters (such as `ahrsfilter` and `imufilter`) adopt the error-state Kalman filter, in which the state deviation from the reference state is estimated. Meanwhile, other filters (such as `insfilterMARG` and `insfilterAsync`) use the extended Kalman filter approach, in which the state is estimated directly.

To achieve high estimation accuracy, it is important to tune the filter properties and parameters properly. The toolbox offers the built-in `tune` function to tune parameters and sensor noise for the `imufilter`, `ahrsfilter`, and `insfilterAsync` directly.

The table lists the inputs, outputs, assumptions, and algorithms for all the inertial sensor fusion filters.

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used
ecompass	• Accelerometer • Magnetometer	Orientation	The filter assumes no sustained linear and angular acceleration other than gravitational acceleration.	TRIAD method
ahrsfilter	• Accelerometer • Gyroscope • Magnetometer	Orientation and angular velocity	The filter assumes no sustained linear and angular acceleration other than gravitational acceleration.	Error-state Kalman filter
ahrs10filter	• Accelerometer • Gyroscope • Magnetometer • Altimeter	Orientation, altitude, vertical velocity, delta angle bias, delta velocity bias, geomagnetic field vector, magnetometer bias	The filter assumes piecewise constant linear acceleration in the vertical direction, and no sustained linear and angular acceleration other than gravitational acceleration in other directions.	Discrete extended Kalman filter

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used
imufilter	• Accelerometer • Gyroscope	Orientation and angular velocity	The filter assumes no sustained linear and angular acceleration other than gravitational acceleration.	Error-state Kalman filter
complementaryFilter	• Accelerometer • Gyroscope • Magnetometer(optional)	Orientation and angular velocity	The filter assumes no sustained linear and angular acceleration other than gravitational acceleration.	Non-Kalman filter based approach: • Use high and low pass filters to reduce noise in various sensor readings. • Fuse the filtered sensor readings based on their assigned weights.
insfilterMARG	• Accelerometer • Gyroscope • Magnetometer • GPS	Orientation, position, velocity, delta angle bias, delta velocity bias, geomagnetic field vector, magnetometer bias	The prediction step takes the accelerometer and gyroscope inputs. Therefore, the filter assumes: • Piecewise constant linear acceleration. • Piecewise constant angular velocity. • Accelerometer and gyroscope run at the same rate with no sample dropping.	Discrete extended Kalman filter

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used
`insfilterAsync`	• Accelerometer • Gyroscope • Magnetometer • GPS	Orientation, angular velocity, position, velocity, acceleration, accelerometer bias, gyroscope bias, geomagnetic field vector, magnetometer bias	The filter assumes: • Constant angular velocity • Constant acceleration The filter does not require the sensors to be synchronous and each sensor can have sample dropping.	Continuous discrete extended Kalman Filter
`insfilterNonholonomic`	• Accelerometer • Gyroscope • GPS	Orientation, position, velocity, gyroscope bias, accelerometer bias	The prediction step takes the accelerometer and gyroscope inputs. Therefore, the filter assumes: • Piece-wise constant linear acceleration. • Piece-wise constant angular velocity. • Accelerometer and gyroscope run at the same rate with no sample dropping. Also, the filter assumes the platform moves forward without side slip.	Discrete extended Kalman Filter

Object	Sensors and Inputs	States and Outputs	Assumptions or Limitations	Algorithm Used
`insfilterErrorState`	• Accelerometer • Gyroscope • Magnetometer • GPS • Visual odometry scale	Orientation, position, velocity, gyroscope bias, accelerometer bias, and visual odometry scale	The prediction step takes the accelerometer and gyroscope inputs. Therefore, the filter assumes: • Piece-wise constant linear acceleration. • Piece-wise constant angular velocity. • Accelerometer and gyroscope run at the same rate with no sample dropping.	Error-state Kalman filter

See Also
`tunerconfig`

Multi-Object Tracking

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

Tracking and Tracking Filters

Multi-Object Tracking

You can use multi-sensor, multi-target trackers, `trackerGNN`, `trackerJPDA`, and `trackerTOMHT`, to track multiple targets. These trackers implement the multi-object tracking problem using the measurement-to-track association approach. Tracks are initiated and updated using sensor detections of targets. Trackers take several steps when new detections are made:

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

When tracking multiple objects using these trackers, there are several things to consider:

- Decide which tracker to use.

  - `trackerGNN` uses a global nearest-neighbor assignment algorithm, which maintains a single hypothesis about the tracked object. The tracker offers low computation cost but is not robust during ambiguous association events.
  - `trackerTOMHT` assigns detections based on a track-oriented, multi-hypothesis approach, which maintains multiple hypotheses about the tracked object. The tracker is robust during ambiguous data association events but is computationally more expensive.
  - `trackerJPDA` uses a joint probabilistic data association approach, which applies a soft assignment where multiple detections can contribute to each track. The tracker balances the robustness and computation cost between `trackerGNN` and `trackerTOMHT`.

  See the "Tracking Closely Spaced Targets Under Ambiguity" on page 5-167 example for a comparison between these three trackers.

- Decide which type of tracking filter to use.

  The choice of tracking filter depends on the expected dynamics of the object you want to track. The toolbox provides multiple Kalman filters including the Linear Kalman filter, `trackingKF`, the Extended Kalman filter, `trackingEKF`, the Unscented Kalman filter, `trackingUKF`, and the Cubature Kalman filter, `trackingCKF`. The linear Kalman filter is used when the dynamics of the object follow a linear model and the measurements are linear functions of the state vector. The extended, unscented, and cubature Kalman filters are used when the dynamics are nonlinear, the measurement model is nonlinear, or both. The toolbox also provides non-Gaussian filters such as the particle filter, `trackingPF`, Gaussian-sum filter, `trackingGSF`, and the Interacting Multiple Model (IMM) filter, `trackingIMM`. See the "Tracking with Range-Only Measurements" on page 5-284 and "Tracking Maneuvering Targets" on page 5-192 examples for more information about these filters.

  You can set the type of filter by specifying the `FilterInitializationFcn` property of a tracker. For example, if you set the `FilterInitializationFcn` property to `@initcaekf`, then the tracker uses the `initcaekf` function to create a constant-acceleration extended Kalman filter for a new track generated from detections.

- Decide which track logic to use.

  You can specify the conditions under which a track is confirmed or deleted by setting the `TrackLogic` property. Three algorithms are supported:

  - `'History'` — Track confirmation and deletion are based on the number of times the track has been assigned to a detection in the last several tracker updates. You can use this logic with `trackerGNN` and `trackerJPDA`.
  - `'Score'` — Track confirmation and deletion are based on a log-likelihood computation. A high score means that the track is more likely to be valid. A low score means that the track is more likely to be false. You can use this logic with `trackerGNN` and `trackerTOMHT`.
  - `'Integrated'` — Track confirmation and deletion are based on the probability of track existence. You can use this logic with `trackerJPDA`.

  For more details, see the "Introduction to Track Logic" on page 5-76 example.

You can also use a multi-sensor, multi-target tracker, `trackerPHD`, to track multiple targets simultaneously. `trackerPHD` approaches the multi-object tracking problem using the random finite set (RFS) method and tracks the probability hypothesis density (PHD) of a scenario. `trackerPHD` extracts peaks from the PHD-intensity to represent potential targets and maintain identities of targets by assigning a label to each component. The toolbox offers one realization of PHD, `ggiwphd`, which represents the PHD of extended targets using a Gamma Gaussian Inverse-Wishart (GGIW) target-state model. You can represent the configurations of sensors for `trackerPHD` using `trackingSensorConfiguration`.

Multi-Object Tracker Properties

trackerGNN Properties

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

trackerGNN Properties

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

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

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

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

trackerGNN Input

The input to the `trackerGNN` consists of a list of detections, the update time, cost matrix, and other data. Detections are specified as a cell array of `objectDetection` objects (see "Detections" on page 2-15). The input arguments are listed here.

trackerGNN Input

tracker	A `trackerGNN` object.
detections	Cell array of `objectDetection` objects (see "Detections" on page 2-15).
time	Time to which all the tracks are to be updated and predicted. The time at this execution step must be greater than the value in the previous call.
costmatrix	Cost matrix for assigning detections to tracks. A real *T*-by-*D* matrix, where *T* is the number of tracks listed in the `allTracks` argument returned from the previous call to `step`. *D* is the number of detections that are input in the current call. A larger cost matrix entry means a lower likelihood of assignment.
detectableTrackIDs	IDs of tracks that the sensors expect to detect, specified as an *M*-by-1 or *M*-by-2 matrix. The first column consists of track IDs, as reported in the `TrackID` field of the tracker output. The second column is optional and allows you to add the detection probability for each track.

trackerGNN Output

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

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

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

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

The fields contained in the `struct` are:

trackerGNN Output `struct`

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

Introduction to Estimation Filters

Background

Estimation Systems

For many autonomous systems, the knowledge of the system state is a prerequisite for designing any applications. In reality, however, the state is often not directly obtainable. The system state is usually inferred or estimated based on the system outputs measured by certain instruments (such as sensors) and the flow of the state governed by a dynamic or motion model. Some simple techniques, such as least square estimation or batch estimation, are sufficient in solving static or offline estimation problems. For online and real time (sequential) estimation problems, more sophisticated estimation filters are usually applied.

An estimation system is composed of a dynamic or motion model that describes the flow of the state and a measurement model that describes how the measurements are obtained. Mathematically, these two models can be represented by an equation of motion and a measurement equation. For example, the equation of motion and measurement equation for a general nonlinear discrete estimation system can be written as:

$$x_{k+1} = f(x_k)$$

$$y_k = h(x_k)$$

where k is the time step, x_k is the system state at time step k, $f(x_k)$ is the state-dependent equation of motion, $h(x_k)$ is the state dependent measurement equation, and y_k is the output.

Noise Distribution

In most cases, building a perfect model to capture all the dynamic phenomenon is not possible. For example, including all frictions in the motion model of an autonomous vehicle is impossible. To compensate for these unmodeled dynamics, process noise (w) is often added to the dynamic model. Moreover, when measurements are taken, multiple sources of errors, such as calibration errors, are inevitably included in the measurements. To account for these errors, proper measurement noise must be added to the measurement model. An estimation system including these random noises and errors is called a stochastic estimation system, which can be represented by:

$$x_{k+1} = f(x_k, w_k)$$

$$y_k = h(x_k, v_k)$$

where w_k and v_k represent process noise and measurement noise, respectively.

For most engineering applications, the process noise and measurement noise are assumed to follow zero-mean Gaussian or normal distributions, or are at least be approximated by Gaussian distributions. Also, because the exact state is unknown, the state estimate is a random variable, usually assumed to follow Gaussian distributions. Assuming Gaussian distributions for these variables greatly simplifies the design of an estimation filter, and form the basis of the Kalman filter family.

A Gaussian distribution for a random variable (x) is parametrized by a mean value μ and a covariance matrix P, which is written as $x \sim N(\mu, P)$. Given a Gaussian distribution, the mean, which is also the most likely value of x, is defined by expectation (E) as:

$$\mu = E[x]$$

The mean is also called the first moment of x about the origin. The covariance that describes of the uncertainty of x is defined by expectation (E) as:

$$P = E\left[(x - \mu)(x - \mu)^T\right]$$

The covariance is also called the second moment of x about its mean.

If the dimension of x is one, P is only a scalar. In this case, the value of P is usually denoted by σ^2 and called variance. The square root, σ, is called the standard deviation of x. The standard deviation has important physical meaning. For example, the following figure shows the probability density function (which describes the likelihood that x takes a certain value) for a one-dimensional Gaussian distribution with mean equal to μ and standard deviation equal to σ. About 68% of the data fall within the 1σ boundary of x, 95% of the data fall within the 2σ boundary, and 99.7% of the data fall within the 3σ boundary.

Gaussian Distribution
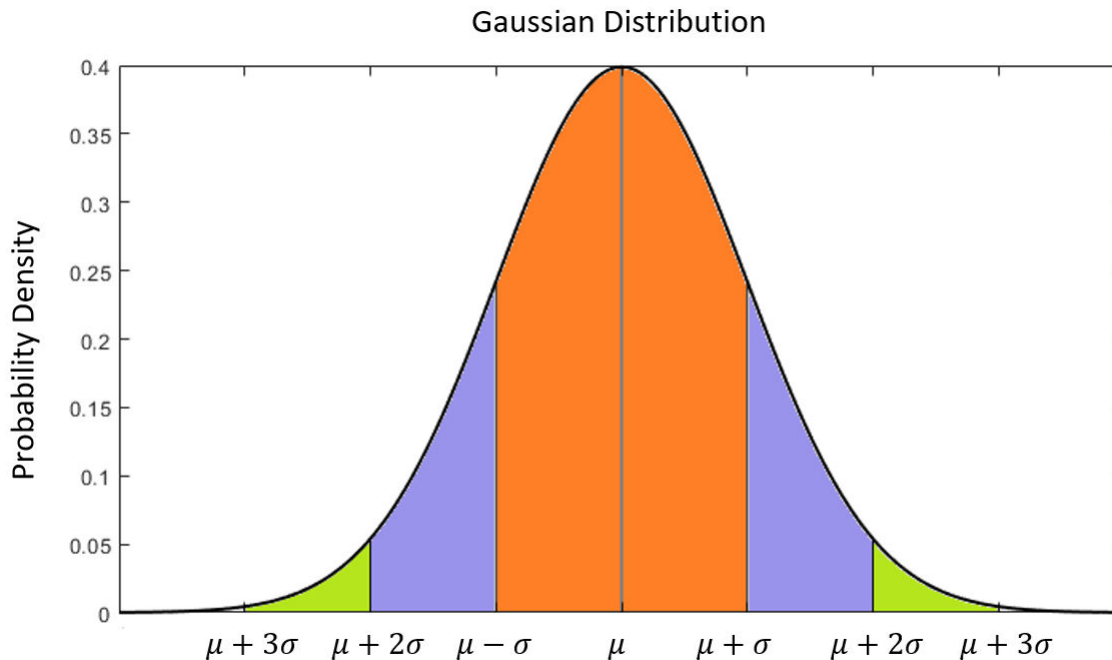
Even though the Gaussian distribution assumption is the dominant assumption in engineering applications, there exist systems whose state cannot be approximated by Gaussian distributions. In this case, non-Kalman filters (such as a particle filter) is required to accurately estimate the system state.

Filter Design

The goal of designing a filter is to estimate the state of a system using measurements and system dynamics. Since the measurements are usually taken at discrete time steps, the filtering process is usually separated into two steps:

1 Prediction: Propagate state and covariance between discrete measurement time steps (k = 1, 2, 3, …, N) using dynamic models. This step is also called flow update.

2 Correction: Correct the state estimate and covariance at discrete time steps using measurements. This step is also called measurement update.

For representing state estimate and covariance status in different steps, $x_{k|k}$ and $P_{k|k}$ denote the state estimate and covariance after correction at time step k, whereas $x_{k+1|k}$ and $P_{k+1|k}$ denote the state estimate and covariance predicted from the previous time step k to the current time step $k+1$.

Prediction

In the prediction step, the state propagation is straightforward. The filter only needs to substitute the state estimate into the dynamic model and propagate it forward in time as $x_{k+1|k} = f(x_{k|k})$.

The covariance propagation is more complicated. If the estimation system is linear, then the covariance can be propagated ($P_{k|k} \rightarrow P_{k+1|k}$) exactly in a standard equation based on the system properties. For nonlinear systems, accurate covariance propagation is challenging. A major difference between different filters is how they propagate the system covariance. For example:

- A linear Kalman filter uses a linear equation to exactly propagate the covariance.
- An extended Kalman filter propagates the covariance based on linear approximation, which renders large errors when the system is highly nonlinear.
- An unscented Kalman filter uses unscented transformation to sample the covariance distribution and propagate it in time.

How the state and covariance are propagated also greatly affects the computation complexity of a filter. For example:

- A linear Kalman filter uses a linear equation to exactly propagate the covariance, which is usually computationally efficient.
- An extended Kalman filter uses linear approximations, which require calculation of Jacobian matrices and demand more computation resources.
- An unscented Kalman filter needs to sample the covariance distribution and therefore requires the propagation of multiple sample points, which is costly for high-dimensional systems.

Correction

In the correction step, the filter uses measurements to correct the state estimate through measurement feedback. Basically, the difference between the true measurement and the predicted measurement is added to the state estimate after it is multiplied by a feedback gain matrix. For example, in an extended Kalman filter, the correction for the state estimate is given by:

$$x_{k+1|k+1} = x_{k+1|k} + K_k(y_{k+1} - h(x_{k+1|k}))$$

As mentioned, $x_{k+1|k}$ is the state estimate before (priori) correction and $x_{k+1|k+1}$ is the state estimate after (posteriori) correction. K_k is the Kalman gain governed by an optimal criterion, y_k is the true measurement, and $h(x_{k+1|k})$ is the predicted measurement.

In the correction step, the filter also corrects the estimate error covariance. The basic idea is to correct the probabilistic distribution of x using the distribution information of y_{k+1}. This is called the posterior probability density of x given y. In a filter, the prediction and correction steps are processed recursively. The flowchart shows the general algorithms for Kalman filters.

General Algorithms for Kalman Filters
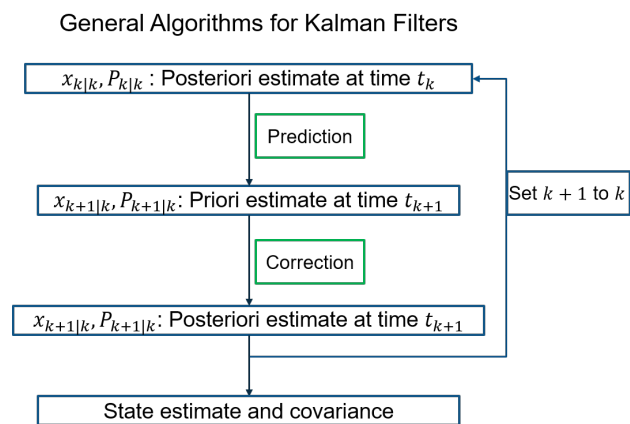
Estimation Filters in Sensor Fusion and Tracking Toolbox

Sensor Fusion and Tracking Toolbox offers multiple estimation filters you can use to estimate and track the state of a dynamic system.

Kalman Filter

The classical Kalman filter (`trackingKF`) is the optimal filter for linear systems with Gaussian process and measurement noise. A linear estimation system can be given as:

$$x_{k+1} = A_k x_k + w_k$$

$$y_k = H_k x_k + v_k$$

Both the process and measurement noise are assumed to be Gaussian, that is:

$$w_k \sim N(0, Q_k)$$

$$v_k \sim N(0, R_k)$$

Therefore, the covariance matrix can be directly propagated between measurement steps using a linear algebraic equation as:

$$P_{k+1|k} = A_k P_{k|k} A_k^T + Q_k$$

The correction equations for the measurement update are:

$$x_{k+1|k+1} = x_{k+1|k} + K_k(y_k - H_k x_{k+1|k})$$

$$P_{k+1|k+1} = (I - K_k H_k)P_{k+1|k}$$

To calculate the Kalman gain matrix (K_k) in each update, the filter needs to calculate the inverse of a matrix:

$$K_k = P_{k|k-1} H_k^T \left[H_k P_{k|k-1} H_k^T + R_k \right]^{-1}$$

Since the dimension of the inverted matrix is equal to that of the estimated state, this calculation requires some computation efforts for a high dimensional system. For more details, see "Linear Kalman Filters" on page 4-20.

Alpha-Beta Filter

The alpha-beta filter (`trackingABF`) is a suboptimal filter applied to linear systems. The filter can be regarded as a simplified Kalman filter. In a Kalman filter, the Kalman gain and covariance matrices are calculated dynamically and updated in each step. However, in an alpha-beta filter, these matrices are constant. This treatment sacrifices the optimality of a Kalman filter but improves the computation efficiency. For this reason, an alpha-beta filter might be preferred when the computation resources are limited.

Extended Kalman Filter

The most popular extended Kalman filter (`trackingEKF`) is modified from the classical Kalman filter to adapt to the nonlinear models. It works by linearizing the nonlinear system about the state estimate and neglecting the second and higher order nonlinear terms. Its formulations are basically the same as those of a linear Kalman filter except that the A_k and H_k matrices in the Kalman filter are replaced by the Jacobian matrices of $f(x_k)$ and $h(x_k)$:

$$A_k = \left. \frac{\partial f(x_k)}{\partial x_k} \right|_{x_{k|k-1}}$$

$$H_k = \left. \frac{\partial h(x_k)}{\partial x_k} \right|_{x_{k|k-1}}$$

If the true dynamics of the estimation system are close to the linearized dynamics, then using this linear approximation does not yield significant errors for a short period of time. For this reason, an EKF can produce relatively accurate state estimates for a mildly nonlinear estimation system with short update intervals. However, since an EKF neglects higher order terms, it can diverge for highly nonlinear systems (quadrotors, for example), especially with large update intervals.

Compared to a KF, an EKF needs to derive the Jacobian matrices, which requires the system dynamics to be differentiable, and to calculate the Jacobian matrices to linearize the system, which demands more computation assets.

Note that for estimation systems with state expressed in spherical coordinates, you can use `trackingMSCEKF`.

Unscented Kalman Filter

The unscented Kalman filter (`trackingUKF`) uses an unscented transformation (UT) to approximately propagate the covariance distribution for a nonlinear model. The UT approach samples the covariance Gaussian distribution at the current time, propagates the sample points (called sigma points) using the nonlinear model, and approximates the resulting covariance distribution assumed to be Gaussian by evaluating these propagated sigma points. The figure illustrates the difference between the actual propagation, the linearized propagation, and the UT propagation of the uncertainty covariance.
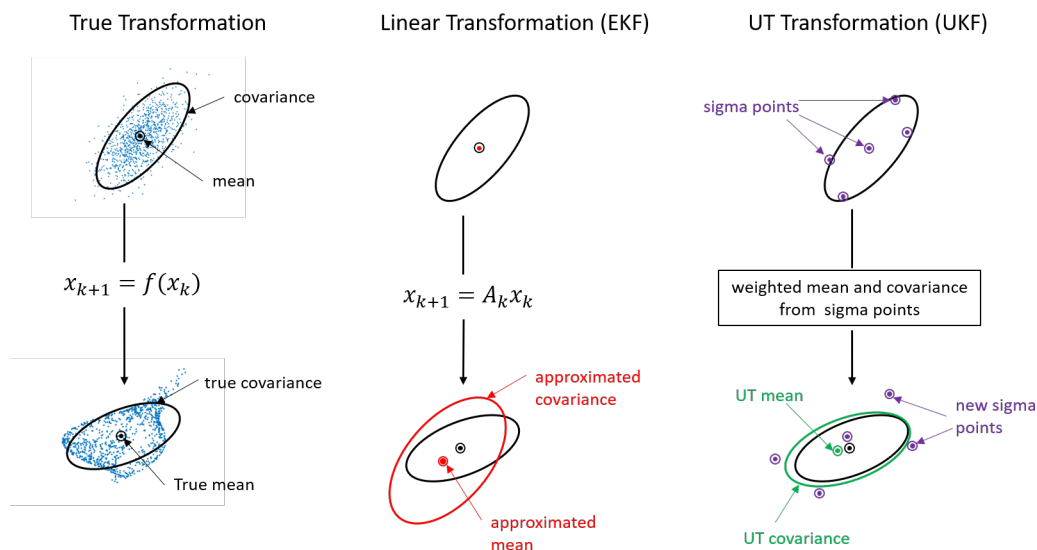
Compared to the linearization approach taken by an EKF, the UT approach results in more accurate propagation of covariance and leads to more accurate state estimation, especially for highly nonlinear systems. UKF does not require the derivation and calculation of Jacobian matrices. However, UKF requires the propagation of $2n+1$ sigma points through the nonlinear model, where n is the dimension of the estimated state. This can be computationally expensive for high dimensional systems.

Cubature Kalman Filter

The cubature Kalman filter (`trackingCKF`) takes a slightly different approach than UKF to generate $2n$ sample points used to propagate the covariance distribution, where n is the dimension of the estimated state. This alternate sample point set often results in better statistical stability and avoids divergence which might occur in UKF, especially when running in a single-precision platform. Note that a CKF is essentially equivalent to a UKF when the UKF parameters are set to $\alpha = 1$, $\beta = 0$, and $\kappa = 0$. See `trackingUKF` for the definition of these parameters.

Gaussian-Sum Filter

The Gaussian-Sum filter (`trackingGSF`) uses the weighted sum of multiple Gaussian distributions to approximate the distribution of the estimated state. The estimated state is given by a weighted sum of Gaussian states:

$$x_k = \sum_{i=1}^{N} c_k^i x_k^i$$

where N is the number of Gaussian states maintained in the filter, and $c_k{}^i$ is the weight for the corresponding Gaussian state, which is modified in each update based on the measurements. The multiple Gaussian states follow the same dynamic model as:

$$x_{k+1}^i = f(x_k^i, w_k^i), \text{ for } i = 1, 2, ..., N.$$

The filter is effective in estimating the states of an incompletely observable estimation system. For example, the filter can use multiple angle-parametrized extended Kalman filters to estimate the system state when only range measurements are available. See "Tracking with Range-Only Measurements" on page 5-284 for an example.

Interactive Multiple Model Filter

The interactive multiple model filter (`trackingIMM`) uses multiple Gaussian filters to track the position of a target. In highly maneuverable systems, the system dynamics can switch between multiple models (constant velocity, constant acceleration, and constant turn for example). Modelling the motion of a target using only one motion model is difficult. A multiple model estimation system can be described as:

$$x_{k+1}^i = f_i(x_k^i, w_k^i)$$

$$y_k^i = h_i(x_k^i, v_k^i)$$

where $i = 1, 2, ..., M$, and M is the total number of dynamic models. The IMM filter resolves the target motion uncertainty by using multiple models for a maneuvering target. The filter processes all the models simultaneously and represents the overall estimate as the weighted sum of the estimates from these models, where the weights are the probability of each model. See "Tracking Maneuvering Targets" on page 5-192 for an example.

Particle Filter

The particle filter (`trackingPF`) is different from the Kalman family of filters (EKF and UKF, for example) as it does not rely on the Gaussian distribution assumption, which corresponds to a parametric description of uncertainties using mean and variance. Instead, the particle filter creates multiple simulations of weighted samples (particles) of a system's operation through time, and then analyzes these particles as a proxy for the unknown true distribution. A brief introduction of the particle filter algorithm is shown in the figure.

1. Weighted particles from initialization or last update. The weight is represented by the size of each particle.

2. Extract particles according to their weights.

3. Propagate the particle state one step forward using dynamic model.

4. Create multiple sub-particles according to the weight of the original particle following a certain distribution.

5. Resampling: Compare the predicted measurements given by these particles and the true measurements to update the particle weights.

6. New particles are obtained with weights.

t_k

t_{k+1}

The motivation behind this approach is a law-of-large-numbers argument — as the number of particles gets large, their empirical distribution gets close to the true distribution. The main advantage of a particle filter over various Kalman filters is that it can be applied to non-Gaussian

distributions. Also, the filter has no restriction on the system dynamics and can be used with highly nonlinear system. Another benefit is the filter's inherent ability to represent multiple hypotheses about the current state. Since each particle represents a hypothesis of the state with a certain associated likelihood, a particle filter is useful in cases where there exists ambiguity about the state.

Along with these appealing properties is the high computation complexity of a particle filter. For example, a UKF requires propagating 13 sample points to estimate the 3-D position and velocity of an object. However, a particle filter may require thousands of particles to obtain a reasonable estimate. Also, the number of particles needed to achieve good estimation grows very quickly with the state dimension and can lead to particle deprivation problems in high dimensional spaces. Therefore, particle filters have been mostly applied to systems with a reasonably low number of dimensions (for example robots).

How to Choose a Tracking Filter

The following table lists all the tracking filters available in Sensor Fusion and Tracking Toolbox and how to choose them given constraints on system nonlinearity, state distribution, and computational complexity.

Filter Name	Supports Nonlinear Models	Gaussian State	Computational Complexity	Comments
Alpha-Beta			Low	Suboptimal filter.
Kalman		✓	Medium Low	Optimal for linear systems.
Extended Kalman	✓	✓	Medium	Uses linearized models to propagate uncertainty covariance.
Unscented Kalman	✓	✓	Medium High	Samples the uncertainty covariance to propagate the sample points. May become numerically unstable in a single-precision platform.
Cubature Kalman	✓	✓	Medium High	Samples the uncertainty covariance to propagate the sample points. Numerically stable.

Gaussian-Sum	✓	✓ (Assumes a weighted sum of distributions)	High	Good for partially observable cases (angle-only tracking for example).
Interacting Multiple Models (IMM)	✓ Multiple models	✓ (Assumes a weighted sum of distributions)	High	Maneuvering objects (which accelerate or turn, for example)
Particle	✓		Very High	Samples the uncertainty distribution using weighted particles.

References

[1] Wang, E.A., and R. Van Der Merwe. "The Unscented Kalman Filter for Nonlinear Estimation." *IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium.* No. 00EX373, 2000, pp. 153–158.

[2] Fang, H., N. Tian, Y. Wang, M. Zhou, and M.A. Haile. "Nonlinear Bayesian Estimation: From Kalman Filtering to a Broader Horizon." *IEEE/CAA Journal of Automatica Sinica.* Vol. 5, Number 2, 2018, pp. 401–417.

[3] Arasaratnam, I., and S. Haykin. "Cubature Kalman Filters." *IEEE Transactions on automatic control.* Vol. 54, Number 6, 2009, pp. 1254–1269.

[4] Konatowski, S., P. Kaniewski, and J. Matuszewski. "Comparison of Estimation Accuracy of EKF, UKF and PF Filters." *Annual of Navigation.* Vol. 23, Number 1, 2016, pp. 69–87.

[5] Darko, J. "Object Tracking: Particle Filter with Ease." https://www.codeproject.com/Articles/865934/Object-Tracking-Particle-Filter-with-Ease.

Linear Kalman Filters

When you use a Kalman filter to track objects, you use a sequence of detections or measurements to construct a model of the object motion. Object motion is defined by the evolution of the state of the object. The Kalman filter is an optimal, recursive algorithm for estimating the track of an object. The filter is recursive because it updates the current state using the previous state, using measurements that may have been made in the interval. A Kalman filter incorporates these new measurements to keep the state estimate as accurate as possible. The filter is optimal because it minimizes the mean-square error of the state. You can use the filter to predict future states or estimate the current state or past state.

State Equations

For most types of objects tracked in Sensor Fusion and Tracking Toolbox, the state vector consists of one-, two- or three-dimensional positions and velocities.

Start with Newton equations for an object moving in the *x*-direction at constant acceleration and convert these equations to space-state form.

$$m\ddot{x} = f$$
$$\ddot{x} = \frac{f}{m} = a$$

If you define the state as

$$x_1 = x$$
$$x_2 = \dot{x},$$

you can write Newton's law in state-space form.

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}a$$

You use a linear dynamic model when you have confidence that the object follows this type of motion. Sometimes the model includes process noise to reflect uncertainty in the motion model. In this case, Newton's equations have an additional term.

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}a + \begin{bmatrix} 0 \\ 1 \end{bmatrix}v_k$$

v_k is the unknown noise perturbations of the acceleration. Only the statistics of the noise are known. It is assumed to be zero-mean Gaussian white noise.

You can extend this type of equation to more than one dimension. In two dimensions, the equation has the form

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} 0 \\ a_x \\ 0 \\ a_y \end{bmatrix} + \begin{bmatrix} 0 \\ v_x \\ 0 \\ v_y \end{bmatrix}$$

The 4-by-4 matrix on the right side is the state transition model matrix. For independent x- and y-motions, this matrix is block diagonal.

When you transition to discrete time, you integrate the equations of motion over the length of the time interval. In discrete form, for a sample interval of T, the state-representation becomes

$$\begin{bmatrix} x_{1,k+1} \\ x_{2,k+1} \end{bmatrix} = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}\begin{bmatrix} x_{1,k} \\ x_{2,k} \end{bmatrix} + \begin{bmatrix} 0 \\ T \end{bmatrix}a + \begin{bmatrix} 0 \\ 1 \end{bmatrix}\tilde{v}$$

The quantity x_{k+1} is the state at discrete time $k+1$, and x_k is the state at the earlier discrete time, k. If you include noise, the equation becomes more complicated, because the integration of noise is not straightforward.

The state equation can be generalized to

$$x_{k+1} = F_k x_k + G_k u_k + v_k$$

F_k is the state transition matrix and G_k is the control matrix. The control matrix takes into account any known forces acting on the object. Both of these matrices are given. The last term represents noise-like random perturbations of the dynamic model. The noise is assumed to be zero-mean Gaussian white noise.

Continuous-time systems with input noise are described by linear stochastic differential equations. Discrete-time systems with input noise are described by linear stochastic differential equations. A state-space representation is a mathematical model of a physical system where the inputs, outputs, and state variables are related by first-order coupled equations.

Measurement Models

Measurements are what you observe about your system. Measurements depend on the state vector but are not always the same as the state vector. For instance, in a radar system, the measurements can be spherical coordinates such as range, azimuth, and elevation, while the state vector is the Cartesian position and velocity. For the linear Kalman filter, the measurements are always linear functions of the state vector, ruling out spherical coordinates. To use spherical coordinates, use the extended Kalman filter.

The measurement model assumes that the actual measurement at any time is related to the current state by

$$z_k = H_k x_k + w_k$$

w_k represents measurement noise at the current time step. The measurement noise is also zero-mean white Gaussian noise with covariance matrix Q described by $Q_k = E[n_k n_k^T]$.

Linear Kalman Filter Equations

Without noise, the dynamic equations are

$$x_{k+1} = F_k x_k + G_k u_k.$$

Likewise, the measurement model has no measurement noise contribution. At each instance, the process and measurement noises are not known. Only the noise statistics are known. The

$$z_k = H_k x_k$$

You can put these equations into a recursive loop to estimate how the state evolves and also how the uncertainties in the state components evolve.

Filter Loop

Start with a best estimate of the state, $x_{0/0}$, and the state covariance, $P_{0/0}$. The filter performs these steps in a continual loop.

1   Propagate the state to the next step using the motion equations.

$$x_{k+1|k} = F_k x_{k|k} + G_k u_k.$$

Propagate the covariance matrix as well.

$$P_{k+1|k} = F_k P_{k|k} F_k^T + Q_k.$$

The subscript notation $k+1|k$ indicates that the quantity is the optimum estimate at the $k+1$ step propagated from step k. This estimate is often called the *a priori* estimate.

Then predict the measurement at the updated time.

$$z_{k+1|k} = H_{k+1} x_{k+1|k}$$

2   Use the difference between the actual measurement and predicted measurement to correct the state at the updated time. The correction requires computing the Kalman gain. To do this, first compute the measurement prediction covariance (innovation)

$$S_{k+1} = H_{k+1} P_{k+1|k} H_{k+1}^T + R_{k+1}$$

Then the Kalman gain is

$$K_{k+1} = P_{k+1|k} H_{k+1}^T S_{k+1}^{-1}$$

and is derived from using an optimality condition.

3   Correct the predicted estimate with the measurement. Assume that the estimate is a linear combination of the predicted state and the measurement. The estimate after correction uses the subscript notation, $k+1|k+1$. is computed from

$$x_{k+1|k+1} = x_{k+1|k} + K_{k+1}(z_{k+1} - z_{k+1|k})$$

where K_{k+1} is the Kalman gain. The corrected state is often called the *a posteriori* estimate of the state because it is derived after the measurement is included.

Correct the state covariance matrix

$$P_{k+1|k+1} = P_{k+1|k} - K_{k+1}S_{k+1}K'_{k+1}$$

Finally, you can compute a measurement based upon the corrected state. This is not a correction to the measurement but is a best estimate of what the measurement would be based upon the best estimate of the state. Comparing this to the actual measurement gives you an indication of the performance of the filter.

This figure summarizes the Kalman loop operations.
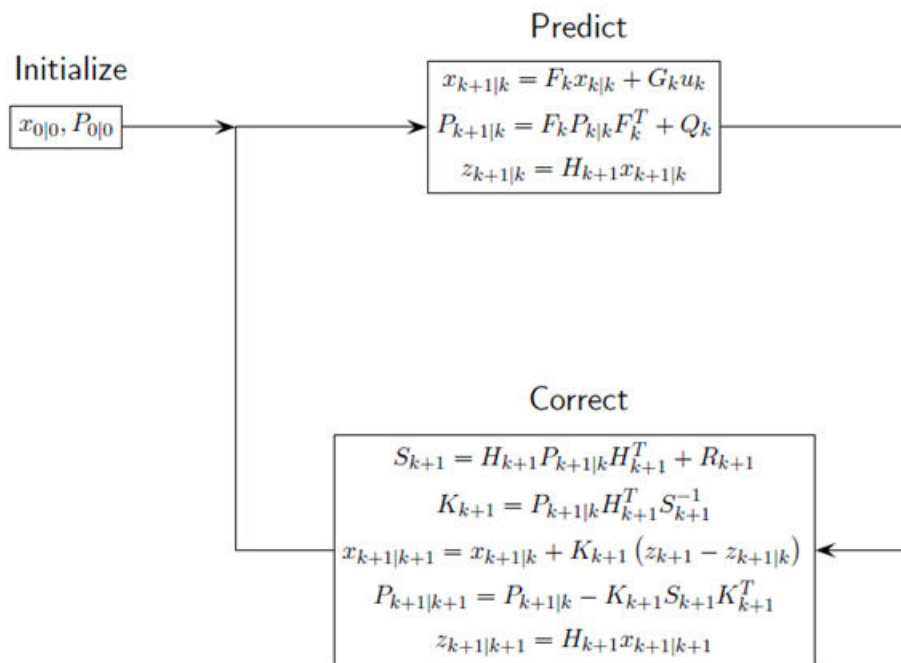
Constant Velocity Model

The linear Kalman filter contains a built-in linear constant-velocity motion model. Alternatively, you can specify the transition matrix for linear motion. The state update at the next time step is a linear function of the state at the present time. In this filter, the measurements are also linear functions of the state described by a measurement matrix. For an object moving in 3-D space, the state is described by position and velocity in the x-, y-, and z-coordinates. The state transition model for the constant-velocity motion is

$$
\begin{bmatrix} x_{k+1} \\ v_{x,k+1} \\ y_{k+1} \\ v_{y,k+1} \\ z_{k+1} \\ v_{z,k+1} \end{bmatrix} = \begin{bmatrix} 1 & T & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & T & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ y_k \\ v_{y,k} \\ z_k \\ v_{z,k} \end{bmatrix}
$$

The measurement model is a linear function of the state vector. The simplest case is one where the measurements are the position components of the state.

$$
\begin{bmatrix} m_{x,k} \\ m_{y,k} \\ m_{z,k} \end{bmatrix} = \begin{bmatrix} 1\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ 1\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0\ 1\ 0 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ y_k \\ v_{y,k} \\ z_k \\ v_{z,k} \end{bmatrix}
$$

Constant Acceleration Model

The linear Kalman filter contains a built-in linear constant-acceleration motion model. Alternatively, you can specify the transition matrix for constant-acceleration linear motion. The transition model for linear acceleration is

$$
\begin{bmatrix} x_{k+1} \\ v_{x,k+1} \\ a_{x,k+1} \\ y_{k+1} \\ v_{y,k+1} \\ a_{y,k+1} \\ z_{k+1} \\ v_{z,k+1} \\ a_{z,k+1} \end{bmatrix} = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & T & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ a_{x,k} \\ y_k \\ v_{y,k} \\ a_{y,k} \\ z_k \\ v_{z,k} \\ a_{z,k} \end{bmatrix}
$$

The simplest case is one where the measurements are the position components of the state.

$$
\begin{bmatrix} m_{x,k} \\ m_{y,k} \\ m_{z,k} \end{bmatrix} = \begin{bmatrix} 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ a_{x,k} \\ y_k \\ v_{y,k} \\ a_{y,k} \\ z_k \\ v_{z,k} \\ a_{y,k} \end{bmatrix}
$$

See Also

Objects
`trackingKF`

Extended Kalman Filters

In this section...

Use an extended Kalman filter when object motion follows a nonlinear state equation or when the measurements are nonlinear functions of the state. A simple example is when the state or measurements of the object are calculated in spherical coordinates, such as azimuth, elevation, and range.

State Update Model

The extended Kalman filter formulation linearizes the state equations. The updated state and covariance matrix remain linear functions of the previous state and covariance matrix. However, the state transition matrix in the linear Kalman filter is replaced by the Jacobian of the state equations. The Jacobian matrix is not constant but can depend on the state itself and time. To use the extended Kalman filter, you must specify both a state transition function and the Jacobian of the state transition function.

Assume there is a closed-form expression for the predicted state as a function of the previous state, controls, noise, and time.

$$x_{k+1} = f(x_k, u_k, w_k, t)$$

The Jacobian of the predicted state with respect to the previous state is

$$F^{(x)} = \frac{\partial f}{\partial x}.$$

The Jacobian of the predicted state with respect to the noise is

$$F^{(w)} = \frac{\partial f}{\partial w_i}.$$

These functions take simpler forms when the noise enters linearly into the state update equation:

$$x_{k+1} = f(x_k, u_k, t) + w_k$$

In this case, $F^{(w)} = 1_M$.

Measurement Model

In the extended Kalman filter, the measurement can be a nonlinear function of the state and the measurement noise.

$$z_k = h(x_k, v_k, t)$$

The Jacobian of the measurement with respect to the state is

$$H^{(x)} = \frac{\partial h}{\partial x}.$$

The Jacobian of the measurement with respect to the measurement noise is

$$H^{(v)} = \frac{\partial h}{\partial v}.$$

These functions take simpler forms when the noise enters linearly into the measurement equation:

$$z_k = h(x_k, t) + v_k$$

In this case, $H^{(v)} = 1_N$.

Extended Kalman Filter Loop

This extended kalman filter loop is almost identical to the linear Kalman filter loop except that:

- The exact nonlinear state update and measurement functions are used whenever possible and the state transition matrix is replaced by the state Jacobian
- The measurement matrices are replaced by the appropriate Jacobians.

Predict

Initialize

$$x_{0|0}, P_{0|0}$$

$$x_{k+1|k} = f(x_{k|k}, u^k)$$
$$P_{k+1|k} = F_k^{(x)} P_{k|k} F_k^{(x)T} + F^{(v)} Q_k F^{(v)T}$$
$$z_{k+1|k} = h(x_{k+1|k})$$

Correct

$$S_{k+1} = H_{k+1}^{(x)} P_{k+1|k} H_{k+1}^{(x)T} + H^{(w)} R_{k+1} H^{(w)T}$$
$$K_{k+1} = P_{k+1|k} H_{k+1}^T S_{k+1}^{-1}$$
$$x_{k+1|k+1} = x_{k+1|k} + K_{k+1} \left(z_{k+1} - z_{k+1|k} \right)$$
$$P_{k+1|k+1} = P_{k+1|k} - K_{k+1} S_{k+1} K_{k+1}^T$$
$$z_{k+1|k+1} = h(x_{k+1|k+1})$$
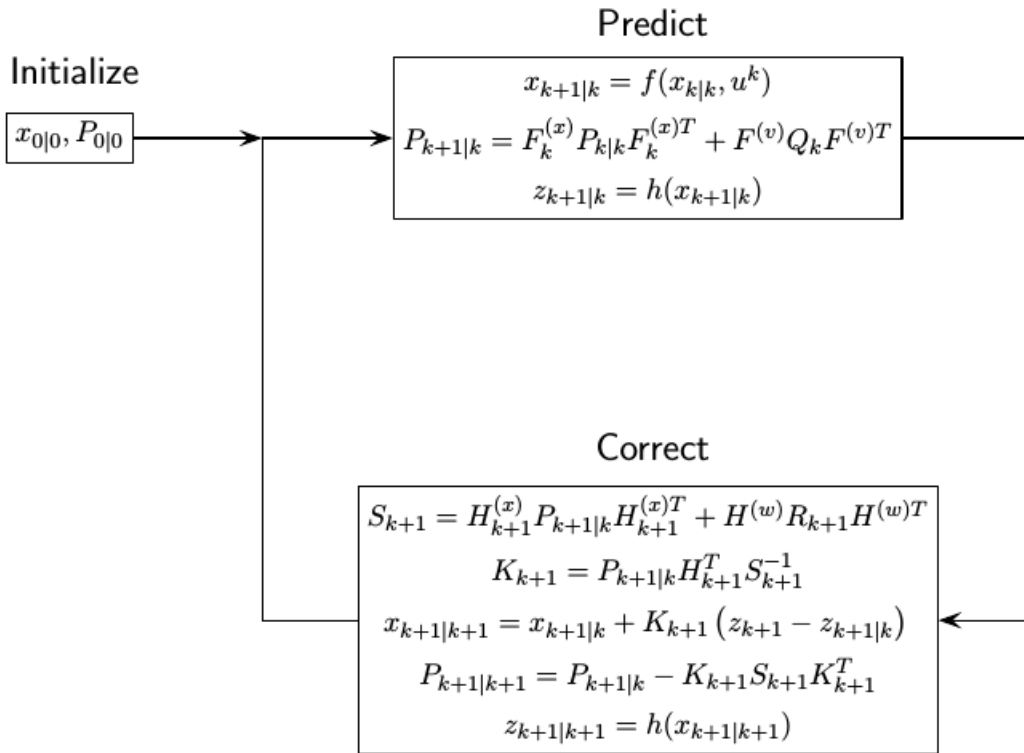
Predefined Extended Kalman Filter Functions

Sensor Fusion and Tracking Toolbox provides predefined state update and measurement functions to use in the extended Kalman filter.

Motion Model	Function Name	Function Purpose
Constant velocity	constvel	Constant-velocity state update model
	constveljac	Constant-velocity state update Jacobian
	cvmeas	Constant-velocity measurement model
	cvmeasjac	Constant-velocity measurement Jacobian
Constant acceleration	constacc	Constant-acceleration state update model
	constaccjac	Constant-acceleration state update Jacobian
	cameas	Constant-acceleration measurement model
	cameasjac	Constant-acceleration measurement Jacobian
Constant turn rate	constturn	Constant turn-rate state update model
	constturnjac	Constant turn-rate state update Jacobian
	ctmeas	Constant turn-rate measurement model
	ctmeasjac	Constant-turnrate measurement Jacobian

See Also

Objects
trackingEKF

Introduction to Multiple Target Tracking

Background

Tracking is essential for the guidance, navigation, and control of autonomous systems. A tracking system estimates targets (number of targets and their states) and evaluates the situational environment in an area of interest by taking detections (kinematic parameters and attributes) and tracking these targets with time. The simplest tracking system is a single target tracking (STT) system in a clutterless environment, which assumes one target only in an area of interest. An STT does not require data assignment or association, because the detection of the standalone target can be directly fed to an estimator or filter used to estimate the state of the target.

Modern tracking systems usually involve multiple target tracking (MTT) systems, in which one or more sensors generate multiple detections from multiple targets, and one or more tracks are used to estimate the states of these targets. An MTT must assign detections to tracks before the detections can be used to update the tracks. The MTT assignment problem is challenging because of several factors:

- Target or detection distribution — If targets are sparsely distributed, then associating a target to its corresponding detection is relatively easy. However, if targets or detections are densely distributed, the assignment becomes ambiguous because assigning a target to a detection or a nearby detection rarely makes any differences on the cost.

- Probability of detection (P_d) of the sensor — P_d describes the probability that a target is detected by the sensor if the target is within the field of view of the sensor. If the P_d of a sensor is small, then the true target might not generate any detection during a sensor scan. As a result, the track represented by the true target may steal detections from other tracks.

- Sensor resolution — Sensor resolution determines the sensor's ability to distinguish between detections from two targets. If the sensor resolution is low, then two targets in proximity might only give rise to one detection. This violates the common assumption that each detection can only be assigned to one track and results in unresolvable assignment conflicts between tracks.

- Clutter or false alarm rate of the sensor — False alarms introduce additional possible assignments and therefore increase the complexity of data assignment.

- The number of targets and detections — The number of possible assignments increases exponentially as the number of targets and detections increases. Therefore, obtaining an optimal assignment requires more computations.

Elements of an MTT System
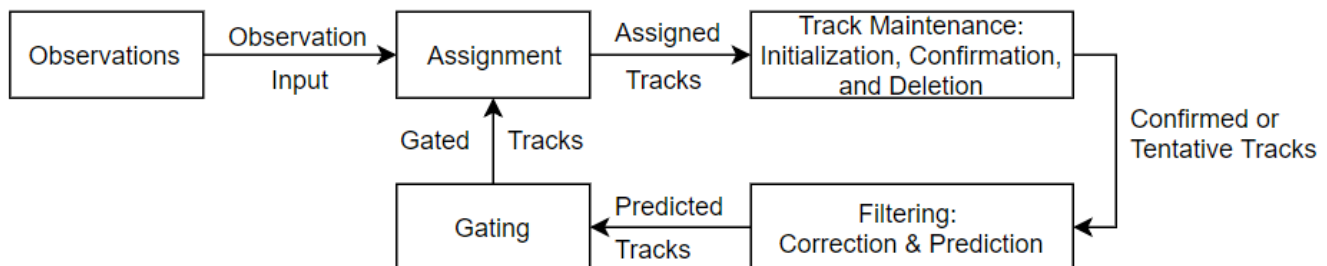
The figure gives a structural representation of the functional elements of a simple recursive MTT system [1]. In real world applications, the functions of these elements can overlap considerably. However, this representation provides a convenient partitioning to introduce the typical functions in an MTT system.

To interpret this diagram, assume a tracker has maintained confirmed or tentative tracks from the previous scan. Now, the system considers whether to update tracks based on any new detections received from sensors. To assign detections to the corresponding tracks:

1 The internal filter (such as a Kalman filter) predicts the confirmed or tentative tracks from the previous step to the current step.

2 The tracker uses the predicted estimate and covariance to form a validation gate around the predicted track.

3 The detections falling within the gate of a track are considered as candidates for assignment to the track.

4 An assignment algorithm (based on the specific tracker, such as GNN or TOMHT) determines the track-to-detection association.

5 Based on the assignment, the tracker executes track maintenance, including initialization, confirmation, and deletion:

- Unassigned observations can initiate new tentative tracks.

- A tentative track becomes confirmed if the quality of the track satisfies the confirmation criteria.

- Low-quality tracks are deleted based on the deletion criteria.

6 The new track set (tentative and confirmed) is predicted to the next scan step to form validation gates.

Detections

Detection is a collective term used to refer to all the observed or measured quantities included in a report output (see `objectDetection` for example) from a sensor. In general, an observation may contain measured kinematic quantities (such as range, line of sight, and range-rate) and measured attributes (such as target type, identification number, and shape). A detection should also contain the time at which measurements are obtained.

For point target tracking, detections received from a single sensor scan can contain at most one observation from each target. This assumption greatly simplifies the assignment problem. One sensor can generate zero detections for a target within its field of view, because the probability of detection, P_{d}, of each sensor is usually less than 1. Also, each sensor can generate false alarm detections that do not correspond to true targets.

High-resolution sensors may generate multiple detections per target, which requires partitioning the detections into one representative detection before feeding to assignment-based trackers (such as `trackerGNN`, `trackerJPDA`, and `trackerTOMHT`). See "Extended Object Tracking of Highway Vehicles with Radar and Camera" on page 5-147 for more details.

Gating and Assignment

For details about gating and assignment, see "Introduction to Assignment Methods in Tracking Systems" on page 4-33, which provides a comprehensive introduction of assignment methods. This section only covers the basics of gating and assignment used in the three assignment-based trackers, `trackerGNN`, `trackerJPDA`, and `trackerTOMHT`.

Gating is a screening mechanism used to determine which detections are valid candidates to update existing tracks. The purpose of gating is to reduce unnecessary computation in track-to-detection assignment. A validation gate of a predicted track is formed using the predicted state and its associated covariance, such that the detections with high probability of association fall within the validation gate of a track. Only the detections within the gate of a track are considered for assignment with the track.

After gating, the assignment function determines which track-to-detection assignments to make. Three methods of assignment are used with three trackers in the toolbox:

- `trackerGNN` — Global nearest data association. Based on likelihood theory, the goal of the GNN method is to minimize an overall distance function that considers all track-to-detection assignments.

- `trackerJPDA` — Joint probability data association. The JPDA method applies a soft assignment, such that detections within the validation gate of a track can all make weighted contributions to the track based on their probability of association.

- `trackerTOMHT` — Track-oriented multiple hypothesis tracking. Unlike GNN and JPDA, MHT is a deferred decision approach, which allows difficult data association situations to be postponed until more information is received.

The decision of which tracker to use depends on the type of targets and computational resources available:

- The GNN algorithm is the simplest to employ. It has low computational cost and can result in adequate performance for tracking sparsely distributed targets.

- The JPDA algorithm, which requires more computational cost, is also applicable for widely spaced targets. It usually performs better in a clutter environment than GNN.

- The TOMHT tracker, which requires heavily on computational resources, normally results in the best performance among all the three trackers, especially for densely distributed targets.

For more details, see the "Tracking Closely Spaced Targets Under Ambiguity" on page 5-167 example for a comparison of these three trackers.

Track Maintenance

Track maintenance refers to the function of track initiation, confirmation, and deletion.

Track Initiation. When a detection is not assigned to an existing track, a new track might need to be created:

- The GNN approach starts new tentative tracks on observations that are not assigned to existing tracks.

- The JPDA approach starts new tentative tracks on observations with probability of assignment lower than a specified threshold.

- The MHT approach starts new tentative tracks on observations whose distances to existing tracks are larger than a specified threshold. The tracker uses subsequent data to determine which of these newly initiated tracks are valid.

Track Confirmation. Once a tentative track is formed, a confirmation logic identifies the status of the track. Three track confirmation logics are used in the toolbox:

- History Logic: A track is confirmed if the track has been assigned to a detection for at least *M* updates during the last *N* updates. You can set the specific values for *M* and *N*. `trackerGNN` and `trackerJPDA` use this logic.

- Track Score Logic: A track is confirmed if its score is higher than a specified threshold. A higher track score means that the track is more likely to be valid. The score is the ratio of the probability that the track is from a real target to the probability that the track is false. `trackerGNN` and `trackerTOMHT` use this logic.

- Integrated Logic: A track is confirmed if its integrated probability of existence is higher than a threshold. `trackerJPDA` uses this logic.

Track Deletion. A track is deleted if it is not updated within some reasonable time. The track deletion criteria are similar to the track confirmation criteria:

- History Logic: A track is deleted if the track has not been assigned to a detection at least *P* times during last *R* updates.

- Track Score Logic: A track is deleted if its score decreases from the maximum score by a specified threshold.

- Integrated Logic: A track is deleted if its integrated probability of existence is lower than a specified threshold.

For more details, see the "Introduction to Track Logic" on page 5-76 example.

Filtering

The main functions of a tracking filter are:

1 Predict tracks to the current time.
2 Calculate distances from the predicted tracks to detections and the associated likelihoods for gating and assignment.
3 Correct the predicted tracks using assigned detections.

Sensor Fusion and Tracking Toolbox offers multiple tracking filters that can be used with the three assignment-based trackers (`trackerGNN`, `trackerJPDA`, and `trackerTOMHT`). For a comprehensive introduction of these filters, see "Introduction to Estimation Filters" on page 4-9.

Tracking Metrics

Sensor Fusion and Tracking Toolbox provides tools to analyze the tracking performance if the truths are known:

- You can use `trackAssignmentMetrics` to evaluate the performance of track assignment and maintenance. `trackAssignmentMetrics` provides indexes such as number of the track swaps, number of divergence steps, and number of redundant assignments.

- You can use `trackErrorMetrics` to evaluate the accuracy of tracking. `trackErrorMetrics` provides multiple root mean square (RMS) error values, which numerically illustrate the accuracy performance of the tracker.

- You can use `trackOSPAMetric` to compute the optimal subpattern assignment metric. `trackErrorMetrics` provides three scalar error components — localization error, labelling error, and cardinality error to evaluate tracking performance.

Non-Assignment-Based Trackers

`trackerGNN`, `trackerJPDA`, and `trackerTOMHT` are assignment-based trackers, meaning that the track-to-detection assignment is required. The toolbox also offers a random finite set (RFS) based tracker, `trackerPHD`. You can use its supporting features `ggiwphd` to track extended objects and `gmphd` to track both extended objects and point targets.

See Also

`ggiwphd` | `gmphd` | `objectDetection` | `trackerGNN` | `trackerJPDA` | `trackerPHD` | `trackerTOMHT`

References

[1] Blackman, S., and R. Popoli. *Design and Analysis of Modern Tracking Systems.* Artech House Radar Library, Boston, 1999.

[2] Musicki, D., and R. Evans. "Joint Integrated Probabilistic Data Association: JIPDA." *IEEE transactions on Aerospace and Electronic Systems .* Vol. 40, Number 3, 2004, pp. 1093 –1099.

[3] Werthmann, J. R.. "Step-by-Step Description of a Computationally Efficient Version of Multiple Hypothesis Tracking." In *International Society for Optics and Photonics*, Vol. 1698, pp. 228 – 301, 1992.

Introduction to Assignment Methods in Tracking Systems

Background

In a multiple target tracking (MTT) system, one or more sensors generate multiple detections from multiple targets in a scan. To track these targets, one essential step is to assign these detections correctly to the targets or tracks maintained in the tracker so that these detections can be used to update these tracks. If the number of targets or detections is large, or there are conflicts between different assignment hypotheses, assigning detections is challenging.

Depending on the dimension of the assignment, assignment problems can be categorized into:

- 2-D assignment problem – assigns n targets to m observations. For example, assign 5 tracks to 6 detections generated from one sensor at one time step.
- S-D assignment problem – assigns n targets to a set (m_1, m_2, m_3, ...) of observations. For example, assign 5 tracks to 6 detections from one sensor, and 4 detections from another sensor at the same time. This example is a typical 3-D assignment problem.

To illustrate the basic idea of an assignment problem, consider a simple 2-D assignment example. One company tries to assign 3 jobs to 3 workers. Because of the different experience levels of the workers, not all workers are able to complete each job with the same effectiveness. The cost (in hours) of each worker to finish each job is given by the cost matrix shown in the table. An assignment rule is that each worker can only take one job, and one job can only be taken by one worker. To guarantee efficiency, the object of this assignment is to minimize the total cost.

Worker	Job		
	1	2	3
1	41	72	**39**
2	22	**29**	49
3	**27**	39	60

Since the numbers of workers and jobs are both small in this example, all the possible assignments can be obtained by enumeration, and the minimal cost solution is highlighted in the table with assignment pairs (1, 3), (2, 2) and (3, 1). In practice, as the size of the assignment becomes larger, the optimal solution is difficult to obtain for 2-D assignment. For an S-D assignment problem, the optimal solution may not be obtainable in practice.

2-D Assignment in Multiple Target Tracking

In the 2-D MTT assignment problem, a tracker tries to assign multiple tracks to multiple detections. Other than the dimensionality challenge mentioned above, a few other factors can significantly change the complexity of the assignment:

- Target or detection distribution — If targets are sparsely distributed, associating a target to its corresponding detection is relatively easy. However, if targets or detections are densely distributed, assignments become ambiguous because assigning a target to a detection or another nearby detection rarely makes any differences on the cost.
- Probability of detection (P_d) of the sensor — P_d describes the probability that a target is detected by the sensor if the target is within the field of view of the sensor. If the P_d of a sensor is small,

then the true target may not give rise to any detection during a sensor scan. As a result, the track represented by the true target may steal detections from other tracks.

- Sensor resolution — Sensor resolution determines the sensor's ability to distinguish the detections from two targets. If the sensor resolution is low, then two targets in proximity may only give rise to one detection. This violates the common assumption that each detection can only be assigned to one track and results in unresolvable assignment conflicts between tracks.

- Clutter or false alarm rate of the sensor — False alarms introduce additional possible assignments and therefore increase the complexity of data assignment.

The complexity of the assignment task can determine which assignment methods to apply. In Sensor Fusion and Tracking Toolbox toolbox, three 2-D assignment approaches are employed corresponding to three different trackers:

- `trackerGNN` — adopts a global nearest data assignment approach
- `trackerJPDA` — adopts a joint probability data association approach
- `trackerTOMHT` — adopts a tracker-oriented multiple hypothesis tracking approach

Note that each tracker processes the data from sensors sequentially, meaning that each tracker only deals with the assignment problem with the detections of one sensor at a time. Even with this treatment, there may still be too many assignment pairs. To reduce the number of track and detection pairs considered for assignment, the gating technique is frequently used.

Gating

Gating is a screening mechanism to determine which observations are valid candidates to update existing tracks and eliminate unlikely detection-to-track pairs using the distribution information of the predicted tracks. To establish the validation gate for a track at the current scan, the estimated track for the current step is predicted from the previous step.

For example, a tracker confirms a track at time t_k and receives several detections at time t_{k+1}. To form a validation gate at time t_{k+1}, the tracker first needs to obtain the predicted measurement as:

$$\widehat{y}_{k+1} = h(\widehat{x}_{k+1|k})$$

where $\hat{x}_{k+1|k}$ is the track estimate predicted from time t_k and $h(\hat{x}_{k+1|k})$ is the measurement model that outputs the expected measurement given the track state. The observation residual vector is

$$\tilde{y} = y_{k+1} - \widehat{y}_{k+1}$$

where y_{k+1} is the actual measurement. To establish the boundary of the gate, the detection residual covariance S is used to form an ellipsoidal validation gate. The ellipsoidal gate that establishes a spatial ellipsoidal region in the measurement space is defined in Mahalanobis distance as:

$$d^2(y_{k+1}) = \tilde{y}^T S^{-1} \tilde{y} \le G$$

where G is the gating threshold which you can specify based on the assignment requirement. Increasing the threshold can incorporate more detections into the gate.

After the assignment gate is established for each track, the gating status of each detection y_i ($i = 1$, ...,n) can be determined by comparing its Mahalanobis distance $d^2 (y_i)$ with the gating threshold G. If $d^2 (y_i) < G$, then detection y_i is inside the gate of the track and will be considered for association. Otherwise, the possibility of the detection associated with the track is removed. In Figure 1, T_1 represents a predicted track estimate, and O_1 – O_6 are six detections. Based on the gating result, O_1, O_2, and O_3 are within the validation gate in the figure.
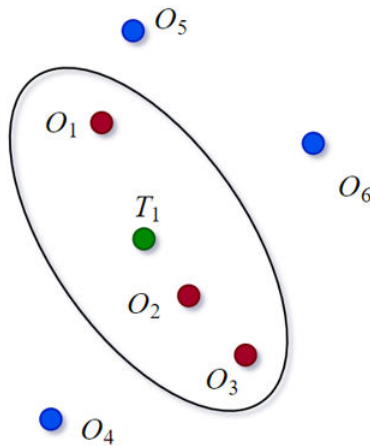
Figure 1. A gating example

Global Nearest Neighbor (GNN) Method

The GNN method is a single hypothesis assignment method. For each new data set, the goal is to assign the global nearest observations to existing tracks and to create new track hypotheses for unassigned detections.

The GNN assignment problem can be easily solved if there are no conflicts of association between tracks. The tracker only needs to assign a track to its nearest neighbor. However, conflict situations (see Figure 2) occur when there is more than one observation within a track's validation gate or an observation is in the gates of more than one track. To resolve these conflicts, the tracker must evaluate a cost matrix.
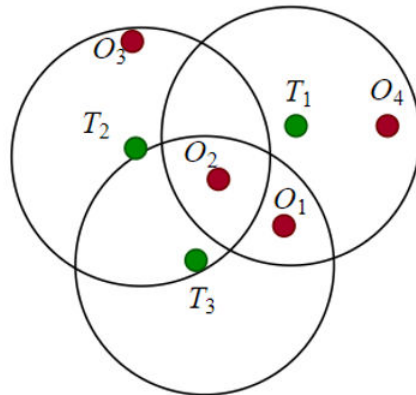
Figure 2. GNN with association conflicts

The elements of a cost matrix for the GNN method includes the distance from tracks to detections and other factors you might want to consider. For example, one approach is to define a generalized statistical distance between observation j to track i as:

$$C_{ij} = d_{ij} + \ln\left(\left|S_{ij}\right|\right)$$

where d_{ij} is the Mahalanobis distance and $\ln(|S_{ij}|)$, the logarithm of the determinant of the residual covariance matrix, is used to penalize tracks with greater prediction uncertainty.

For the assignment problem given in Figure 2, the following table shows a hypothetical cost matrix. The nonallowed assignments, which failed the gating test, are denoted by X. (In practice, the costs of nonallowed assignments can be denoted by large values, such as 1000.)

Tracks	Observations			
	O_1	O_2	O_3	O_4
T_1	9	6	X	**6**
T_2	X	**3**	10	X
T_2	**8**	4	X	X

For this problem, the highlighted optimal solution can be found by enumeration. Detection O_3 is unassigned, so the tracker will use it to create a new tentative track. For more complicated GNN assignment problems, more accurate formulations and more efficient algorithms to obtain the optimal or suboptimal solution are required.

A general 2-D assignment problem can be formed as following. Given the cost matrix element C_{ij}, find an assignment $Z = \{z_{ij}\}$ that minimizes

$$J = \sum_{i=0}^{n} \sum_{j=0}^{m} C_{ij} z_{ij}$$

subject to two constraints:

$$\sum_{i=0}^{m} z_{ij} = 1, \forall j$$

$$\sum_{j=0}^{n} z_{ij} = 1, \forall i$$

If track i is assigned to observation j, then $z_{ij} = 1$. Otherwise, $z_{ij} = 0$. $z_{i0} = 1$ represents the hypothesis that track i is not assigned to any detection. Similarly, $z_{0j} = 1$ represents the hypothesis that observation j is not assigned to any track. The first constraint means each detection can be assigned to no more than one track. The second constraint means each track can be assigned to no more than one detection.

Sensor Fusion and Tracking Toolbox provides multiple functions to solve 2-D GNN assignment problems:

- `assignmunkres` – Uses the Munkres algorithm, which guarantees an optimal solution but may require more calculation operations.
- `assignauction` – Uses the auction algorithm, which requires fewer operations but can possibly converge on an optimal or suboptimal solution.
- `assignjv` – Uses the Joker-Volgenant algorithm, which also converges on an optimal or suboptimal solution but usually with a faster converging speed.

In `trackerGNN`, you can select the assignment algorithm by specifying the `Assignment` property.

K Best Solutions to the 2-D Assignment Problem

Because of the uncertainty nature of assignment problems, only obtaining a solution (optimal or suboptimal) may not be sufficient. To account for multiple hypotheses about the assignment between tracks and detections, multiple suboptimal solutions are required. These suboptimal solutions are called K best solutions to the assignment problem.

The K best solutions are usually obtained by varying the solution obtained by any of the previously mentioned assignment functions. Then, at the next step, the K best solution algorithm removes one track-to-detection pair in the original solution and finds the next best solution. For example, for this cost matrix:

$$\begin{bmatrix} 10 & 5 & 8 & 9 \\ 7 & \times & 20 & \times \\ \times & 21 & \times & \times \\ \times & 15 & 17 & \times \\ \times & \times & 16 & 22 \end{bmatrix}$$

each row represents the cost associated with a track, and each column represents the cost associated with a detection. As highlighted, the optimal solution is (7,15,16, 9) with a cost of 47. In the next step, remove the first pair (corresponding to 7), and the next best solution is (10,15, 20, 22) with a cost of 67. After that, remove the second pair (corresponding to 15), and the next best solution is (7, 5,16, 9) with a cost of 51. After a few steps, the five best solutions are:

Solution	Cost
(7,15,16, 9)	47
(7,5,17, 22)	51
(7,15, 8, 22)	52
(7, 21,16, 9)	53
(7, 21,17, 9)	53

See the "Find Five Best Solutions Using Assignkbest" example, which uses the `assignkbest` function to find the K best solutions.

Joint Probability Data Association (JPDA) Method

While the GNN method makes a rigid assignment of a detection to a track, the JPDA method applies a soft assignment so that detections within the validation gate of a track can all make weighted contributions to the track based on their probability of association.

For example, for the gating results shown in Figure 1, a JPDA tracker calculates the possibility of association between track T_1 and observations O_1, O_2, and O_3. Assume the association probability of these three observations are p_{11}, p_{12}, and p_{13}, and their residuals relative to track T_1 are $\tilde{y}_{11}$, $\tilde{y}_{12}$, and $\tilde{y}_{13}$, respectively. Then the weighted sum of the residuals associated with track T_1 is:

$$\tilde{y}_1 = \sum_{j=1}^{3} p_{1j}\tilde{y}_{1j}$$

In the tracker, the weighted residual is used to update track T_1 in the correction step of the tracking filter. In the filter, the probability of unassignment, p_{10}, is also required to update track T_1. For more details, see "JPDA Correction Algorithm for Discrete Extended Kalman Filter".

The JPDA method requires one more step when there are conflicts between assignments in different tracks. For example, in the following figure, track T_2 conflicts with T_1 on the assignment of observation O_3. Therefore, to calculate the association probability p_{13}, the joint probability that T_2 is not assigned to O_3 (that is T_2 is assigned to O_6 or unassigned) must be accounted for.
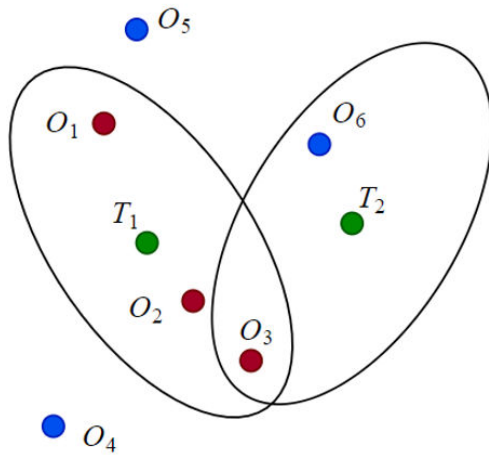
Track-Oriented Multiple Hypothesis Tracking (TOMHT) Method

Unlike the JPDA method, which combines all detections within the validation gate using a weighted sum, the TOMHT method generates multiple hypotheses or branches of the tracks based on the detections within the gate and propagates high-likelihood branches between scan steps. After propagation, these hypotheses can be tested and pruned based on the new set of detections.

For example, for the gating scenario shown in Figure 1, a TOMHT tracker considers the following four hypotheses:

- Assign no detection to T_1 resulting in hypothesis T_{10}
- Assign O_1 to T_1 resulting in hypothesis T_{11}
- Assign O_2 to T_1 resulting in hypothesis T_{12}
- Assign O_3 to T_1 resulting in hypothesis T_{13}

Given the assignment threshold, the tracker will calculate the possibility of each hypothesis and discard hypotheses with probability lower than the threshold. Hypothetically, if only p_{10} and p_{11} are larger than the threshold, then only T_{10} and T_{11} are propagated to the next step for detection update.

S-D Assignment in Multiple Target Tracking

In an S-D assignment problem, the dimension of assignment S is larger than 2. Note that all three trackers (`trackerGNN`, `trackerJPDA`, and `trackerTOMHT`) process detections from each sensor sequentially, which results in a 2-D assignment problem. However, some applications require a tracker that processes simultaneous observations from multiple sensor scans all at once, which requires solving an S-D assignment problem. Meanwhile, the S-D assignment is widely used in tracking applications such as static data fusion, which preprocesses the detection data before fed to a tracker.

An S-D assignment problem for static data fusion has S scans of a surveillance region from multiple sensors simultaneously, and each scan consists of multiple detections. The detection sources can be real targets or false alarms. The object is to detect an unknown number of targets and estimate their states. For example, as shown in the Figure 4, three sensor scans produce six detections. The detections in the same color belong to the same scan. Since each scan generates two detections, there are probably two targets in the region of surveillance. To choose between different assignment or association possibilities, evaluate the cost matrix.

Region of Surveillance
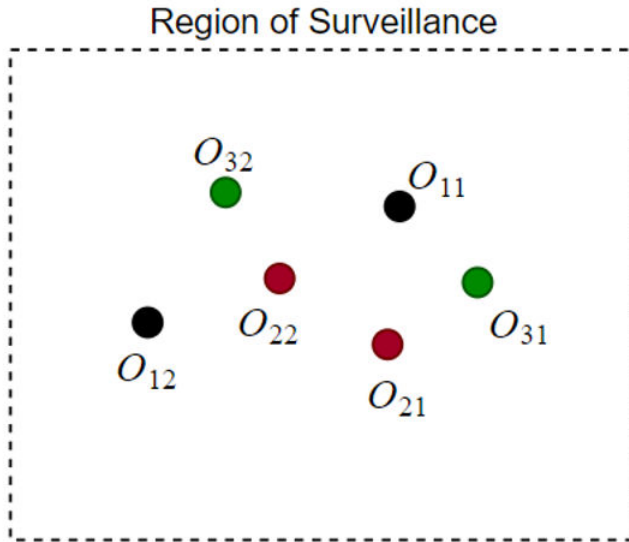
The calculation of the cost can depend on many factors, such as the distance between detections and the covariance distribution of each detection. To illustrate the basic concept, the assignment costs for a few hypotheses are hypothetically given in the table [1].

Assignment Hypotheses	First Scan Observations (O_{1x})	Second Scan Observation (O_{2x})	Third Scan Observation (O_{3x})	Cost
1	0	1	1	−10.2
2	1	2	0	−10.9
3	1	1	1	−18.0
4	1	1	2	−14.8
5	1	2	1	−17.0
6	2	0	1	−13.2
7	2	0	2	−10.6
8	2	2	0	−11.1
9	2	1	2	−14.1
10	2	2	2	−16.7

In the table, 0 denotes a track is associated with no detection in that scan. Assume the hypotheses not shown in the table are truncated by gating or neglected because of high costs. To concisely represent each track, use c_{ijk} to represent the cost for association of observation i in scan 1, j in scan 2, and k in scan 3. For example, for the assignment hypothesis 1, c_{011} = -10.2. Several track hypotheses conflict with other in the table. For instance, the two most likely assignments, c_{111} and c_{121} are incompatible because they share the same observation in scans 1 and 3.

The goal of solving an S-D assignment problem is to find the most likely compatible assignment hypothesis accounting for all the detections. When $S \geq 3$, however, the problem is known to scale with the number of tracks and detections at an exponential rate (NP-hard). The Lagrangian relaxation method is commonly used to obtain the optimal or sub-optimal solution for an S-D assignment problem efficiently.

Brief Introduce to the Lagrangian Relaxation Method for 3-D Assignment

Three scans of data have a number of M_1, M_2, and M_3 observations, respectively. Denote an observation of scan 1, 2, and 3 as i, j, and k, respectively. For example, $i = 1, 2, ..., M_1$. Use z_{ijk} to represent the track formation hypothesis of O_{1i}, O_{2j}, and O_{3k}. If the hypothesis is valid, then $z_{ijk} = 1$; otherwise, $z_{ijk} = 0$. As mentioned, c_{ijk} is used to represent the cost of z_{ijk} association. c_{ijk} is 0 for false alarms and negative for possible associations. The S-D optimization problem can be formulated as:

$$J(z) = \min_{i,j,k} \sum_{i=0}^{M_1} \sum_{j=0}^{M_2} \sum_{k=0}^{M_3} c_{ijk} z_{ijk}$$

subject to three constraints:

$$\sum_{i=0}^{M_1} \sum_{j=0}^{M_2} z_{ijk} = 1, \forall k = 1, 2, ..., M_3$$

$$\sum_{i=0}^{M_1} \sum_{k=0}^{M_3} z_{ijk} = 1, \forall j = 1, 2, ..., M_2$$

$$\sum_{j=0}^{M_2} \sum_{k=0}^{M_3} z_{ijk} = 1, \forall i = 1, 2, ..., M_1$$

The optimization function chooses associations to minimize the total cost. The three constraints ensure that each detection is accounted for (either included in an assignment or treated as false alarm).

The Lagrangian relaxation method approaches this 3-D assignment problem by relaxing the first constraint using Lagrange multipliers. Define a new function $L(\lambda)$:

$$L(\lambda) = \sum_{k=0}^{M_3} \lambda_k \left[\sum_{i=0}^{M_1} \sum_{j=0}^{M_2} z_{ijk} - 1 \right]$$

4-43

where λ_k, $k = 1, 2, ..., M_3$ are Lagrange multipliers. Subtract L from the original object function $J(z)$ to get a new object function, and the first constraint in k is relaxed. Therefore, the 3-D assignment problem reduces to a 2-D assignment problem, which can be solved by any of the 2-D assignment method. For more details, see [1].

The Lagrangian relaxation method allows the first constraint to be mildly violated, and therefore can only guarantee a suboptimal solution. For most applications, however, this is sufficient. To specify the solution accuracy, the method uses the solution gap, which defines the difference between the current solution and the potentially optimistic solution. The gap is nonnegative, and a smaller solution gap corresponds to a solution closer to the optimal solution.

Sensor Fusion and Tracking Toolbox provides `assignsd` to solve for S-D assignment using the Lagrangian relaxation method. Similar to the K best 2-D assignment solver `assignkbest`, the toolbox also provides a K best S-D assignment solver, `assignkbestsd`, which is used to provide multiple suboptimal solutions for an S-D assignment problem.

See "Tracking Using Distributed Synchronous Passive Sensors" on page 5-243 for the application of S-D assignment in static detection fusion.

See Also

`assignTOMHT` | `assignauction` | `assignjv` | `assignkbest` | `assignkbestsd` | `assignmunkres` | `assignsd` | `trackerGNN` | `trackerJPDA` | `trackerTOMHT`

References

[1] Blackman, S., and R. Popoli. *Design and Analysis of Modern Tracking Systems.* Artech House Radar Library, Boston, 1999.

[2] Musicki, D., and R. Evans. "Joint Integrated Probabilistic Data Association: JIPDA." *IEEE Transactions on Aerospace and Electronic Systems .* Vol. 40, Number 3, 2004, pp 1093 –1099.

Introduction to Track-To-Track Fusion

Track-To-Track Fusion Versus Central-Level Tracking

A multiple sensor tracking system can provide better performance than a single sensor system because it can provide broader coverage and better visibility. Moreover, fusing detections from different types of sensors can also improve the quality and accuracy of the target estimates. Two types of architecture are commonly used in a multiple sensor tracking system. In the first type of architecture — central-level tracking — the detections from all the sensors are sent directly to a tracking system that maintains tracks based on all the detections. Theoretically, the central-level tracking architecture can achieve the best performance because it can fully use all the information contained in the detections. However, you can also apply a hierarchical structure with sensor-level tracking combined with track-level fusion for a multiple sensor system. The figure shows a typical central-level tracking system and a typical track-to-track fusion system based on sensor-level tracking and track-level fusion.

Central-Level Tracking

Sensor-Level Tracking and Track-Level Fusion
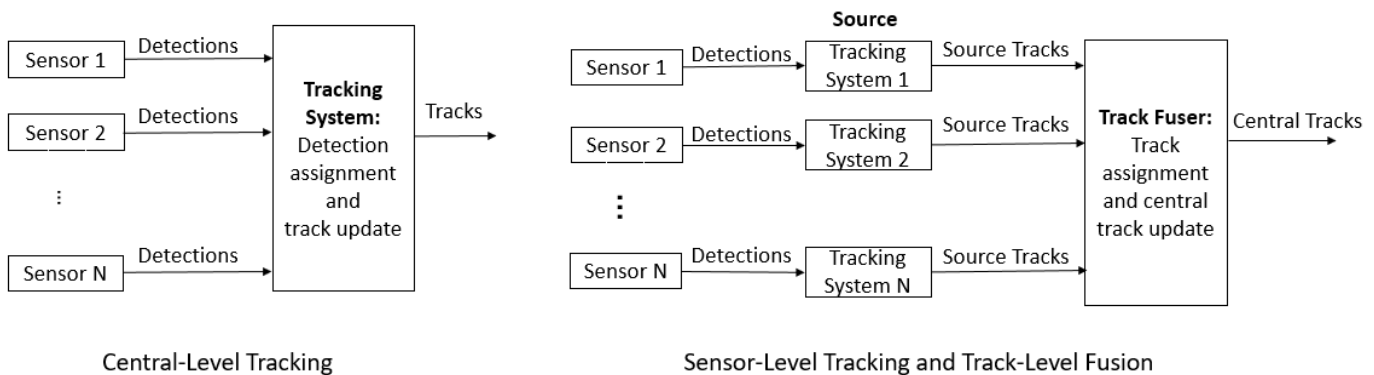
To represent each element in a track-to-track fusion system, call tracking systems that output tracks to a fuser as sources, and call the outputted tracks from sources as source tracks or local tracks. Call the tracks maintained in the fuser as central tracks.

Benefits and Challenges of Track-To-Track Fusion

In some cases, a track-to-track fusion architecture may be preferable to a central-level tracking architecture. These cases include:

- In many applications, a tracking system not only needs to track targets in its environment for self-navigation, but also needs to transfer its maintained tracks to surrounding tracking systems for better overall navigation performance. For example, an autonomous vehicle that tracks its own situational environment can also share the maintained tracks with other vehicles to facilitate their navigation.

- In practice, many sensors directly output tracks instead of detections. Therefore, to combine information from sensors that output tracks, the track-level fusion is required.

- When communication bandwidth is limited, transmitting a track list is often more efficient than transmitting a set of detections. This can be particularly important for cases in which the track list is provided at a reduced rate relative to the scan rate.

- When the number of sensors and detections is large, the computation complexity for the centralized tracking system can be high, especially for detection assignment. The track-to-track

fusion architecture can distribute some assignment and estimation workloads to the sensor-level tracking, which reduces the computation complexity of the fuser.

Despite all the advantages favoring the track-to-track fusion architecture, it also poses additional complexity and challenges to the tracking system. Unlike detections, which can be assumed to be conditionally independent, the track estimates from each source are correlated with each other because they share a common prediction error resulting from a common process model. Therefore, computing a fused track using a standard filtering approach might lead to incorrect results. The following effects must be considered:

- Common process noise — Since the sensors observe and track the same target, they share some common dynamics. As a result, target maneuvering can lead to a mean error that is common to all sensors.
- Time-correlated measurement noise — If the track fusion is repeated over time, the standard Kalman filter assumption that measurements are not correlated over time is violated, because the sensor-level track state estimation errors are correlated over time.

Track Fuser and Tracking Architecture

You can use the `trackFuser` in Sensor Fusion and Tracking Toolbox for the purpose of track-to-track fusion. The `trackFuser` System object™ provides two algorithms to combine source tracks considering the correction effects between different tracks. You can choose the algorithm by specifying the `StateFusion` property of `trackFuser` as:

- `'Cross'` — Uses the cross-covariance fusion algorithm.
- `'Intersection'` — Uses the covariance intersection fusion algorithm.

You can also customize your own fusion algorithm.

Other than the standard track-to-track fusion architecture shown in the preceding figure, you can also use other types of architectures with `trackFuser`. For example, the following figure illustrates a two-vehicle tracking system.

On each vehicle, two sensors track the nearby targets with associated trackers. Each vehicle also has a fuser that fuses source tracks from two trackers. Fuser 6 can transmit its maintained central tracks to Fuser 3. With this architecture, Vehicle 1 can possibly identify targets (Target 2 in the figure) that are not within the field of view of its own sensors.
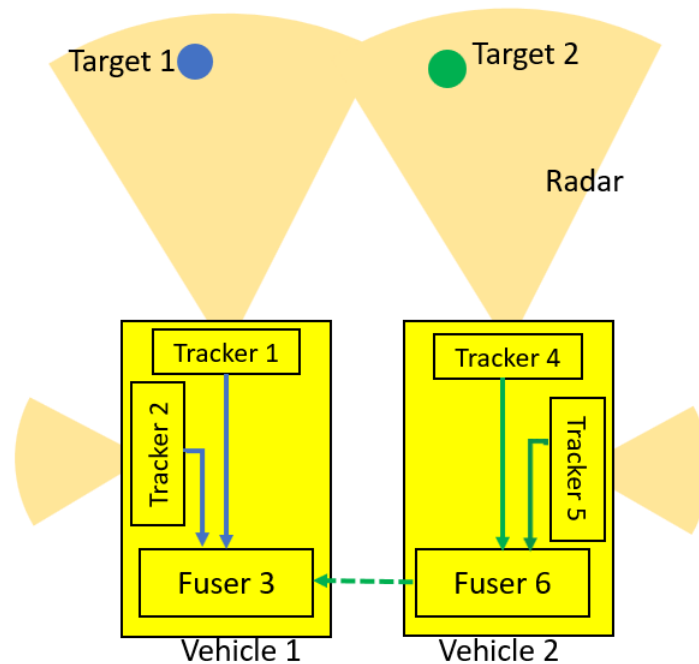
To reduce rumor propagation, you can treat the source tracks from Fuser 6 to Fuser 3 as external by specifying the `IsInternalSource` property of `fuserSourceConfiguration` as `false` when setting up the `SourceConfigurations` property of `TrackFuser`.

Since tracks reported by different trackers can be expressed in different coordinate frames, you need to specify the coordinate transformation between a source and a fuser by specifying the `fuserSourceConfiguration` property.

See Also

`fuserSourceConfiguration` | `trackFuser` | `trackerGNN` | `trackerJPDA` | `trackerPHD` | `trackerTOMHT`

References

[1] Chong, C. Y., S. Mori, W. H. Barker, and K. C. Chang. "Architectures and Algorithms for Track Association and Fusion." *IEEE Aerospace and Electronic Systems Magazine*, Vol. 15, No. 1, 2000, pp. 5 – 13.

Multiple Extended Object Tracking

In traditional tracking systems, the point target model is commonly used. In a point target model:

- Each object is modeled as a point without any spatial extent.
- Each object gives rise to at most one measurement per sensor scan.

Though the point target model simplifies tracking systems, the assumptions above may not be valid when modern tracking systems are considered:

- In modern tracking systems, the dimensions of the extended object play a significant role. For example, in autonomous vehicles, target dimensions must be considered properly to avoid collision with objects around the autonomous system.
- Modern sensors have a high resolution, and an object can occupy more than one resolution cell. As a result, the sensor may report multiple detections for that object. In this case, the point model cannot fully exploit the sensor ability to detect object extent.

In extended object tracking, a sensor can return multiple detections per scan for an extended object. The differences between extended object tracking and point object tracking are more about the sensor properties rather than object properties. For example, if the resolution of a sensor is high enough, even an object with small dimensions can still occupy several resolution cells of the sensor.

Sensor Fusion and Tracking Toolbox offers several methods and examples for multiple extended object tracking. Depending on the assumptions made in the detection and tracker, these methods can be separated into the following categories:

- One detection per object.

  In this category, the conventional trackers (such as `trackerGNN`, `trackerJPDA`, and `trackerTOMHT`) are used, which assume one detection per object. This category can further be divided into two methods:

  - A point detection per object.

    In this method, even though the sensor returns multiple detections per object, these detections are first converted into one representative point detection with certain covariance to account for the distribution of these detections. Then the representative point detection is processed by a conventional tracker, which models the object as a point target and tracks its kinematic state. Even though this method is simple to use, it overlooks the ability of the sensor to detect the object dimension.

    The Point Object Tracker approach shown in the first part of "Extended Object Tracking of Highway Vehicles with Radar and Camera" on page 5-147 example adopts this method.

  - An extended object detection per object.

    In this method, the multiple detections of an extended object are converted into a single parameterized shape detection. The shape detection includes the kinematic states of the object, as well as its extent parameters such as length, width and height. Then the shape detection is processed by a conventional tracker, which models the object as an extended object by tracking both the object kinematic state and its dimensions.

    In the "Track Vehicles Using Lidar: From Point Cloud to Track List" on page 5-369 example, the Lidar detections of each vehicle are converted into a cuboid detection with length, width,

and height. A JPDA tracker is used to track the position, velocity and dimensions for all the vehicles with these cuboid detections.

- Multiple detections per object.

  In this category, extended object trackers (such as `trackerPHD`) are used, which assume multiple detections per object. The detections are fed directly to the tracker, and the tracker models the extended object using certain default geometric shapes with variable sizes.

  In the "Extended Object Tracking of Highway Vehicles with Radar and Camera" on page 5-147 example, the GGIW-PHD Extended Object Tracker approach represents vehicle shapes as ellipses, and the Prototype Extended Object Tracker approach represents vehicle shapes as rectangles.

  In the "Extended Object Tracking With Radar For Marine Surveillance" on page 5-388 example, the GGIW-PHD tracker models the ship shapes as ellipses.

Configure Time Scope MATLAB Object

When you use the `timescope` object in MATLAB®, you can configure many settings and tools from the window. These sections show you how to use the Time Scope interface and the available tools.

Signal Display

This figure highlights the important aspects of the Time Scope window in MATLAB.
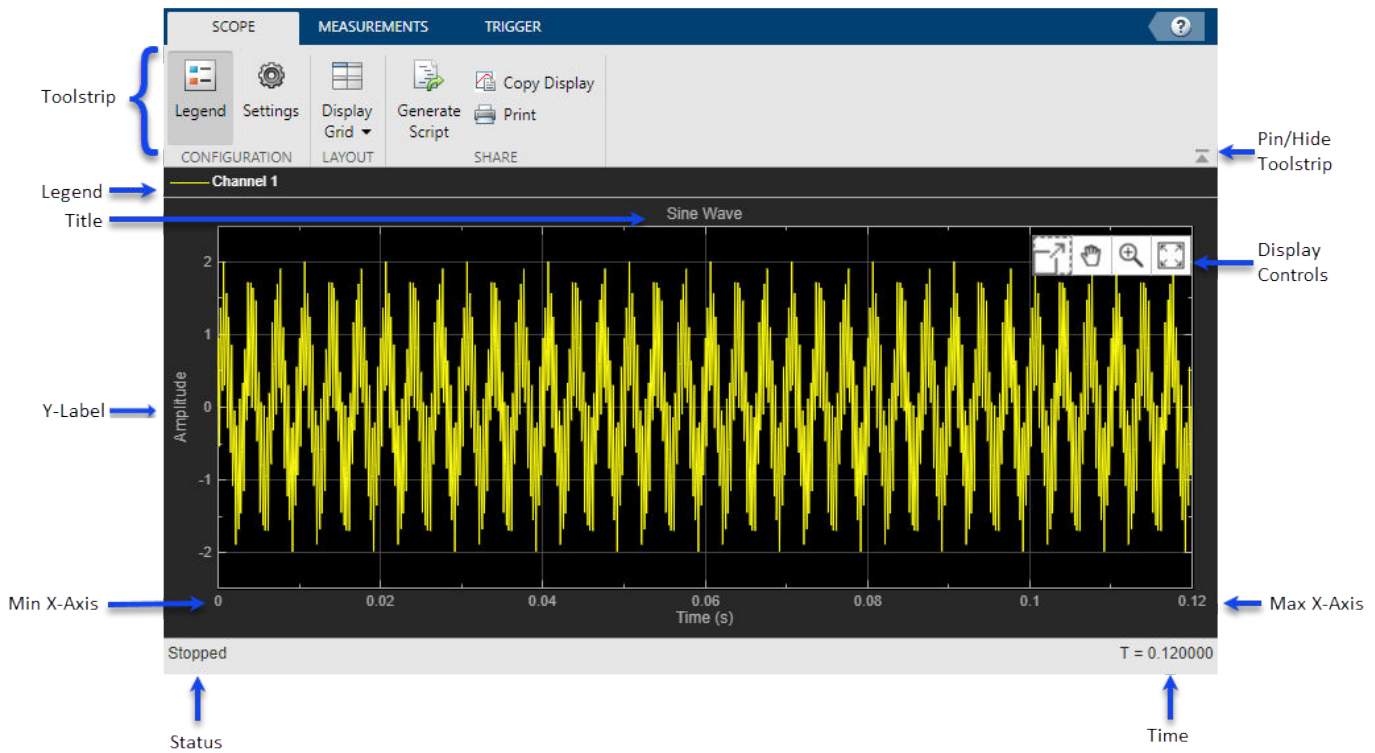
- **Min X-Axis** — Time scope sets the minimum *x*-axis limit using the value of the TimeDisplayOffset property. To change the **Time Offset** from the Time Scope window, click **Settings** (⚙) on the **Scope** tab. Under **Data and Axes**, set the **Time Offset**.

- **Max X-Axis** — Time scope sets the maximum *x*-axis limit by summing the value of the **Time Offset** property with the span of the *x*-axis values. If **Time Span** is set to `Auto`, the span of *x*-axis is `10/SampleRate`.

  The values on the *x*-axis of the scope display remain the same throughout the simulation.

- **Status** — Provides the current status of the plot. The status can be:

  - `Processing` — Occurs after you run the object and before you run the `release` function.
  - `Stopped` — Occurs after you create the scope object and before you first call the object. This status also occurs after you call `release`.

- **Title**, **YLabel** — You can customize the title and the *y*-axis label from **Settings** or by using the `Title` and `YLabel` properties.

- **Toolstrip**

  - **Scope** tab — Customize and share the time scope. For example, showing and hiding the legend
  - **Measurements** tab — Turn on and control different measurement tools.
  - **Trigger** tab — Turn on and modify triggers.

  Use the pin button ⊤ to keep the toolstrip showing or the arrow button ⊼ to hide the toolstrip.

Multiple Signal Names and Colors

By default, if the input signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, the legend for a two-channel signal will display the default names

Channel 1, Channel 2. To show the legend, on the **Scope** tab, click **Settings** (⚙). Under **Display and Labels**, select **Show Legend**. If there are a total of seven input channels, the legend displayed is:

By default, the scope has a black axes background and chooses line colors for each channel in a manner similar to the Simulink® Scope block. When the scope axes background is black, it assigns each channel of each input signal a line color in the order shown in the legend. If there are more than seven channels, then the scope repeats this order to assign line colors to the remaining channels. When the axes background is not black, the signals are colored in this order:

To choose line colors or background colors, on the **Scope** tab click **Settings**. Use the **Axes** color pallet to change the background of the plot. Click **Line** to choose a line to change, and the **Color** drop-down to change the line color of the selected line.

Configure Scope Settings

On the **Scope** tab, the **Configuration** section allows you to modify the scope.

- The **Legend** button turns the legend on or off. When you show the legend, you can control which signals are shown. If you click a signal name in the legend, the signal is hidden from the plot and shown in grey on the legend. To redisplay the signal, click on the signal name again. This button corresponds to the ShowLegend property in the object.

- The **Settings** button opens the settings window which allows you to customize the data, axes, display settings, labels, and color settings.

On the **Scope** tab, the **Layout** section allows you to modify the scope layout dimensions.

The **Display Grid** button enables you to select the display layout of the scope.

Use timescope Measurements and Triggers

All measurements are made for a specified channel. By default, measurements are applied to the first channel. To change which channel is being measured, use the **Select Channel** drop-down on the **Measurements** tab.

Data Cursors

Use the **Data Cursors** button to display screen cursors. Each cursor tracks a vertical line along the signal. The difference between x- and y-values of the signal at the two cursors is displayed in the box between the cursors.

Signal Statistics

Use the **Signal Statistics** button to display various statistics about the selected signal at the bottom of the time scope window. You can hide or show the **Statistics** panel using the arrow button ⌃ in the bottom right of the panel.

- **Max** — Maximum value within the displayed portion of the input signal.
- **Min** — Minimum value within the displayed portion of the input signal.
- **Peak to Peak** — Difference between the maximum and minimum values within the displayed portion of the input signal.
- **Mean** — Average or mean of all the values within the displayed portion of the input signal.
- **Median** — Median value within the displayed portion of the input signal.
- **RMS** — Root mean squared of the input signal.

To customize which statistics are shown and computed, use the **Signal Statistics** drop-down.
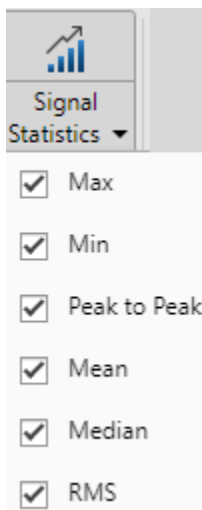
Peak Finder

Use the **Peak Finder** button to display peak values for the selected signal. Peaks are defined as a local maximum where lower values are present on both sides of a peak. End points are not considered peaks. For more information on the algorithms used, see the `findpeaks` function.

When you turn on the peak finder measurements, an arrow appears on the plot at each maxima and a Peaks panel appears at the bottom of the `timescope` window showing the *x* and *y* values at each peak.

You can customize several peak finder settings:

- **Num Peaks** — The number of peaks to show. Must be a scalar integer from 1 through 99.
- **Min Height** — The minimum height difference between a peak and its neighboring samples.
- **Min Distance** — The minimum number of samples between adjacent peaks.
- **Threshold** — The level above which peaks are detected.
- **Label Peaks** — Show labels (**P1**, **P2**, ...) above the arrows on the plot.

Bilevel Measurements

With bilevel measurements, you can measure transitions, aberrations, and cycles.

Bilevel Settings

When using bilevel measurements, you can set these properties:

- **Auto State Level** — When this check box is selected, the Bilevel measurements panel detects the high- and low-state levels of a bilevel waveform. When this check box is cleared, you can enter in values for the high- and low-state levels manually.

  - **High** — Manually specify the value that denotes a positive polarity or high-state level.
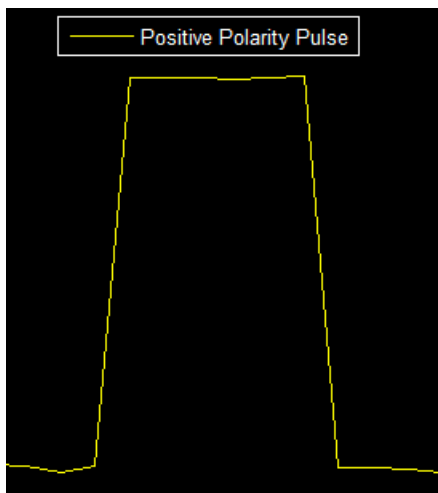
  - **Low** — Manually specify the value that denotes a negative polarity or low-state level.
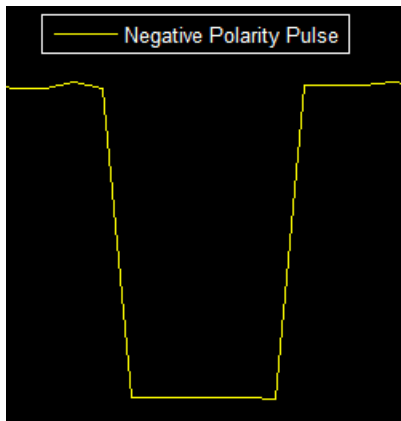
- **State Level Tol. %** — Tolerance levels within which the initial and final levels of each transition must lie to be within their respective state levels. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Upper Ref Level** — Used to compute the end of the rise-time measurement or the start of the fall-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Mid Ref Level** — Used to determine when a transition occurs. This value is expressed as a percentage of the difference between the high- and low-state levels. In the following figure, the mid-reference level is shown as a horizontal line, and its corresponding mid-reference level instant is shown as a vertical line.
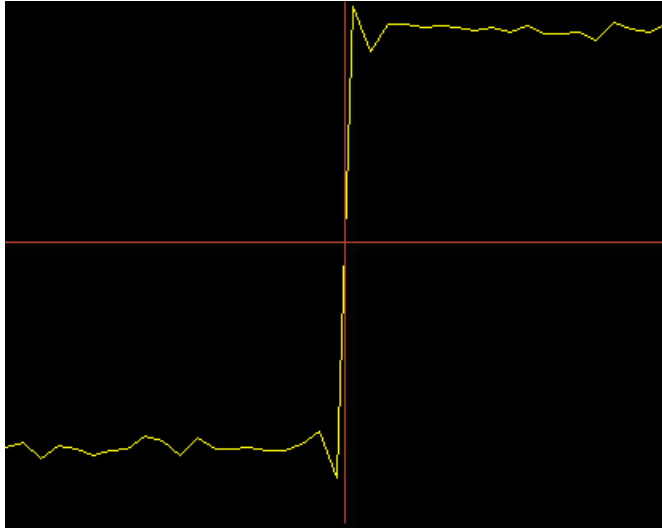
- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.
- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs is used for computing a valid settling time. Settling time is displayed in the **Aberrations** pane.

Transitions

Select **Transitions** to display calculated measurements associated with the input signal changing between its two possible state level values, high and low. The measurements are displayed in the **Transitions** pane at the bottom of the scope window.

The **+ Edges** row measures rising edges or a positive-going transition. A rising edge in a bilevel waveform is a transition from the low-state level to the high-state level with a slope value greater than zero.
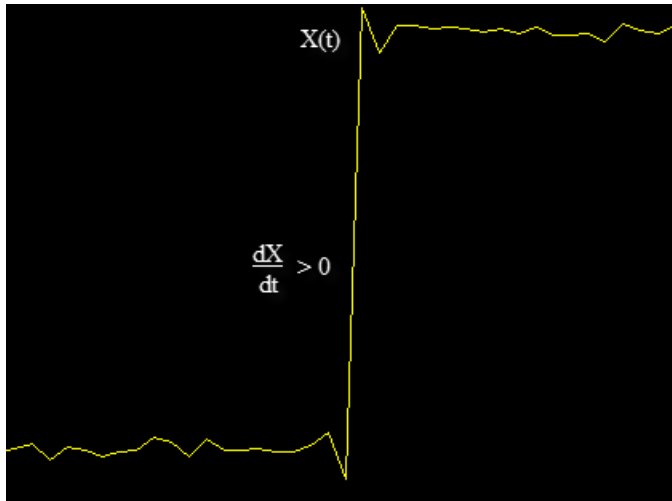
The **- Edges** row measures falling edges or a negative-going transition. A falling edge in a bilevel waveform is a transition from the high-state level to the low-state level with a slope value less than zero.
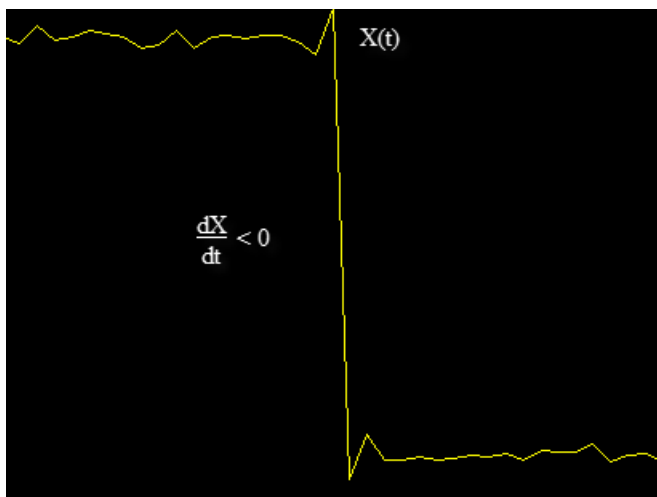
The Transition measurements assume that the amplitude of the input signal is in units of volts. For the transition measurements to be valid, you must convert all input signals to volts.

Aberrations

Select **Aberrations** to display calculated measurements involving the distortion and damping of the input signal such as preshoot, overshoot, and undershoot. Overshoot and undershoot, respectively,

refer to the amount that a signal exceeds and falls below its final steady-state value. Preshoot refers to the amount before a transition that a signal varies from its initial steady-state value. The measurements are displayed in the **Transitions** pane at the bottom of the scope window.

This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.
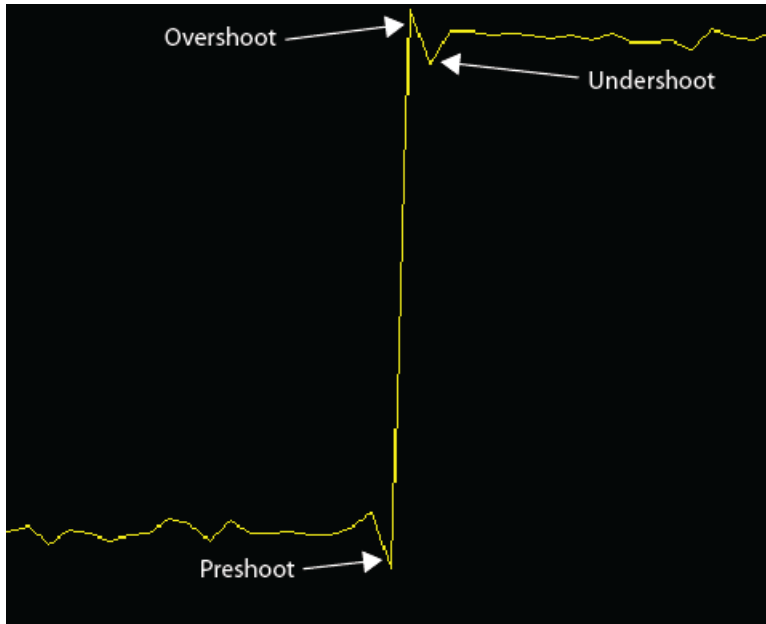
The next figure shows preshoot, overshoot, and undershoot for a falling-edge transition.
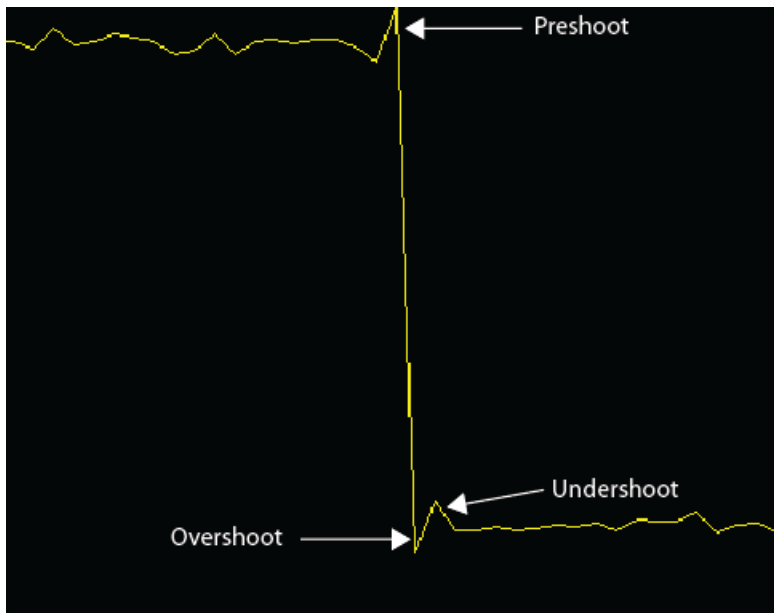
Cycles

Select **Cycles** calculates repetitions or trends in the displayed portion of the input signal. The measurements are displayed in the **Cycles** pane at the bottom of the scope window in two rows: **+ Pulses** for the positive-polarity pulses and **- Pulses** for the negative-polarity pulses.

- **Period** — Average duration between adjacent edges of identical polarity within the displayed portion of the input signal. To calculate period, the `timescope` takes the difference between the mid-reference level instants of the initial transition of each pulse and the next identical-polarity transition. These mid-reference level instants for a positive-polarity pulse appear as red dots in the following figure.
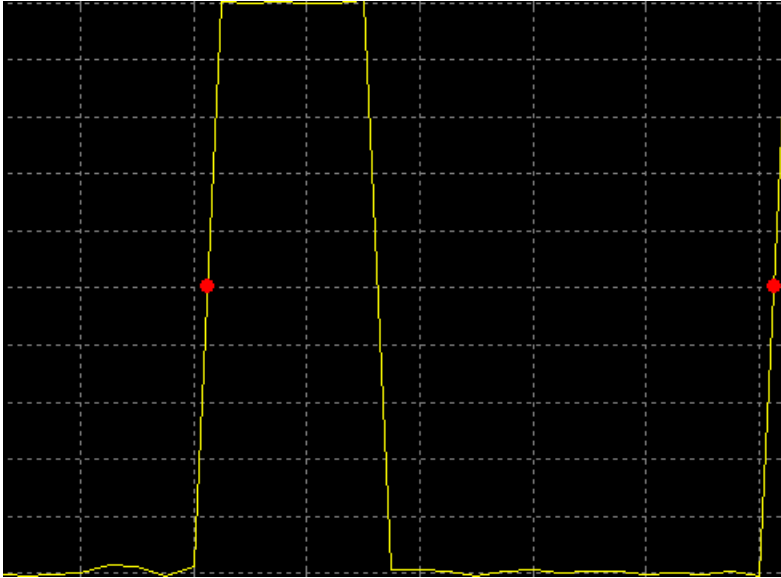
- **Frequency** — Reciprocal of the average period, measured in hertz.

- **Count** — Number of positive- or negative-polarity pulses counted.

- **Width** — Average duration between rising and falling edges of each pulse within the displayed portion of the input signal.

- **Duty Cycle** — Average ratio of pulse width to pulse period for each pulse within the displayed portion of the input signal.

Triggers

Define a trigger event to identify the simulation time of specified input signal characteristics. You can use trigger events to stabilize periodic signals such as a sine wave or capture non-periodic signals such as a pulse that occurs intermittently.
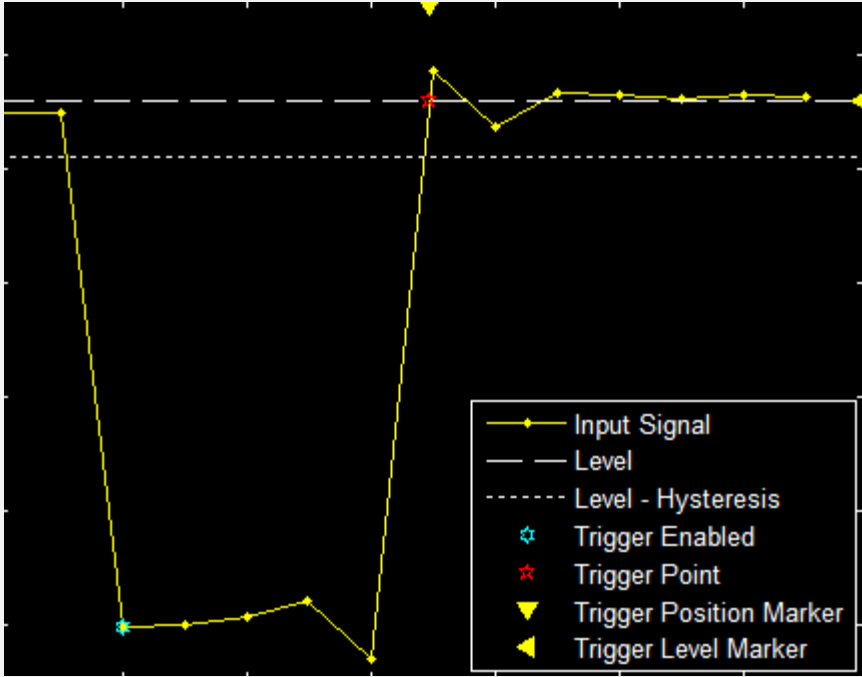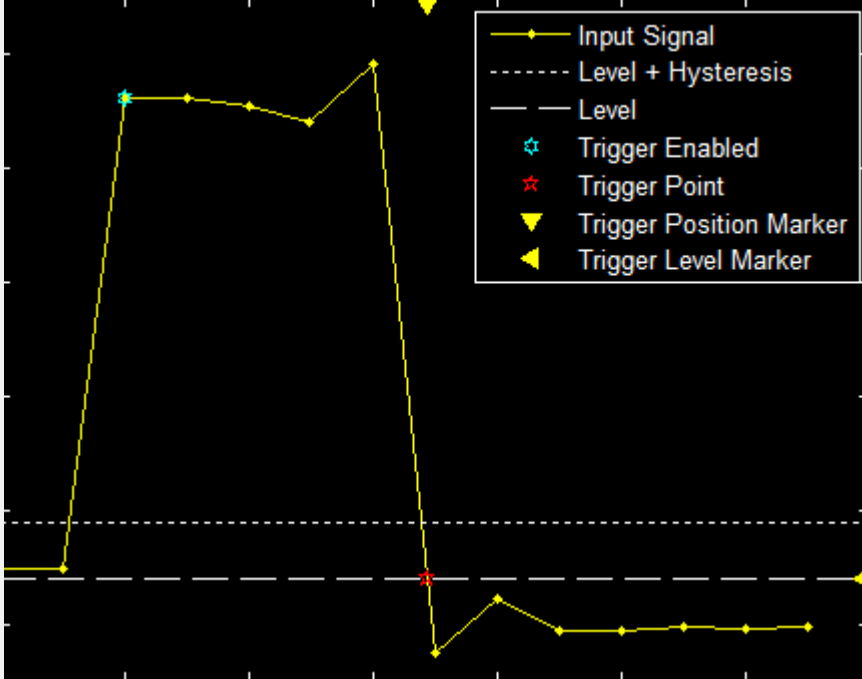
To define a trigger:

1 On the Trigger tab of the scope window, select the channel you want to trigger.

2 Specify when the display updates by selecting a triggering **Mode**.

- **Auto** — Display data from the last trigger event. If no event occurs after one time span, display the last available data.

    Normal — Display data from the last trigger event. If no event occurs, the display remains blank.

- **Once** — Display data from the last trigger event and freeze the display. If no event occurs, the display remains blank. Click the **Rearm** button to look for the next trigger event.

3 Select a triggering type, polarity, and any other properties. See the Trigger Properties table.
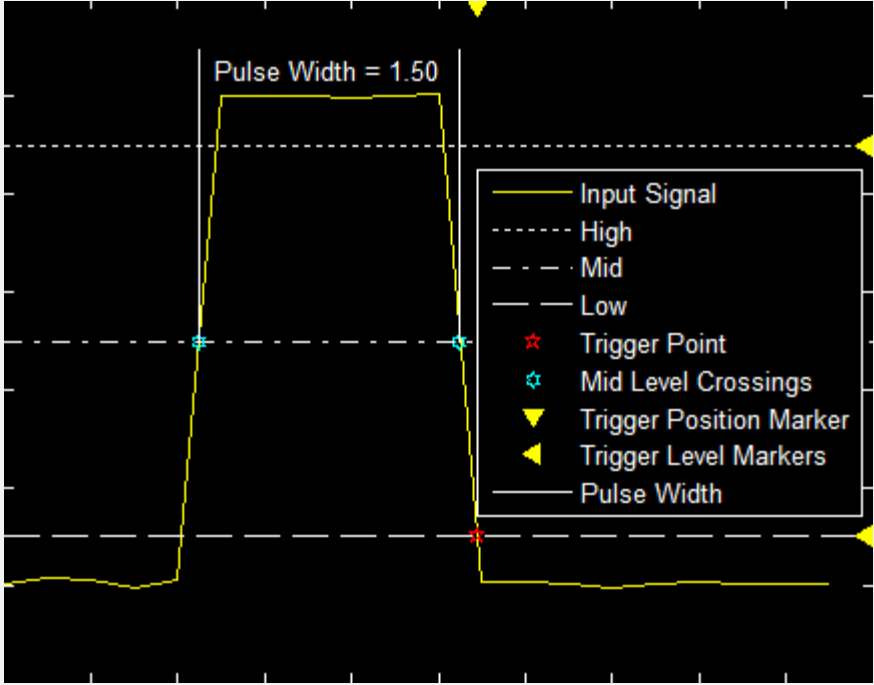
4   Click **Enable Trigger**.
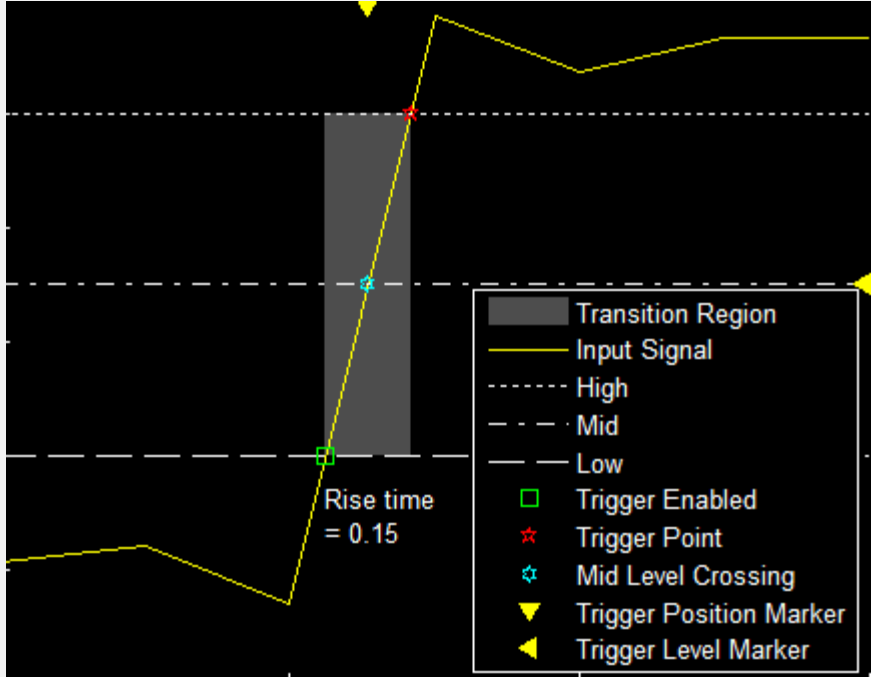
You can set the trigger position to specify the position of the time pointer along the y-axis. You can also drag the time pointer to the left or right to adjust its position.
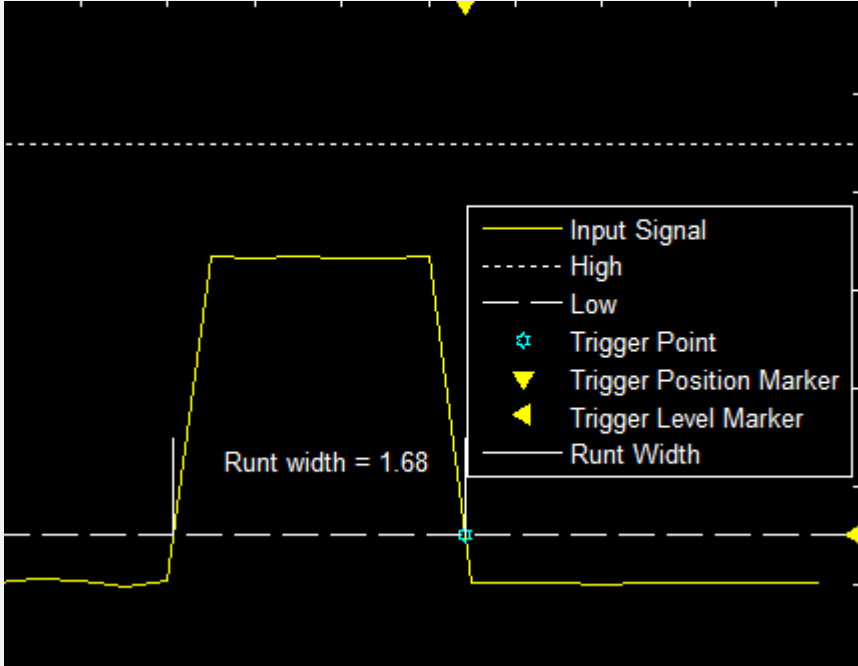
Trigger Properties
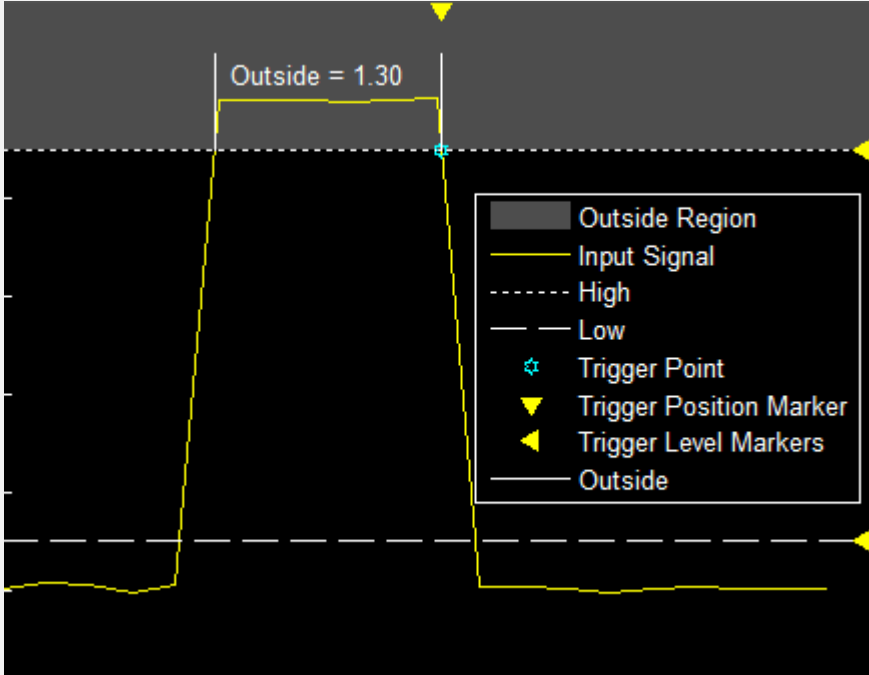
Trigger Type	Trigger Parameters
Edge — Trigger when the signal crosses a threshold.	**Polarity** — Select the polarity for an edge-triggered signal. • Rising — Trigger when the signal is increasing. • Falling — Trigger when the signal value is decreasing.

Trigger Type	Trigger Parameters
	• Either — Trigger when the signal is increasing or decreasing. **Level** — Enter a threshold value for an edge-triggered signal. **Auto level** is 50% **Hysteresis** — Enter a value for an edge-triggered signal. See "Hysteresis of Trigger Signals" on page 4-67 **Delay** — Offset the trigger by a fixed delay in seconds. **Holdoff** — Set the minimum possible time between triggers. **Position** — Set horizontal position of the trigger on the screen.

Trigger Type	Trigger Parameters
`Pulse Width` — Trigger when the signal crosses a low threshold and a high threshold twice within a specified time.	**Polarity** — Select the polarity for a pulse width-triggered signal. • `Positive` — Trigger on a positive-polarity pulse when the pulse crosses the low threshold for a second time. • `Negative` — Trigger on a negative-polarity pulse when the pulse crosses the high threshold for a second time. • `Either` — Trigger on both positive-polarity and negative-polarity pulses. **Note** A glitch-trigger is a special type of a pulse width-trigger. A glitch-trigger occurs for a pulse or spike whose duration is less than a specified amount. You can implement a glitch-trigger by using a pulse-width-trigger and setting the **Max Width** parameter to a small value. **High** — Enter a high value for a pulse-width-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a pulse-width-triggered signal. **Auto level** is 10%. **Min Width** — Enter the minimum pulse width for a pulse-width-triggered signal. Pulse width is measured between the first and second crossings of the middle threshold. **Max Width** — Enter the maximum pulse width for a pulse-width-triggered signal. **Delay** — Offset the trigger by a fixed delay in seconds.
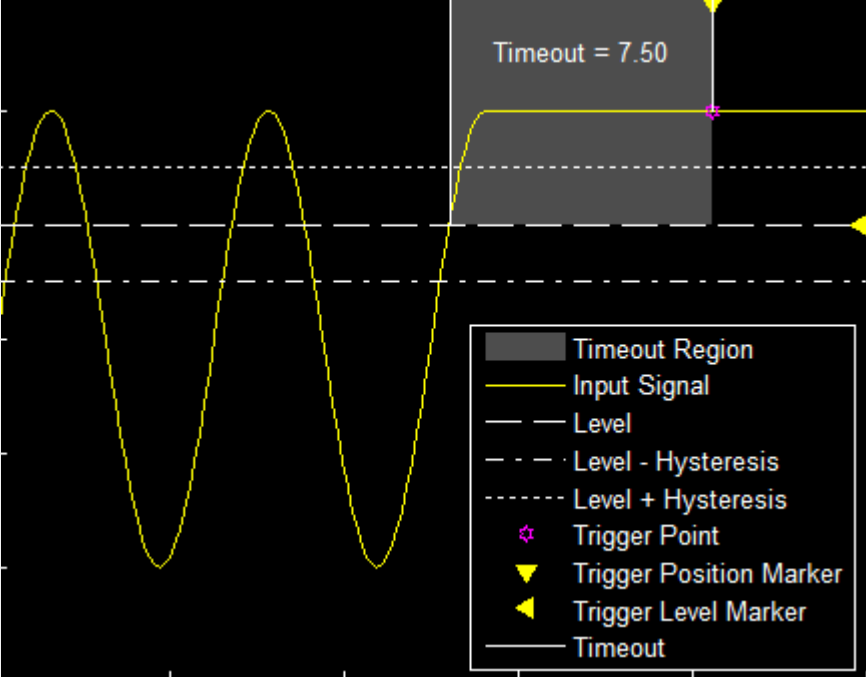
Trigger Type	Trigger Parameters
	Holdoff — Set the minimum possible time between triggers. **Position** — Set horizontal position of the trigger on the screen.
`Transition` — Trigger on the rising or falling edge of a signal that crosses the high and low levels within a specified time range.	**Polarity** — Select the polarity for a transition-triggered signal. • `Rise Time` — Trigger on an increasing signal when the signal crosses the high threshold. • `Fall Time` — Trigger on a decreasing signal when the signal crosses the low threshold. • `Either` — Trigger on an increasing or decreasing signal. **High** — Enter a high value for a transition-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a transition-triggered signal. **Auto level** is 10%. **Min Time** — Enter a minimum time duration for a transition-triggered signal. **Max Time** — Enter a maximum time duration for a transition-triggered signal. **Delay** — Offset the trigger by a fixed delay in seconds. **Holdoff** — Set the minimum possible time between triggers. **Position** — Set horizontal position of the trigger on the screen.

Trigger Type	Trigger Parameters
Runt— Trigger when a signal crosses a low threshold or a high threshold twice within a specified time.	**Polarity** — Select the polarity for a runt-triggered signal. • Positive — Trigger on a positive-polarity pulse when the signal crosses the low threshold a second time without crossing the high threshold. • Negative — Trigger on a negative-polarity pulse. • Either — Trigger on both positive-polarity and negative-polarity pulses. **High** — Enter a high value for a runt-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a runt-triggered signal. **Auto level** is 10%. **Min Width** — Enter a minimum width for a runt-triggered signal. Pulse width is measured between the first and second crossing of a threshold. **Max Width** — Enter a maximum pulse width for a runt-triggered signal. **Delay** — Offset the trigger by a fixed delay in seconds. **Holdoff** — Set the minimum possible time between triggers. **Position** — Set horizontal position of the trigger on the screen.

Trigger Type	Trigger Parameters
`Window` — Trigger when a signal stays within or outside a region defined by the high and low thresholds for a specified time.	**Polarity** — Select the region for a window-triggered signal. • `Inside` — Trigger when a signal leaves a region between the low and high levels. • `Outside` — Trigger when a signal enters a region between the low and high levels.

Trigger Type	Trigger Parameters
	• Either — Trigger when a signal leaves or enters a region between the low and high levels. **High** — Enter a high value for a window-triggered signal. **Auto level** is 90%. **Low** — Enter a low value for a window-trigger signal. **Auto level** is 10%. **Min Time** — Enter the minimum time duration for a window-triggered signal. **Max Time** — Enter the maximum time duration for a window-triggered signal. **Delay** — Offset the trigger by a fixed delay in seconds. **Holdoff** — Set the minimum possible time between triggers. **Position** — Set horizontal position of the trigger on the screen.

Trigger Type	Trigger Parameters
`Timeout` — Trigger when a signal stays above or below a threshold longer than a specified time	**Polarity** — Select the polarity for a timeout-triggered signal. • `Rising` — Trigger when the signal does not cross the threshold from below. For example, if you set **Timeout** to 7.50 seconds, the scope triggers 7.50 seconds after the signal crosses the threshold. • `Falling` — Trigger when the signal does not cross the threshold from above. • `Either` — Trigger when the signal does not cross the threshold from either direction **Level** — Enter a threshold value for a timeout-triggered signal. **Hysteresis** — Enter a value for a timeout-triggered signal. See "Hysteresis of Trigger Signals" on page 4-67. **Timeout** — Enter a time duration for a timeout-triggered signal. Alternatively, a trigger event can occur when the signal stays within the boundaries defined by the hysteresis for 7.50 seconds after the signal crosses the threshold.

Trigger Type	Trigger Parameters
	Delay — Offset the trigger by a fixed delay in seconds.
	Holdoff — Set the minimum possible time between triggers.
	Position — Set horizontal position of the trigger on the screen.
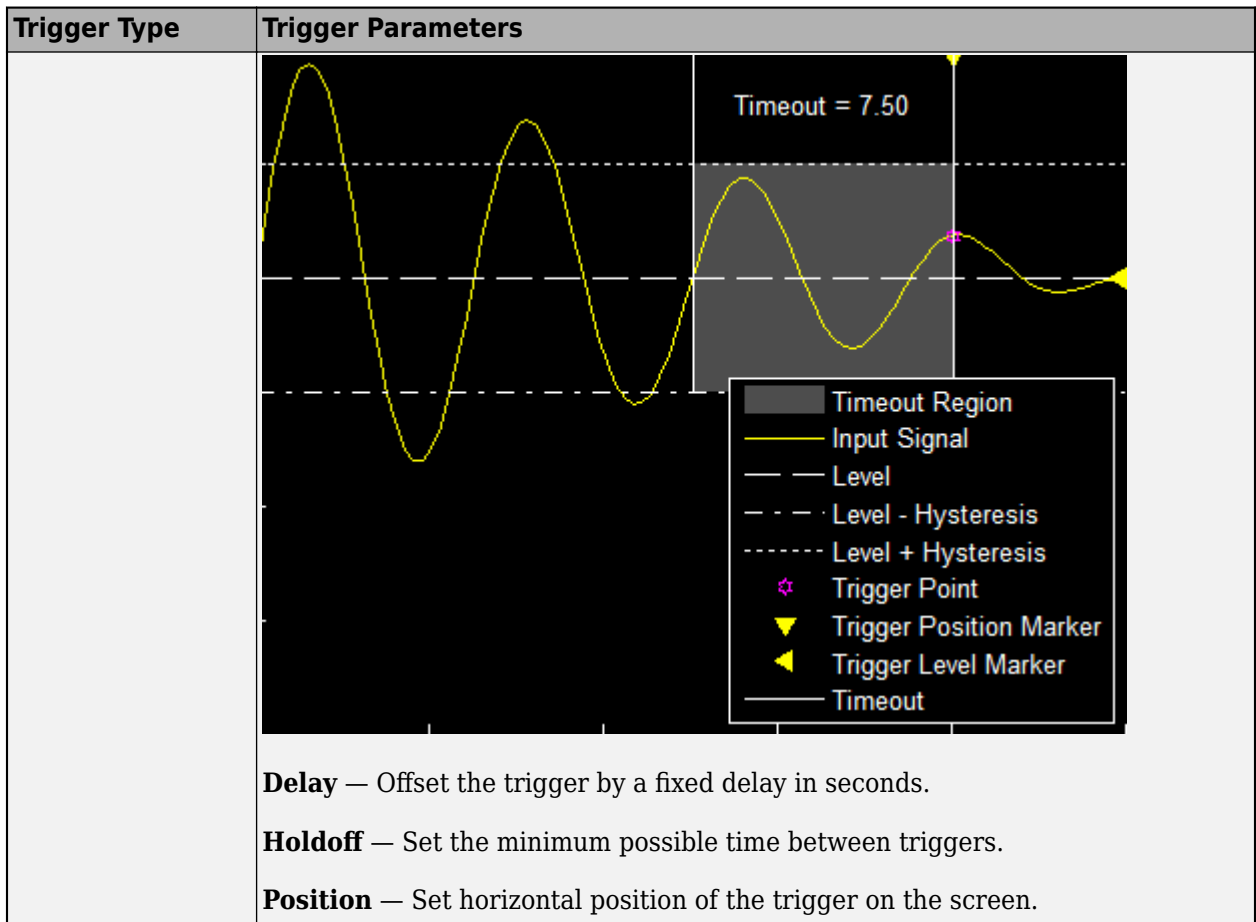
Hysteresis of Trigger Signals

Hysteresis — Specify the hysteresis or noise-reject value. This parameter is visible when you set **Type** to `Edge` or `Timeout`. If the signal jitters inside this range and briefly crosses the trigger level, the scope does not register an event. In the case of an edge trigger with rising polarity, the scope ignores the times that a signal crosses the trigger level within the hysteresis region.
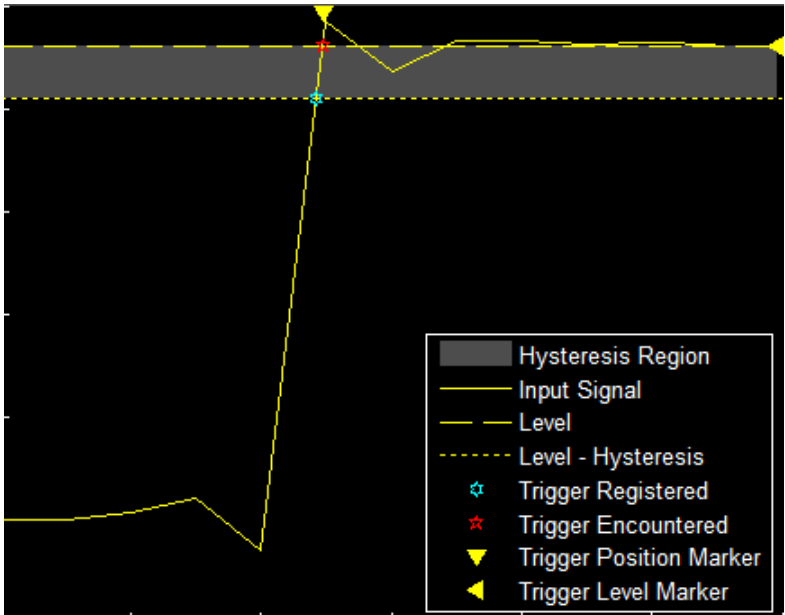
You can reduce the hysteresis region size by decreasing the hysteresis value. In this example, if you set the hysteresis value to 0.07, the scope also considers the second rising edge to be a trigger event.
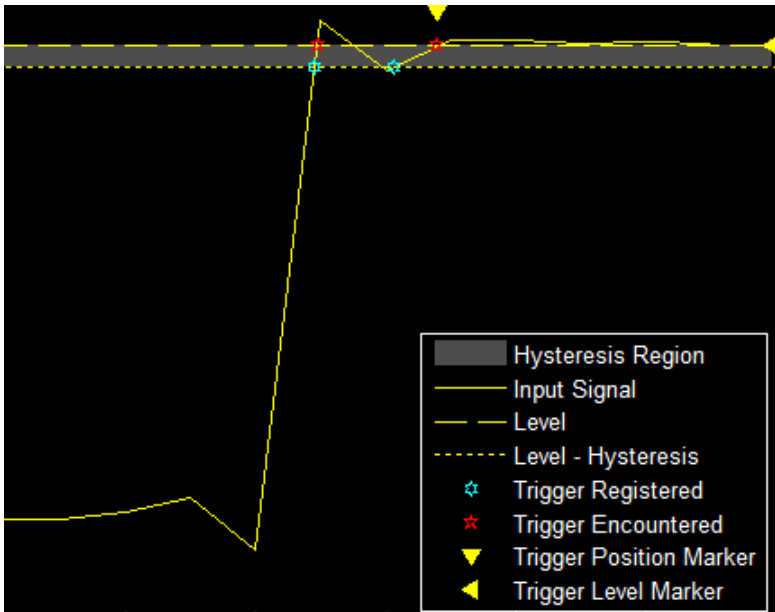
Share or Save the Time Scope

If you want to save the time scope for future use or share it with others, use the buttons in the **Share** section of the **Scope** tab.

- **Generate Script** — Generate a script to re-create your time scope with the same settings. An editor window opens with the code required to re-create your `timescope` object.

- **Copy Display** — Copy the display to your clipboard. You can paste the image in another program to save or share it.
- **Print** — Opens a print dialog box from which you can print out the plot image.

Scale Axes

To scale the plot axes, you can use the mouse to pan around the axes and the scroll button on your mouse to zoom in and out of the plot. Additionally, you can use the buttons that appear when you hover over the plot window.

- — Maximize the axes, hiding all labels and insetting the axes values.

- — Zoom in on the plot.

- — Pan the plot.

- — Autoscale the axes to fit the shown data.

See Also

Objects
timescope

Featured Examples

Air Traffic Control

This example shows how to generate an air traffic control scenario, simulate radar detections from an airport surveillance radar (ASR), and configure a global nearest neighbor (GNN) tracker to track the simulated targets using the radar detections. This enables you to evaluate different target scenarios, radar requirements, and tracker configurations without needing access to costly aircraft or equipment. This example covers the entire synthetic data workflow.

Air Traffic Control Scenario

Simulate an air traffic control (ATC) tower and moving targets in the scenario as *platforms*. Simulation of the motion of the platforms in the scenario is managed by `trackingScenario`.

Create a `trackingScenario` and add the ATC tower to the scenario.

```
% Create tracking scenario
scenario = trackingScenario;

% Add a stationary platform to model the ATC tower
tower = platform(scenario);
```

Airport Surveillance Radar

Add an airport surveillance radar (ASR) to the ATC tower. A typical ATC tower has a radar mounted 15 meters above the ground. This radar scans mechanically in azimuth at a fixed rate to provide 360 degree coverage in the vicinity of the ATC tower. Common specifications for an ASR are listed:

- Sensitivity: 0 dBsm @ 111 km
- Mechanical Scan: Azimuth only
- Mechanical Scan Rate: 12.5 RPM
- Electronic Scan: None
- Field of View: 1.4 deg azimuth, 10 deg elevation
- Azimuth Resolution: 1.4 deg
- Range Resolution: 135 m

Model the ASR with the above specifications using the `monostaticRadarSensor`.

```
rpm = 12.5;
fov = [1.4;10];
scanrate = rpm*360/60;  % deg/s
updaterate = scanrate/fov(1); % Hz

radar = monostaticRadarSensor(1, 'Rotator', ...
    'UpdateRate', updaterate, ...          % Hz
    'FieldOfView', fov, ...                % [az;el] deg
    'MaxMechanicalScanRate', scanrate, ... % deg/sec
    'AzimuthResolution', fov(1), ...       % deg
    'ReferenceRange', 111e3, ...           % m
    'ReferenceRCS', 0, ...                 % dBsm
    'RangeResolution', 135, ...            % m
    'HasINS', true, ...
    'DetectionCoordinates', 'Scenario');

% Mount radar at the top of the tower
```

```
radar.MountingLocation = [0 0 -15];
tower.Sensors = radar;
```

Tilt the radar so that it surveys a region beginning at 2 degrees above the horizon. To do this, enable elevation and set the mechanical scan limits to span the radar's elevation field of view beginning at 2 degrees above the horizon. Because `trackingScenario` uses a North-East-Down (NED) coordinate frame, negative elevations correspond to points above the horizon.

```
% Enable elevation scanning
radar.HasElevation = true;

% Set mechanical elevation scan to begin at 2 degrees above the horizon
elFov = fov(2);
tilt = 2; % deg
radar.MechanicalScanLimits(2,:) = [-fov(2) 0]-tilt; % deg
```

Set the elevation field of view to be slightly larger than the elevation spanned by the scan limits. This prevents raster scanning in elevation and tilts the radar to point in the middle of the elevation scan limits.

```
radar.FieldOfView(2) = elFov+1e-3;
```

The `monostaticRadarSensor` models range and elevation bias due to atmospheric refraction. These biases become more pronounced at lower altitudes and for targets at long ranges. Because the index of refraction changes (decreases) with altitude, the radar signals propagate along a curved path. This results in the radar observing targets at altitudes which are higher than their true altitude and at ranges beyond their line-of-sight range.

Add three airliners within the ATC control sector. One airliner approaches the ATC from a long range, another departs, and the third is flying tangential to the tower. Model the motion of these airliners over a 60 second interval.

`trackingScenario` uses a North-East-Down (NED) coordinate frame. When defining the waypoints for the airliners below, the z-coordinate corresponds to down, so heights above the ground are set to negative values.

```
% Duration of scenario
sceneDuration = 60; % s

% Inbound airliner
ht = 3e3;
spd = 900*1e3/3600; % m/s
wp = [-5e3 -40e3 -ht;-5e3 -40e3+spd*sceneDuration -ht];
traj = waypointTrajectory('Waypoints',wp,'TimeOfArrival',[0 sceneDuration]);
platform(scenario,'Trajectory', traj);

% Outbound airliner
ht = 4e3;
spd = 700*1e3/3600; % m/s
wp = [20e3 10e3 -ht;20e3+spd*sceneDuration 10e3 -ht];
traj = waypointTrajectory('Waypoints',wp,'TimeOfArrival',[0 sceneDuration]);
platform(scenario,'Trajectory', traj);

% Tangential airliner
ht = 4e3;
spd = 300*1e3/3600; % m/s
wp = [-20e3 -spd*sceneDuration/2 -ht;-20e3 spd*sceneDuration/2 -ht];
```

```
traj = waypointTrajectory('Waypoints',wp,'TimeOfArrival',[0 sceneDuration]);
platform(scenario,'Trajectory', traj);
```

GNN Tracker

Create a `trackerGNN` to form tracks from the radar detections generated from the three airliners. Update the tracker with the detections generated after the completion of a full 360 degree scan in azimuth.

The tracker uses the `initFilter` supporting function to initialize a constant velocity extended Kalman filter for each new track. `initFilter` modifies the filter returned by `initcvekf` to match the target velocities and tracker update interval.

```
tracker = trackerGNN( ...
    'Assignment', 'Auction', ...
    'AssignmentThreshold',50, ...
    'FilterInitializationFcn',@initFilter);
```

Visualize on a Map

You use `helperATCMap` to visualize the results on top of a map display. You position the origin of the local North-East-Down (NED) coordinate system used by the tower radar and tracker at the position of Logan airport in Boston. The origin is located at 42.36306 latitude and -71.00639 longitude and 50 meters above the sea level. The helper provides the necessary utilities to convert the coordinates from NED to the Earth-centered coordinate frame used by the map.

```
origin = [42.366978, -71.022362, 50];
mapViewer = helperATCMap('ReferenceLocation',origin);
setCamera(mapViewer, origin + [0 0 1e5]);
showScenario(mapViewer,scenario);
snap(mapViewer);
```
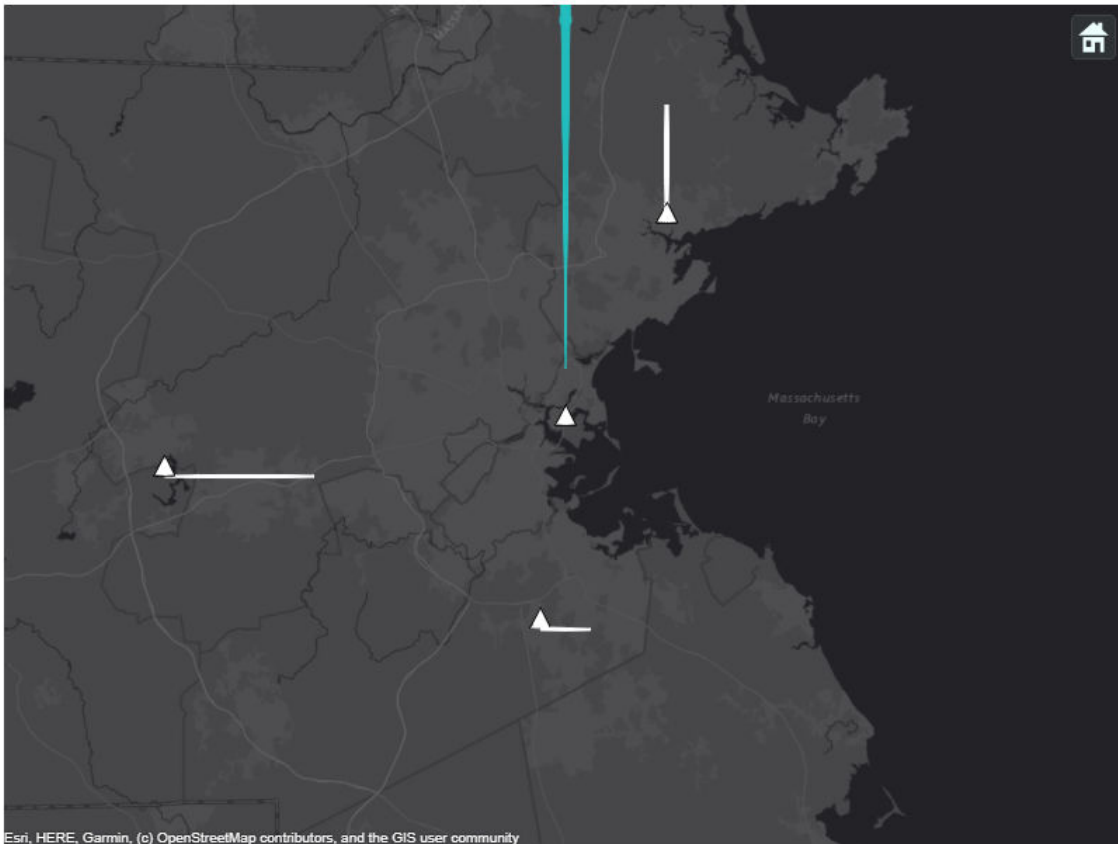
Simulate and Track Airliners

The following loop advances the platform positions until the end of the scenario has been reached. For each step forward in the scenario, the radar generates detections from targets in its field of view. The tracker is updated with these detections after the radar has completed a 360 degree scan in azimuth.

```
% Set simulation to advance at the update rate of the radar
scenario.UpdateRate = radar.UpdateRate;

% Create a buffer to collect the detections from a full scan of the radar
scanBuffer = {};

% Initialize the track array
tracks = [];

% Save visualization snapshots for each scan
allsnaps = {};
scanCount = 0;

% Set random seed for repeatable results
s = rng;
rng(2020)
```

```matlab
while advance(scenario)

    % Update airliner positions
    plotTarget(mapViewer, scenario.Platforms([2 3 4]));

    % Generate detections on targets in the radar's current field of view
    [dets,config] = detect(scenario);
    scanBuffer = [scanBuffer;dets]; %#ok<AGROW>
    % Plot beam and detections
    plotCoverage(mapViewer,coverageConfig(scenario))
    plotDetection(mapViewer,scanBuffer);


    % Update tracks when a 360 degree scan is complete
    simTime = scenario.SimulationTime;
    isScanDone = config.IsScanDone;
    if isScanDone
        scanCount = scanCount+1;
        % Update tracker
        [tracks,~,~,info] = tracker(scanBuffer,simTime);
        % Clear scan buffer for next scan
        scanBuffer = {};
    elseif isLocked(tracker)
        % Predict tracks to the current simulation time
        tracks = predictTracksToTime(tracker,'confirmed',simTime);
    end

    % Update map and take snapshots
    allsnaps = snapPlotTrack(mapViewer,tracks,isScanDone, scanCount, allsnaps);

end
allsnaps = [allsnaps, {snap(mapViewer)}];
```

Show the first snapshot taken at the completion of the radar's second scan.

```matlab
figure
imshow(allsnaps{1});
```

The preceding figure shows the scenario at the end of the radar's second 360 degree scan. Radar detections, shown as light blue dots, are present for each of the simulated airliners. At this point, the tracker has already been updated by one complete scan of the radar. Internally, the tracker has initialized tracks for each of the airliners. These tracks will be shown after the update following this scan, when the tracks are promoted to confirmed, meeting the tracker's confirmation requirement of 2 hits out of 3 updates.

The next two snapshots show tracking of the outbound airliner.

```
figure
imshow(allsnaps{2});
```

```
figure
imshow(allsnaps{3});
```

The previous figures show the track picture before and immediately after the tracker updates after the radar's second scan. The detection in the figure before the tracker update is used to update and confirm the initialized track from the previous scan's detection for this airliner. The next figure shows the confirmed track's position and velocity. The uncertainty of the track's position estimate is shown as the gray ellipse. After only two detections, the tracker has established an accurate estimate of the outbound airliner's position and velocity. The airliner's true altitude is 4 km and it is traveling east at 700 km/hr.

```
figure
imshow(allsnaps{4});
```
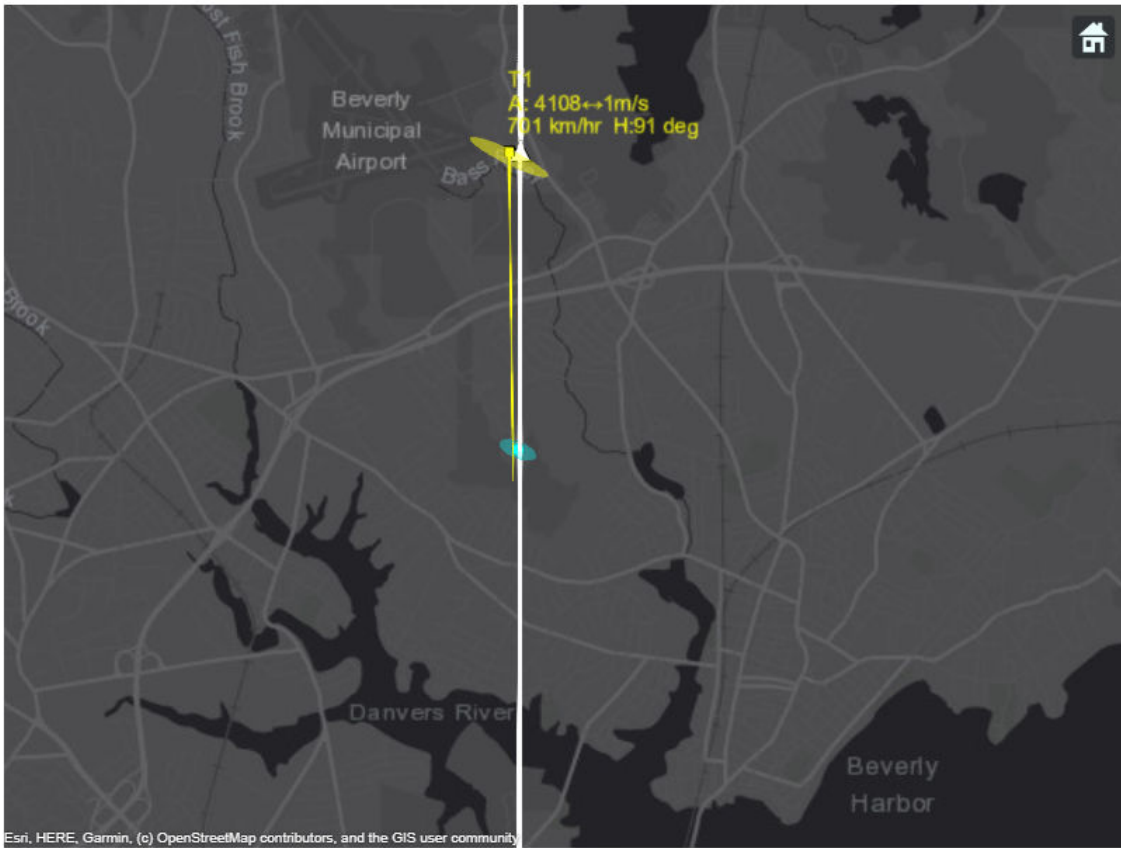
```
figure
imshow(allsnaps{5});
```
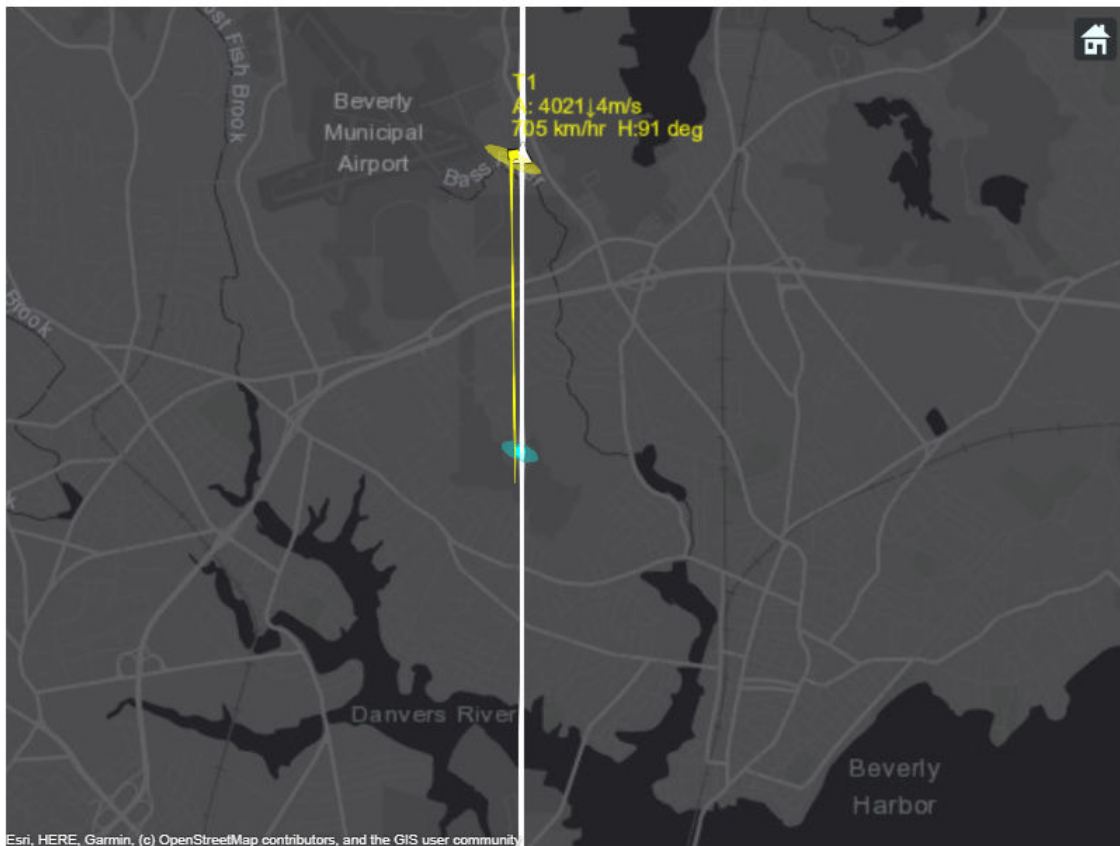
The state of the outbound airliner's track is coasted to the end of the third scan and shown in the figure above along with the most recent detection for the airliner. Notice how the track's uncertainty has grown since it was updated in the previous figure. The track after it has been updated with the detection is shown in the next figure. You notice that the uncertainty of the track position is reduced after the update. The track uncertainty grows between updates and is reduced whenever it is updated with a new measurement. You also observe that after the third update, the track lies on top of the airliner's true position.

```
figure
imshow(allsnaps{6});
```
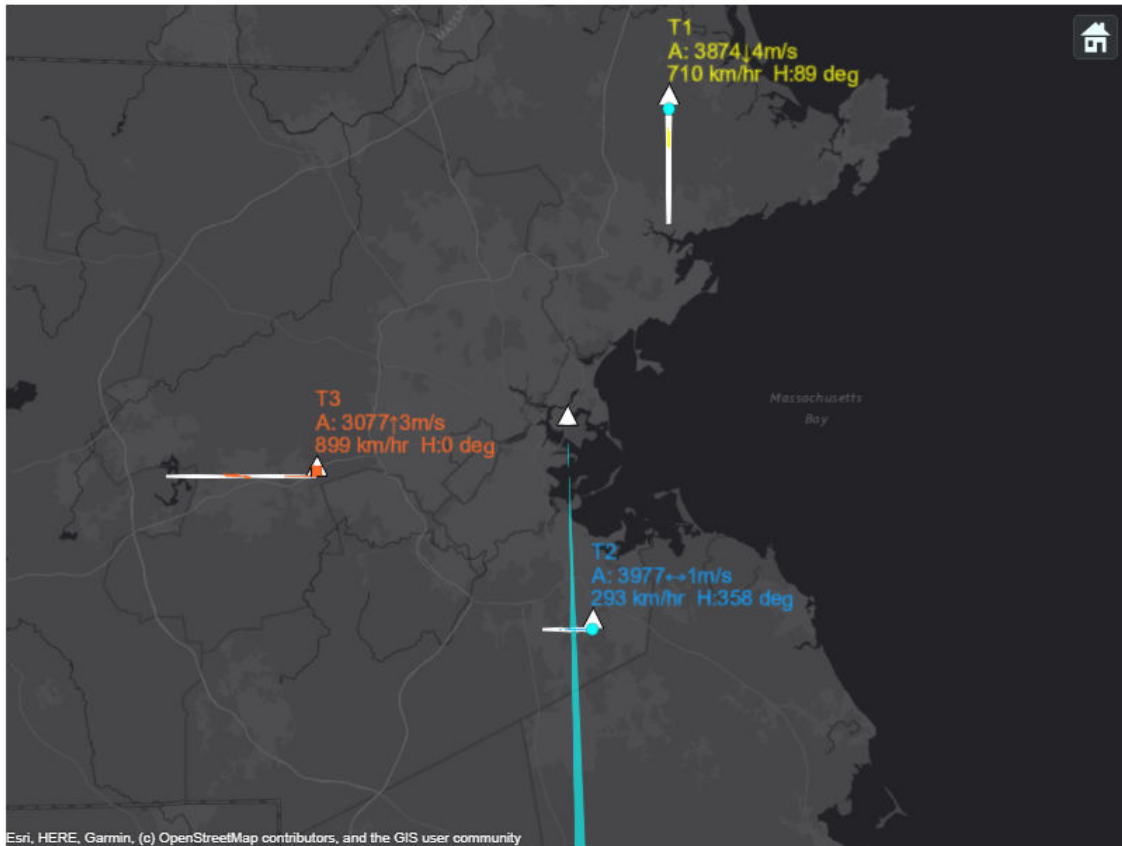
The final figure shows the state of all three airliners' tracks at the end of the scenario. There is exactly one track for each of the three airliners. The same track numbers are assigned to each of the airliners for the entire duration of the scenario, indicating that none of these tracks were dropped during the scenario. The estimated tracks closely match the true position and velocity of the airliners.

```
truthTrackTable = tabulateData(scenario, tracks) %#ok<NOPTS>
```

truthTrackTable=*3×4 table*

TrackID	Altitude		Heading		Speed	
	True	Estimated	True	Estimated	True	Estimated
"T1"	4000	3874	90	89	700	710
"T2"	4000	3977	0	358	300	293
"T3"	3000	3077	0	0	900	899

Visualize tracks in 3D to get a better sense of the estimated altitudes.

```
% Reposition and orient the camera to show the 3-D nature of the map
camPosition = origin + [0.367, 0.495, 1.5e4];
camOrientation = [0, -17, 235]; %Looking south west, 17 degrees below the horizon
setCamera(mapViewer, camPosition, camOrientation);
```

The figure below shows a 3-D map of the scenario. You can see the simulated jets in white triangles with their trajectories depicted as white lines. The radar beam is shown as a blue cone with blue dots representing radar detections. The tracks are shown in yellow, orange, and blue and their information is listed in their respective color. Due to the nature of the 3-D display, some of the markers may be hidden behind others.

You can use the following controls on the map to get different views of the scene:

- To pan the map, you left click on the mouse and drag the map.
- To rotate the map, while holding the ctrl button, you left click on the mouse and drag the map.
- To zoom the map in and out, you use the mouse scroll wheel.

`snap(mapViewer);`
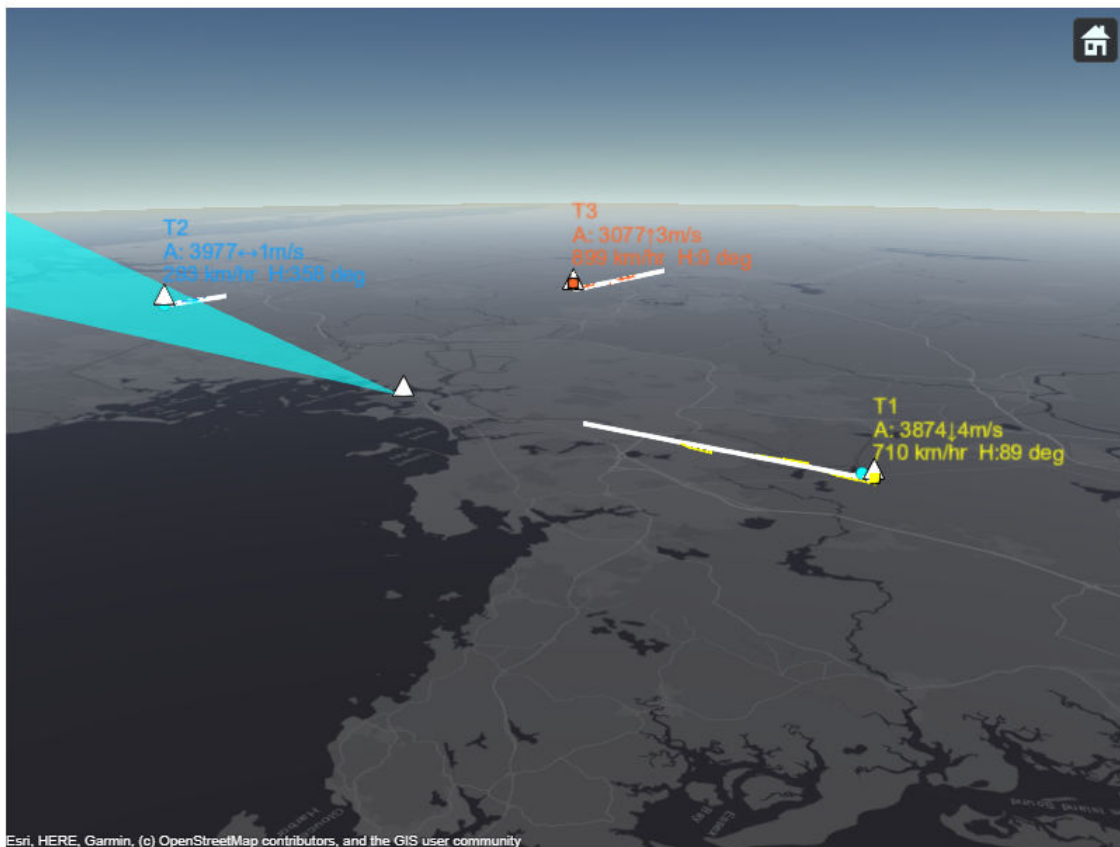
Summary

This example shows how to generate an air traffic control scenario, simulate radar detections from an airport surveillance radar (ASR), and configure a global nearest neighbor (GNN) tracker to track the simulated targets using the radar detections. In this example, you learned how the tracker's history based logic promotes tracks. You also learned how the track uncertainty grows when a track is coasted and is reduced when the track is updated by a new detection.

Supporting Functions

initFilter

This function modifies the function `initcvekf` to handle higher velocity targets such as the airliners in the ATC scenario.

```
function filter = initFilter(detection)
filter = initcvekf(detection);
classToUse = class(filter.StateCovariance);

% Airliners can move at speeds around 900 km/h. The velocity is
% initialized to 0, but will need to be able to quickly adapt to
% aircraft moving at these speeds. Use 900 km/h as 1 standard deviation
% for the initialized track's velocity noise.
spd = 900*1e3/3600; % m/s
velCov = cast(spd^2,classToUse);
cov = filter.StateCovariance;
cov(2,2) = velCov;
cov(4,4) = velCov;
filter.StateCovariance = cov;

% Set filter's process noise to match filter's update rate
scaleAccelHorz = cast(1,classToUse);
scaleAccelVert = cast(1,classToUse);
Q = blkdiag(scaleAccelHorz^2, scaleAccelHorz^2, scaleAccelVert^2);
filter.ProcessNoise = Q;
end
```

tabulateData

This function returns a table comparing the ground truth and tracks

```
function truthTrackTable = tabulateData(scenario, tracks)
% Process truth data
platforms = scenario.Platforms(2:end); % Platform 1 is the radar
numPlats = numel(platforms);
trueAlt = zeros(numPlats,1);
trueSpd = zeros(numPlats,1);
trueHea = zeros(numPlats,1);
for i = 1:numPlats
    traj = platforms{i}.Trajectory;
    waypoints = traj.Waypoints;
    times = traj.TimeOfArrival;
    trueAlt(i) = -waypoints(end,3);
    trueVel = (waypoints(end,:) - waypoints(end-1,:)) / (times(end)-times(end-1));
    trueSpd(i) = norm(trueVel) * 3600 / 1000; % Convert to km/h
    trueHea(i) = atan2d(trueVel(1),trueVel(2));
end
trueHea = mod(trueHea,360);

% Associate tracks with targets
atts = [tracks.ObjectAttributes];
tgtInds = [atts.TargetIndex];

% Process tracks assuming a constant velocity model
numTrks = numel(tracks);
estAlt = zeros(numTrks,1);
```

```
estSpd = zeros(numTrks,1);
estHea = zeros(numTrks,1);
truthTrack = zeros(numTrks,7);
for i = 1:numTrks
    estAlt(i) = -round(tracks(i).State(5));
    estSpd(i) = round(norm(tracks(i).State(2:2:6)) * 3600 / 1000); % Convert to km/h;
    estHea(i) = round(atan2d(tracks(i).State(2),tracks(i).State(4)));
    estHea(i) = mod(estHea(i),360);
    platID = tgtInds(i);
    platInd = platID - 1;
    truthTrack(i,:) = [tracks(i).TrackID, trueAlt(platInd), estAlt(i), trueHea(platInd), estHea(i
        trueSpd(platInd), estSpd(i)];
end

% Organize the data in a table
names = {'TrackID','TrueAlt','EstimatedAlt','TrueHea','EstimatedHea','TrueSpd','EstimatedSpd'};
truthTrackTable = array2table(truthTrack,'VariableNames',names);
truthTrackTable = mergevars(truthTrackTable, (6:7), 'NewVariableName', 'Speed', 'MergeAsTable', 
truthTrackTable.(6).Properties.VariableNames = {'True','Estimated'};
truthTrackTable = mergevars(truthTrackTable, (4:5), 'NewVariableName', 'Heading', 'MergeAsTable'
truthTrackTable.(4).Properties.VariableNames = {'True','Estimated'};
truthTrackTable = mergevars(truthTrackTable, (2:3), 'NewVariableName', 'Altitude', 'MergeAsTable
truthTrackTable.(2).Properties.VariableNames = {'True','Estimated'};
truthTrackTable.TrackID = "T" + string(truthTrackTable.TrackID);
end
```

snapPlotTrack

This function handles moving the `mapViewer` camera, taking relevant snapshots and updating track visuals.

```
function allsnaps = snapPlotTrack(mapViewer,tracks,isScanDone, scanCount,allsnaps)
% Save snapshots during first 4 scans
if isScanDone && any(scanCount == [2 3])
    allsnaps = [allsnaps, {snap(mapViewer)}];
    %move camera
    if scanCount == 2
        % Show the outbound airliner
        setCamera(mapViewer, [42.5650  -70.8990 7e3]);
        allsnaps = [allsnaps, {snap(mapViewer)}];
    end

end

% Update display with current track positions
plotTrack(mapViewer,tracks);

if isScanDone && any(scanCount == [2 3])
    % Take a snapshot of confirmed track
    allsnaps = [allsnaps, {snap(mapViewer)}];
    % Reset Camera view to full scene
    if scanCount == 3
        origin = [42.366978, -71.022362, 50];
        setCamera(mapViewer, origin + [0 0 1e5]);
    end
end
end
```

IMU and GPS Fusion for Inertial Navigation

This example shows how you might build an IMU + GPS fusion algorithm suitable for unmanned aerial vehicles (UAVs) or quadcopters.

This example uses accelerometers, gyroscopes, magnetometers, and GPS to determine orientation and position of a UAV.

Simulation Setup

Set the sampling rates. In a typical system, the accelerometer and gyroscope run at relatively high sample rates. The complexity of processing data from those sensors in the fusion algorithm is relatively low. Conversely, the GPS, and in some cases the magnetometer, run at relatively low sample rates, and the complexity associated with processing them is high. In this fusion algorithm, the magnetometer and GPS samples are processed together at the same low rate, and the accelerometer and gyroscope samples are processed together at the same high rate.

To simulate this configuration, the IMU (accelerometer, gyroscope, and magnetometer) are sampled at 160 Hz, and the GPS is sampled at 1 Hz. Only one out of every 160 samples of the magnetometer is given to the fusion algorithm, so in a real system the magnetometer could be sampled at a much lower rate.

```
imuFs = 160;
gpsFs = 1;

% Define where on the Earth this simulated scenario takes place using the
% latitude, longitude and altitude.
refloc = [42.2825 -72.3430 53.0352];


% Validate that the |gpsFs| divides |imuFs|. This allows the sensor sample
% rates to be simulated using a nested for loop without complex sample rate
% matching.

imuSamplesPerGPS = (imuFs/gpsFs);
assert(imuSamplesPerGPS == fix(imuSamplesPerGPS), ...
    'GPS sampling rate must be an integer factor of IMU sampling rate.');
```

Fusion Filter

Create the filter to fuse IMU + GPS measurements. The fusion filter uses an extended Kalman filter to track orientation (as a quaternion), velocity, position, sensor biases, and the geomagnetic vector.

This `insfilterMARG` has a few methods to process sensor data, including `predict`, `fusemag` and `fusegps`. The `predict` method takes the accelerometer and gyroscope samples from the IMU as inputs. Call the `predict` method each time the accelerometer and gyroscope are sampled. This method predicts the states one time step ahead based on the accelerometer and gyroscope. The error covariance of the extended Kalman filter is updated here.

The `fusegps` method takes GPS samples as input. This method updates the filter states based on GPS samples by computing a Kalman gain that weights the various sensor inputs according to their uncertainty. An error covariance is also updated here, this time using the Kalman gain as well.

The `fusemag` method is similar but updates the states, Kalman gain, and error covariance based on the magnetometer samples.

Though the `insfilterMARG` takes accelerometer and gyroscope samples as inputs, these are integrated to compute delta velocities and delta angles, respectively. The filter tracks the bias of the magnetometer and these integrated signals.

```
fusionfilt = insfilterMARG;
fusionfilt.IMUSampleRate = imuFs;
fusionfilt.ReferenceLocation = refloc;
```

UAV Trajectory

This example uses a saved trajectory recorded from a UAV as the ground truth. This trajectory is fed to several sensor simulators to compute simulated accelerometer, gyroscope, magnetometer, and GPS data streams.

```
% Load the "ground truth" UAV trajectory.
load LoggedQuadcopter.mat trajData;
trajOrient = trajData.Orientation;
trajVel = trajData.Velocity;
trajPos = trajData.Position;
trajAcc = trajData.Acceleration;
trajAngVel = trajData.AngularVelocity;

% Initialize the random number generator used in the simulation of sensor
% noise.
rng(1)
```

GPS Sensor

Set up the GPS at the specified sample rate and reference location. The other parameters control the nature of the noise in the output signal.

```
gps = gpsSensor('UpdateRate', gpsFs);
gps.ReferenceLocation = refloc;
gps.DecayFactor = 0.5;                   % Random walk noise parameter
gps.HorizontalPositionAccuracy = 1.6;
gps.VerticalPositionAccuracy =  1.6;
gps.VelocityAccuracy = 0.1;
```

IMU Sensors

Typically, a UAV uses an integrated MARG sensor (Magnetic, Angular Rate, Gravity) for pose estimation. To model a MARG sensor, define an IMU sensor model containing an accelerometer, gyroscope, and magnetometer. In a real-world application the three sensors could come from a single integrated circuit or separate ones. The property values set here are typical for low-cost MEMS sensors.

```
imu = imuSensor('accel-gyro-mag', 'SampleRate', imuFs);
imu.MagneticField = [19.5281 -5.0741 48.0067];

% Accelerometer
imu.Accelerometer.MeasurementRange =  19.6133;
imu.Accelerometer.Resolution = 0.0023928;
imu.Accelerometer.ConstantBias = 0.19;
imu.Accelerometer.NoiseDensity = 0.0012356;

% Gyroscope
imu.Gyroscope.MeasurementRange = deg2rad(250);
imu.Gyroscope.Resolution = deg2rad(0.0625);
```

```
imu.Gyroscope.ConstantBias = deg2rad(3.125);
imu.Gyroscope.AxesMisalignment = 1.5;
imu.Gyroscope.NoiseDensity = deg2rad(0.025);

% Magnetometer
imu.Magnetometer.MeasurementRange = 1000;
imu.Magnetometer.Resolution = 0.1;
imu.Magnetometer.ConstantBias = 100;
imu.Magnetometer.NoiseDensity = 0.3/ sqrt(50);
```

Initialize the State Vector of the `insfilterMARG`

The `insfilterMARG` tracks the pose states in a 22-element vector. The states are:

```
    State                           Units        State Vector Index
 Orientation as a quaternion                     1:4
 Position (NED)                     m            5:7
 Velocity (NED)                     m/s          8:10
 Delta Angle Bias (XYZ)            rad          11:13
 Delta Velocity Bias (XYZ)         m/s          14:16
 Geomagnetic Field Vector (NED)    uT           17:19
 Magnetometer Bias (XYZ)           uT           20:22
```

Ground truth is used to help initialize the filter states, so the filter converges to good answers quickly.

```
% Initialize the states of the filter

initstate = zeros(22,1);
initstate(1:4) = compact( meanrot(trajOrient(1:100)));
initstate(5:7) = mean( trajPos(1:100,:), 1);
initstate(8:10) = mean( trajVel(1:100,:), 1);
initstate(11:13) =  imu.Gyroscope.ConstantBias./imuFs;
initstate(14:16) =  imu.Accelerometer.ConstantBias./imuFs;
initstate(17:19) =  imu.MagneticField;
initstate(20:22) = imu.Magnetometer.ConstantBias;

fusionfilt.State = initstate;
```

Initialize the Variances of the `insfilterMARG`

The `insfilterMARG` measurement noises describe how much noise is corrupting the sensor reading. These values are based on the `imuSensor` and `gpsSensor` parameters.

The process noises describe how well the filter equations describe the state evolution. Process noises are determined empirically using parameter sweeping to jointly optimize position and orientation estimates from the filter.

```
% Measurement noises
Rmag = 0.09; % Magnetometer measurement noise
Rvel = 0.01; % GPS Velocity measurement noise
Rpos = 2.56; % GPS Position measurement noise

% Process noises
fusionfilt.AccelerometerBiasNoise =  2e-4;
fusionfilt.AccelerometerNoise = 2;
fusionfilt.GyroscopeBiasNoise = 1e-16;
fusionfilt.GyroscopeNoise =  1e-5;
fusionfilt.MagnetometerBiasNoise = 1e-10;
```

```
fusionfilt.GeomagneticVectorNoise = 1e-12;

% Initial error covariance
fusionfilt.StateCovariance = 1e-9*ones(22);
```

Initialize Scopes

The `HelperScrollingPlotter` scope enables plotting of variables over time. It is used here to track errors in pose. The `HelperPoseViewer` scope allows 3-D visualization of the filter estimate and ground truth pose. The scopes can slow the simulation. To disable a scope, set the corresponding logical variable to `false`.

```
useErrScope = true;  % Turn on the streaming error plot
usePoseView = true;  % Turn on the 3-D pose viewer

if useErrScope
    errscope = HelperScrollingPlotter(...
        'NumInputs', 4, ...
        'TimeSpan', 10, ...
        'SampleRate', imuFs, ...
        'YLabel', {'degrees', ...
        'meters', ...
        'meters', ...
        'meters'}, ...
        'Title', {'Quaternion Distance', ...
        'Position X Error', ...
        'Position Y Error', ...
        'Position Z Error'}, ...
        'YLimits', ...
        [ -1, 1
        -2, 2
        -2 2
        -2 2]);
end

if usePoseView
    posescope = HelperPoseViewer(...
        'XPositionLimits', [-15 15], ...
        'YPositionLimits', [-15, 15], ...
        'ZPositionLimits', [-10 10]);
end
```

Simulation Loop

The main simulation loop is a while loop with a nested for loop. The while loop executes at `gpsFs`, which is the GPS sample rate. The nested for loop executes at `imuFs`, which is the IMU sample rate. The scopes are updated at the IMU sample rate.

```
% Loop setup - |trajData| has about 142 seconds of recorded data.
secondsToSimulate = 50; % simulate about 50 seconds
numsamples = secondsToSimulate*imuFs;

loopBound = floor(numsamples);
loopBound = floor(loopBound/imuFs)*imuFs; % ensure enough IMU Samples

% Log data for final metric computation.
pqorient = quaternion.zeros(loopBound,1);
pqpos = zeros(loopBound,3);
```

```matlab
fcnt = 1;

while(fcnt <=loopBound)
    % |predict| loop at IMU update frequency.
    for ff=1:imuSamplesPerGPS
        % Simulate the IMU data from the current pose.
        [accel, gyro, mag] = imu(trajAcc(fcnt,:), trajAngVel(fcnt, :), ...
            trajOrient(fcnt));

        % Use the |predict| method to estimate the filter state based
        % on the simulated accelerometer and gyroscope signals.
        predict(fusionfilt, accel, gyro);

        % Acquire the current estimate of the filter states.
        [fusedPos, fusedOrient] = pose(fusionfilt);

        % Save the position and orientation for post processing.
        pqorient(fcnt) = fusedOrient;
        pqpos(fcnt,:) = fusedPos;

        % Compute the errors and plot.
        if useErrScope
            orientErr = rad2deg(dist(fusedOrient, ...
                trajOrient(fcnt) ));
            posErr = fusedPos - trajPos(fcnt,:);
            errscope(orientErr, posErr(1), posErr(2), posErr(3));
        end

        % Update the pose viewer.
        if usePoseView
            posescope(pqpos(fcnt,:), pqorient(fcnt),  trajPos(fcnt,:), ...
                trajOrient(fcnt,:) );
        end
        fcnt = fcnt + 1;
    end

    % This next step happens at the GPS sample rate.
    % Simulate the GPS output based on the current pose.
    [lla, gpsvel] = gps( trajPos(fcnt,:), trajVel(fcnt,:) );

    % Correct the filter states based on the GPS data and magnetic
    % field measurements.
    fusegps(fusionfilt, lla, Rpos, gpsvel, Rvel);
    fusemag(fusionfilt, mag, Rmag);

end
```
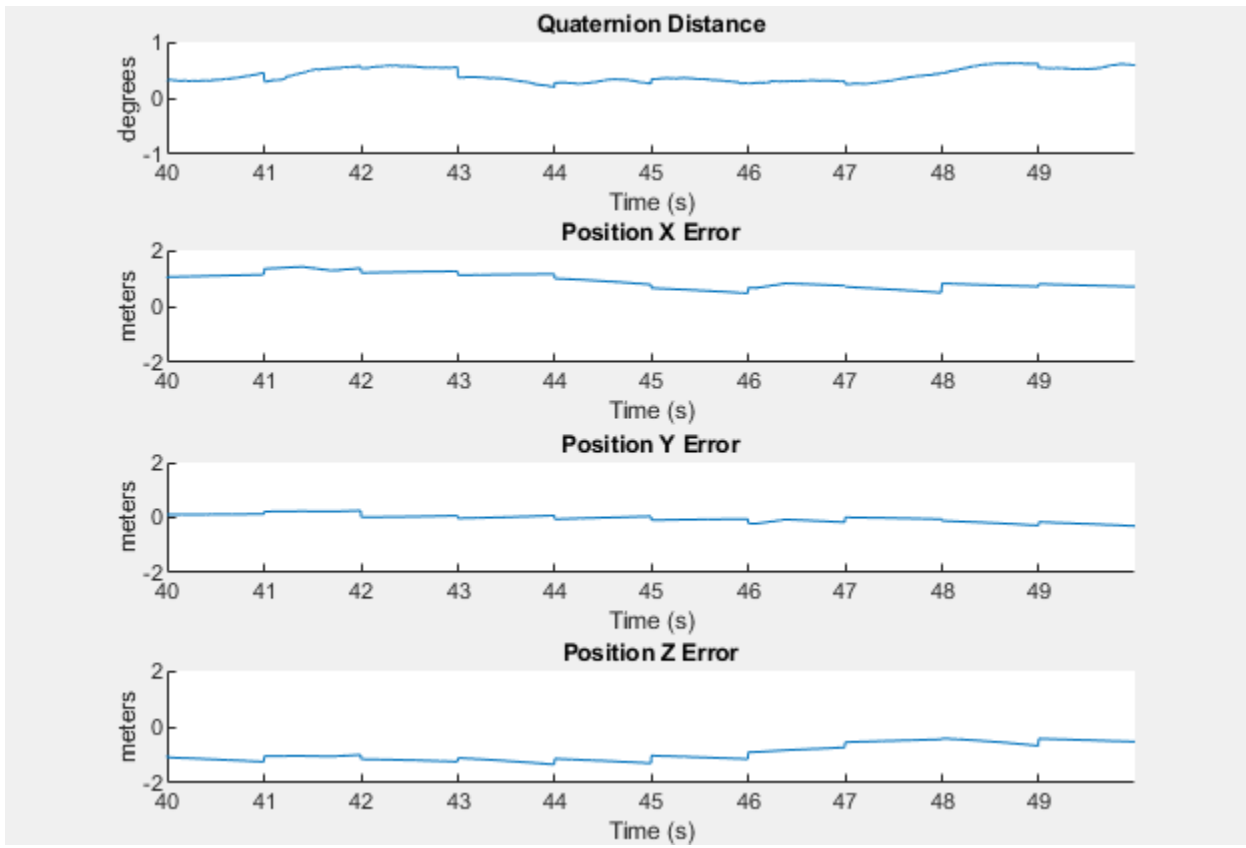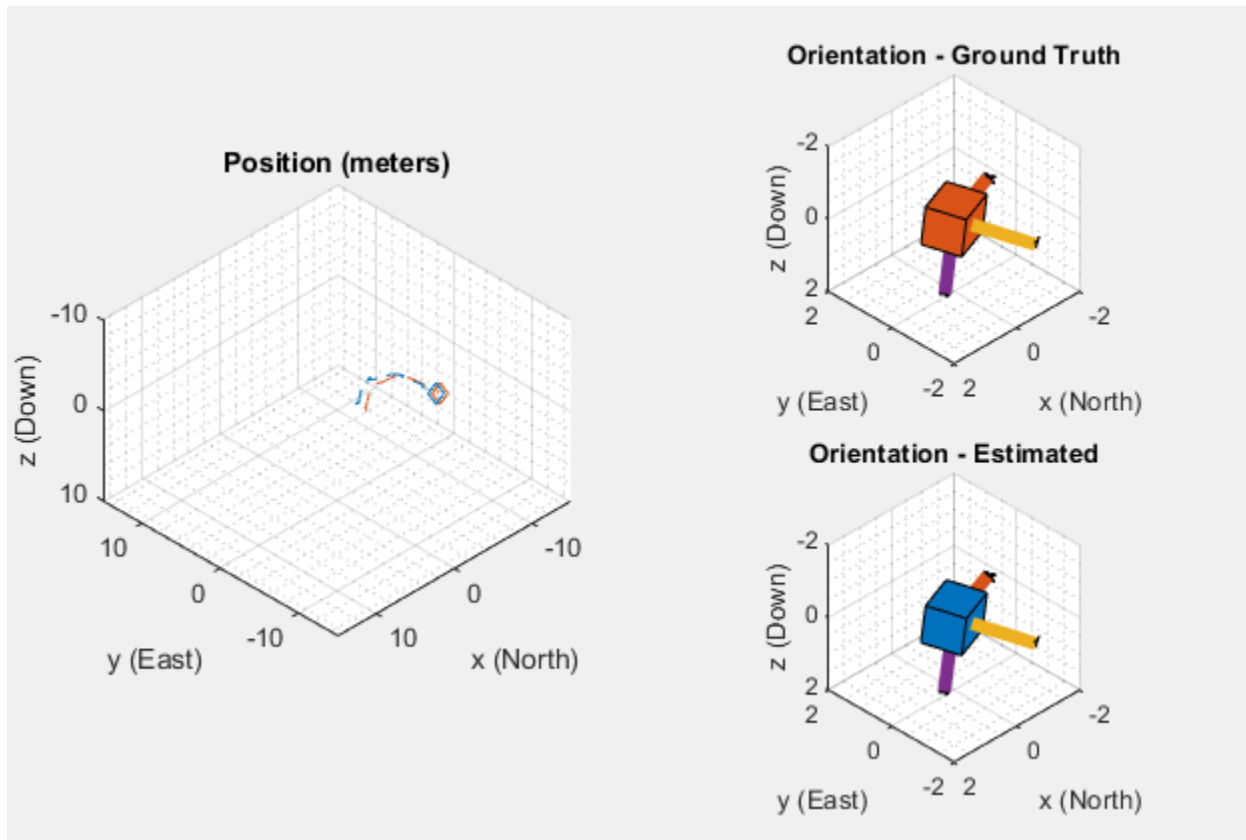
Error Metric Computation

Position and orientation estimates were logged throughout the simulation. Now compute an end-to-end root mean squared error for both position and orientation.

```
posd = pqpos(1:loopBound,:) - trajPos( 1:loopBound, :);

% For orientation, quaternion distance is a much better alternative to
% subtracting Euler angles, which have discontinuities. The quaternion
% distance can be computed with the |dist| function, which gives the
% angular difference in orientation in radians. Convert to degrees
% for display in the command window.

quatd = rad2deg(dist(pqorient(1:loopBound), trajOrient(1:loopBound)) );

% Display RMS errors in the command window.
fprintf('\n\nEnd-to-End Simulation Position RMS Error\n');
```

End-to-End Simulation Position RMS Error

```
msep = sqrt(mean(posd.^2));
fprintf('\tX: %.2f , Y: %.2f, Z: %.2f    (meters)\n\n',msep(1), ...
    msep(2), msep(3));
```

X: 0.57 , Y: 0.53, Z: 0.68   (meters)

```
fprintf('End-to-End Quaternion Distance RMS Error (degrees) \n');
```

End-to-End Quaternion Distance RMS Error (degrees)

```
fprintf('\t%.2f (degrees)\n\n', sqrt(mean(quatd.^2)));
```

    0.32 (degrees)

Estimate Position and Orientation of a Ground Vehicle

This example shows how to estimate the position and orientation of ground vehicles by fusing data from an inertial measurement unit (IMU) and a global positioning system (GPS) receiver.

Simulation Setup

Set the sampling rates. In a typical system, the accelerometer and gyroscope in the IMU run at relatively high sample rates. The complexity of processing data from those sensors in the fusion algorithm is relatively low. Conversely, the GPS runs at a relatively low sample rate and the complexity associated with processing it is high. In this fusion algorithm the GPS samples are processed at a low rate, and the accelerometer and gyroscope samples are processed together at the same high rate.

To simulate this configuration, the IMU (accelerometer and gyroscope) is sampled at 100 Hz, and the GPS is sampled at 10 Hz.

```
imuFs = 100;
gpsFs = 10;

% Define where on the Earth this simulation takes place using latitude,
% longitude, and altitude (LLA) coordinates.
localOrigin = [42.2825 -71.343 53.0352];

% Validate that the |gpsFs| divides |imuFs|. This allows the sensor sample
% rates to be simulated using a nested for loop without complex sample rate
% matching.

imuSamplesPerGPS = (imuFs/gpsFs);
assert(imuSamplesPerGPS == fix(imuSamplesPerGPS), ...
    'GPS sampling rate must be an integer factor of IMU sampling rate.');
```

Fusion Filter

Create the filter to fuse IMU + GPS measurements. The fusion filter uses an extended Kalman filter to track orientation (as a quaternion), position, velocity, and sensor biases.

The `insfilterNonholonomic` object that has two main methods: `predict` and `fusegps`. The `predict` method takes the accelerometer and gyroscope samples from the IMU as input. Call the `predict` method each time the accelerometer and gyroscope are sampled. This method predicts the states forward one time step based on the accelerometer and gyroscope. The error covariance of the extended Kalman filter is updated in this step.

The `fusegps` method takes the GPS samples as input. This method updates the filter states based on the GPS sample by computing a Kalman gain that weights the various sensor inputs according to their uncertainty. An error covariance is also updated in this step, this time using the Kalman gain as well.

The `insfilterNonholonomic` object has two main properties: `IMUSampleRate` and `DecimationFactor`. The ground vehicle has two velocity constraints that assume it does not bounce off the ground or slide on the ground. These constraints are applied using the extended Kalman filter update equations. These updates are applied to the filter states at a rate of `IMUSampleRate`/ `DecimationFactor` Hz.

```
gndFusion = insfilterNonholonomic('ReferenceFrame', 'ENU', ...
    'IMUSampleRate', imuFs, ...
```

```
    'ReferenceLocation', localOrigin, ...
    'DecimationFactor', 2);
```

Create Ground Vehicle Trajectory

The `waypointTrajectory` object calculates pose based on specified sampling rate, waypoints, times of arrival, and orientation. Specify the parameters of a circular trajectory for the ground vehicle.

```
% Trajectory parameters
r = 8.42; % (m)
speed = 2.50; % (m/s)
center = [0, 0]; % (m)
initialYaw = 90; % (degrees)
numRevs = 2;

% Define angles theta and corresponding times of arrival t.
revTime = 2*pi*r / speed;
theta = (0:pi/2:2*pi*numRevs).';
t = linspace(0, revTime*numRevs, numel(theta)).';

% Define position.
x = r .* cos(theta) + center(1);
y = r .* sin(theta) + center(2);
z = zeros(size(x));
position = [x, y, z];

% Define orientation.
yaw = theta + deg2rad(initialYaw);
yaw = mod(yaw, 2*pi);
pitch = zeros(size(yaw));
roll = zeros(size(yaw));
orientation = quaternion([yaw, pitch, roll], 'euler', ...
    'ZYX', 'frame');

% Generate trajectory.
groundTruth = waypointTrajectory('SampleRate', imuFs, ...
    'Waypoints', position, ...
    'TimeOfArrival', t, ...
    'Orientation', orientation);

% Initialize the random number generator used to simulate sensor noise.
rng('default');
```

GPS Receiver

Set up the GPS at the specified sample rate and reference location. The other parameters control the nature of the noise in the output signal.

```
gps = gpsSensor('UpdateRate', gpsFs, 'ReferenceFrame', 'ENU');
gps.ReferenceLocation = localOrigin;
gps.DecayFactor = 0.5;                    % Random walk noise parameter
gps.HorizontalPositionAccuracy = 1.0;
gps.VerticalPositionAccuracy =  1.0;
gps.VelocityAccuracy = 0.1;
```

IMU Sensors

Typically, ground vehicles use a 6-axis IMU sensor for pose estimation. To model an IMU sensor, define an IMU sensor model containing an accelerometer and gyroscope. In a real-world application,

the two sensors could come from a single integrated circuit or separate ones. The property values set here are typical for low-cost MEMS sensors.

```
imu = imuSensor('accel-gyro', ...
    'ReferenceFrame', 'ENU', 'SampleRate', imuFs);

% Accelerometer
imu.Accelerometer.MeasurementRange =  19.6133;
imu.Accelerometer.Resolution = 0.0023928;
imu.Accelerometer.NoiseDensity = 0.0012356;

% Gyroscope
imu.Gyroscope.MeasurementRange = deg2rad(250);
imu.Gyroscope.Resolution = deg2rad(0.0625);
imu.Gyroscope.NoiseDensity = deg2rad(0.025);
```

Initialize the States of the `insfilterNonholonomic`

The states are:

```
States                            Units    Index
Orientation (quaternion parts)             1:4
Gyroscope Bias (XYZ)              rad/s    5:7
Position (NED)                    m        8:10
Velocity (NED)                    m/s      11:13
Accelerometer Bias (XYZ)         m/s^2    14:16
```

Ground truth is used to help initialize the filter states, so the filter converges to good answers quickly.

```
% Get the initial ground truth pose from the first sample of the trajectory
% and release the ground truth trajectory to ensure the first sample is not
% skipped during simulation.
[initialPos, initialAtt, initialVel] = groundTruth();
reset(groundTruth);

% Initialize the states of the filter
gndFusion.State(1:4) = compact(initialAtt).';
gndFusion.State(5:7) = imu.Gyroscope.ConstantBias;
gndFusion.State(8:10) = initialPos.';
gndFusion.State(11:13) = initialVel.';
gndFusion.State(14:16) = imu.Accelerometer.ConstantBias;
```

Initialize the Variances of the `insfilterNonholonomic`

The measurement noises describe how much noise is corrupting the GPS reading based on the `gpsSensor` parameters and how much uncertainty is in the vehicle dynamic model.

The process noises describe how well the filter equations describe the state evolution. Process noises are determined empirically using parameter sweeping to jointly optimize position and orientation estimates from the filter.

```
% Measurement noises
Rvel = gps.VelocityAccuracy.^2;
Rpos = gps.HorizontalPositionAccuracy.^2;

% The dynamic model of the ground vehicle for this filter assumes there is
% no side slip or skid during movement. This means that the velocity is
% constrained to only the forward body axis. The other two velocity axis
```

```
% readings are corrected with a zero measurement weighted by the
% |ZeroVelocityConstraintNoise| parameter.
gndFusion.ZeroVelocityConstraintNoise = 1e-2;

% Process noises
gndFusion.GyroscopeNoise = 4e-6;
gndFusion.GyroscopeBiasNoise = 4e-14;
gndFusion.AccelerometerNoise = 4.8e-2;
gndFusion.AccelerometerBiasNoise = 4e-14;

% Initial error covariance
gndFusion.StateCovariance = 1e-9*ones(16);
```

Initialize Scopes

The `HelperScrollingPlotter` scope enables plotting of variables over time. It is used here to track errors in pose. The `HelperPoseViewer` scope allows 3-D visualization of the filter estimate and ground truth pose. The scopes can slow the simulation. To disable a scope, set the corresponding logical variable to `false`.

```
useErrScope = true; % Turn on the streaming error plot
usePoseView = true;  % Turn on the 3D pose viewer

if useErrScope
    errscope = HelperScrollingPlotter( ...
            'NumInputs', 4, ...
            'TimeSpan', 10, ...
            'SampleRate', imuFs, ...
            'YLabel', {'degrees', ...
            'meters', ...
            'meters', ...
            'meters'}, ...
            'Title', {'Quaternion Distance', ...
            'Position X Error', ...
            'Position Y Error', ...
            'Position Z Error'}, ...
            'YLimits', ...
            [-1, 1
             -1, 1
             -1, 1
             -1, 1]);
end

if usePoseView
    viewer = HelperPoseViewer( ...
        'XPositionLimits', [-15, 15], ...
        'YPositionLimits', [-15, 15], ...
        'ZPositionLimits', [-5, 5], ...
        'ReferenceFrame', 'ENU');
end
```

Simulation Loop

The main simulation loop is a while loop with a nested for loop. The while loop executes at the `gpsFs`, which is the GPS measurement rate. The nested for loop executes at the `imuFs`, which is the IMU sample rate. The scopes are updated at the IMU sample rate.

```matlab
totalSimTime = 30; % seconds

% Log data for final metric computation.
numsamples = floor(min(t(end), totalSimTime) * gpsFs);
truePosition = zeros(numsamples,3);
trueOrientation = quaternion.zeros(numsamples,1);
estPosition = zeros(numsamples,3);
estOrientation = quaternion.zeros(numsamples,1);

idx = 0;

for sampleIdx = 1:numsamples
    % Predict loop at IMU update frequency.
    for i = 1:imuSamplesPerGPS
        if ~isDone(groundTruth)
            idx = idx + 1;

            % Simulate the IMU data from the current pose.
            [truePosition(idx,:), trueOrientation(idx,:), ...
                trueVel, trueAcc, trueAngVel] = groundTruth();
            [accelData, gyroData] = imu(trueAcc, trueAngVel, ...
                trueOrientation(idx,:));

            % Use the predict method to estimate the filter state based
            % on the accelData and gyroData arrays.
            predict(gndFusion, accelData, gyroData);

            % Log the estimated orientation and position.
            [estPosition(idx,:), estOrientation(idx,:)] = pose(gndFusion);

            % Compute the errors and plot.
            if useErrScope
                orientErr = rad2deg( ...
                    dist(estOrientation(idx,:), trueOrientation(idx,:)));
                posErr = estPosition(idx,:) - truePosition(idx,:);
                errscope(orientErr, posErr(1), posErr(2), posErr(3));
            end

            % Update the pose viewer.
            if usePoseView
                viewer(estPosition(idx,:), estOrientation(idx,:), ...
                    truePosition(idx,:), estOrientation(idx,:));
            end
        end
    end

    if ~isDone(groundTruth)
        % This next step happens at the GPS sample rate.
        % Simulate the GPS output based on the current pose.
        [lla, gpsVel] = gps(truePosition(idx,:), trueVel);

        % Update the filter states based on the GPS data.
        fusegps(gndFusion, lla, Rpos, gpsVel, Rvel);
    end
end
```

Quaternion Distance

Position X Error

Position Y Error

Position Z Error

Position (meters)

Orientation - Ground Truth

Orientation - Estimated
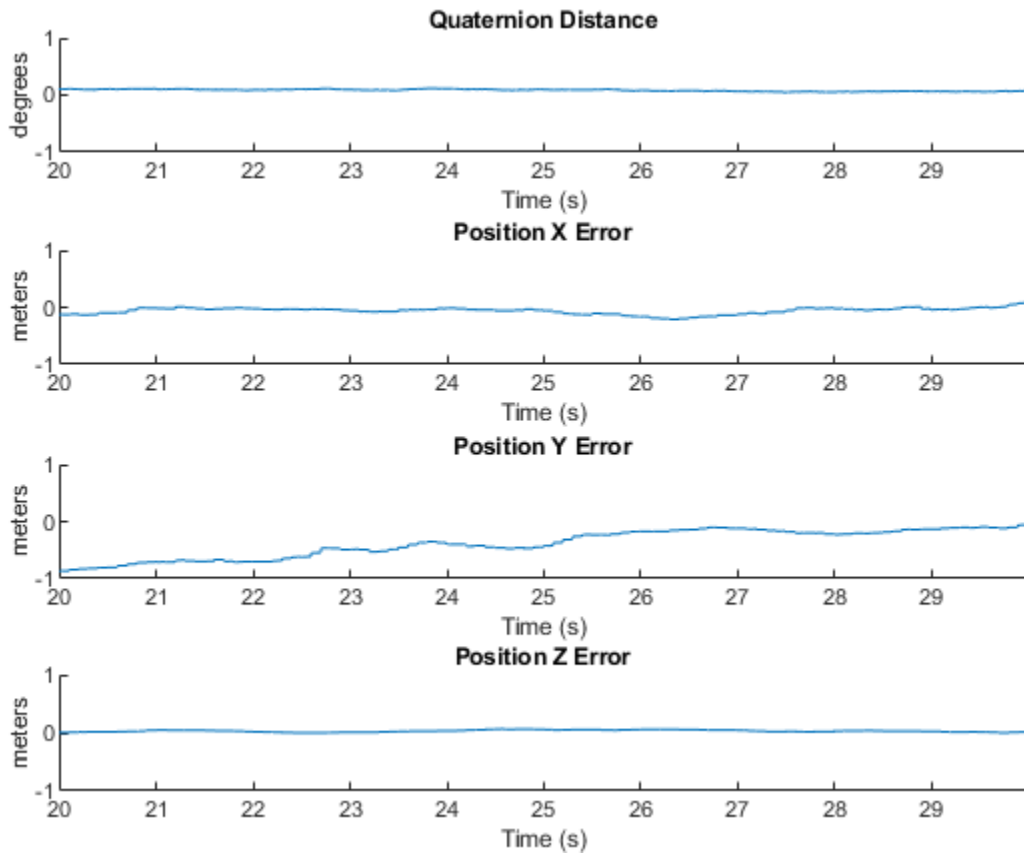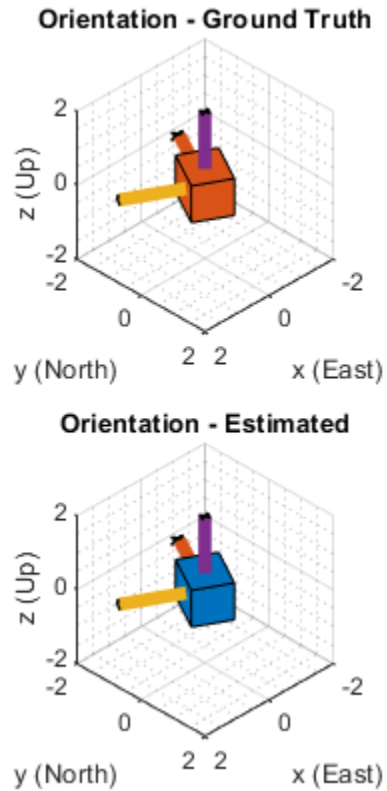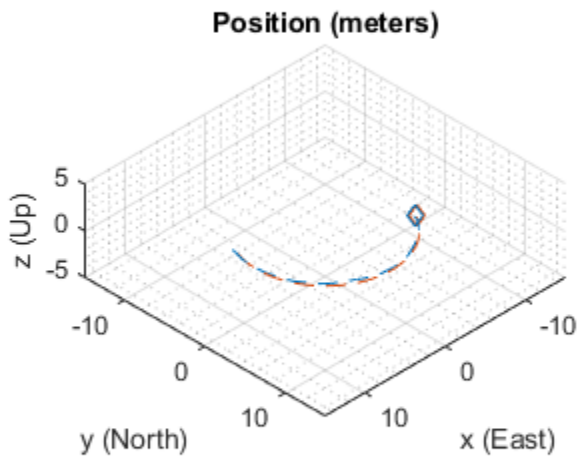
Error Metric Computation

Position and orientation were logged throughout the simulation. Now compute an end-to-end root mean squared error for both position and orientation.

```
posd = estPosition - truePosition;

% For orientation, quaternion distance is a much better alternative to
% subtracting Euler angles, which have discontinuities. The quaternion
% distance can be computed with the |dist| function, which gives the
% angular difference in orientation in radians. Convert to degrees for
% display in the command window.

quatd = rad2deg(dist(estOrientation, trueOrientation));

% Display RMS errors in the command window.
fprintf('\n\nEnd-to-End Simulation Position RMS Error\n');
msep = sqrt(mean(posd.^2));
fprintf('\tX: %.2f , Y: %.2f, Z: %.2f   (meters)\n\n', msep(1), ...
    msep(2), msep(3));

fprintf('End-to-End Quaternion Distance RMS Error (degrees) \n');
fprintf('\t%.2f (degrees)\n\n', sqrt(mean(quatd.^2)));
```

```
End-to-End Simulation Position RMS Error
```

```
    X: 1.16 , Y: 0.99, Z: 0.03   (meters)

End-to-End Quaternion Distance RMS Error (degrees)
    0.09 (degrees)
```

Rotations, Orientation, and Quaternions

This example reviews concepts in three-dimensional rotations and how quaternions are used to describe orientation and rotations. Quaternions are a skew field of hypercomplex numbers. They have found applications in aerospace, computer graphics, and virtual reality. In MATLAB®, quaternion mathematics can be represented by manipulating the `quaternion` class.

The `HelperDrawRotation` class is used to illustrate several portions of this example.

```
dr = HelperDrawRotation;
```

Rotations in Three Dimensions

All rotations in 3-D can be defined by an axis of rotation and an angle of rotation about that axis. Consider the 3-D image of a teapot in the leftmost plot. The teapot is rotated by 45 degrees around the Z-axis in the second plot. A more complex rotation of 15 degrees around the axis [1 0 1] is shown in the third plot. Quaternions encapsulate the axis and angle of rotation and have an algebra for manipulating these rotations. The `quaternion` class, and this example, use the "right-hand rule" convention to define rotations. That is, positive rotations are clockwise around the axis of rotation when viewed from the origin.

```
dr.drawTeapotRotations;
```
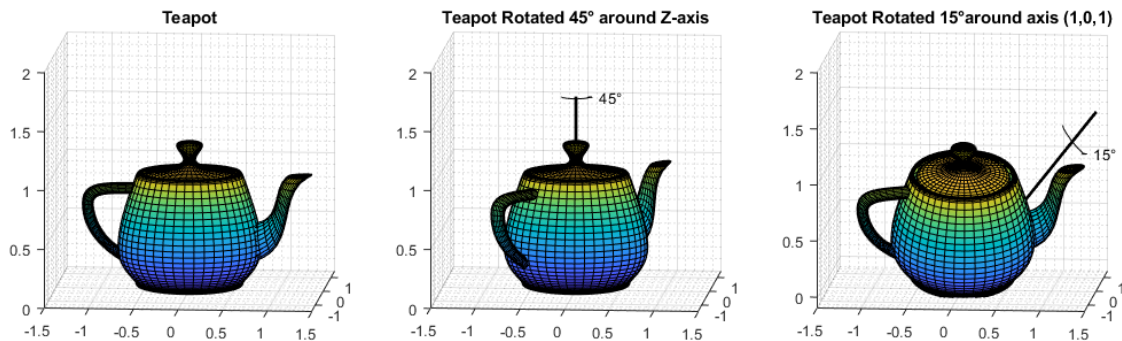

Point Rotation

The vertices of the teapot were rotated about the axis of rotation in the reference frame. Consider a point (0.7, 0.5) rotated 30 degrees about the Z-axis.

```
figure;
dr.draw2DPointRotation(gca);
```
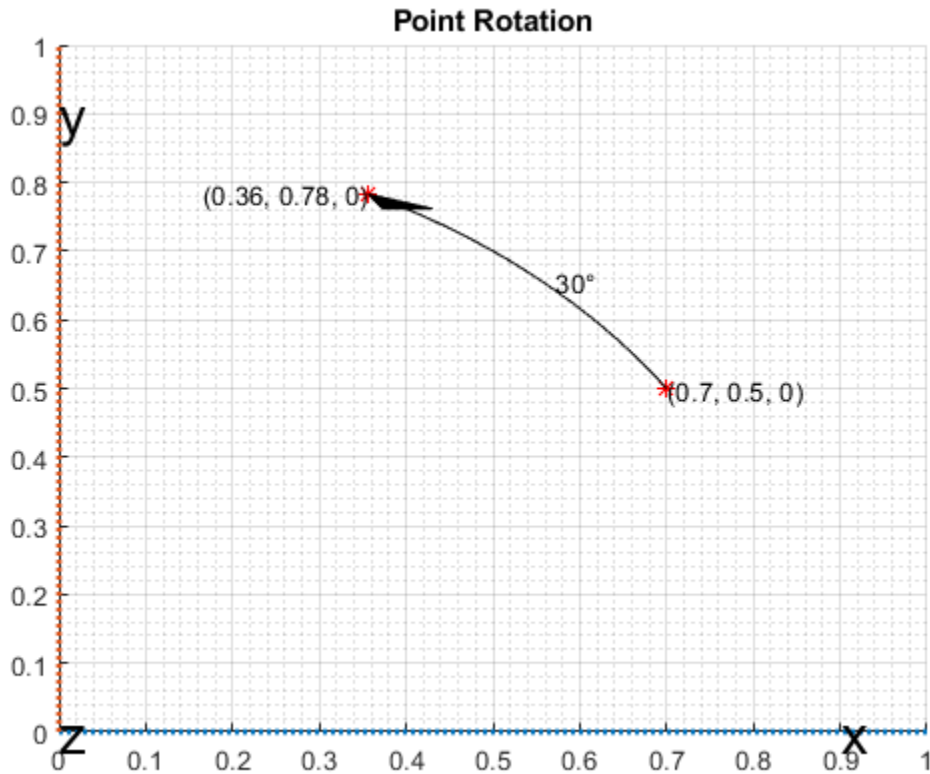
Frame Rotation

Frame rotation is, in some sense, the opposite of point rotation. In frame rotation, the points of the object stay fixed, but the frame of reference is rotated. Again, consider the point (0.7, 0.5). Now the reference frame is rotated by 30 degrees around the Z-axis. Note that while the point (0.7, 0.5) stays fixed, it has different coordinates in the new, rotated frame of reference.

```
figure;
dr.draw2DFrameRotation(gca);
```
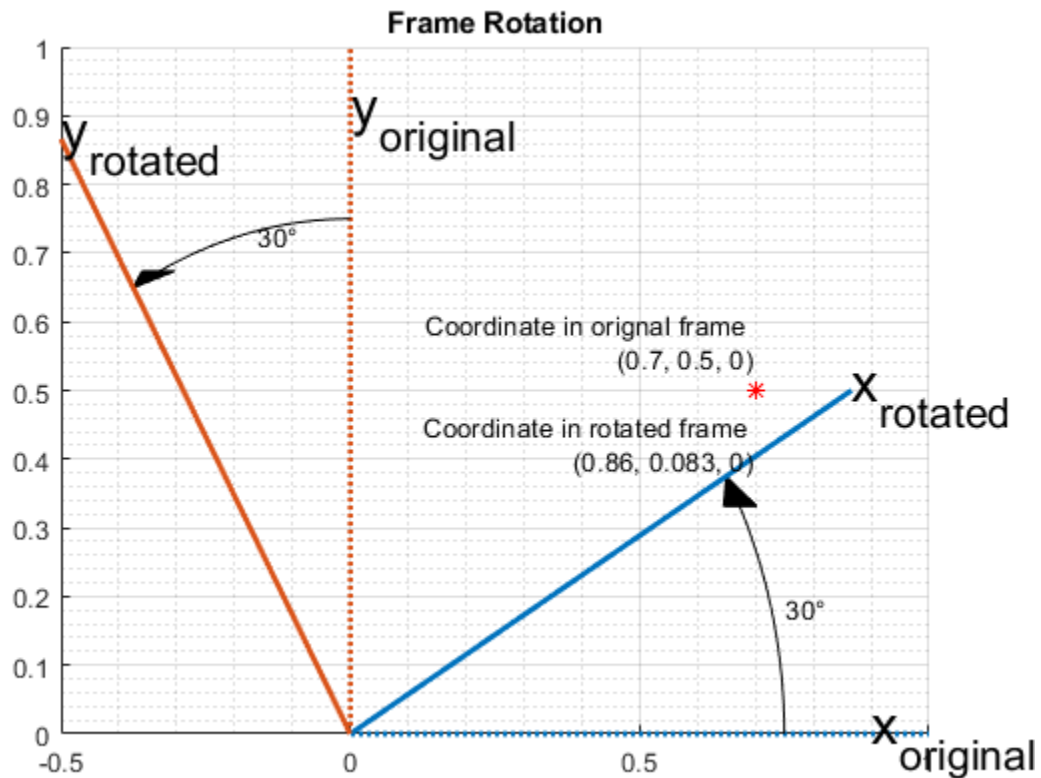
Frame Rotation

Orientation

Orientation refers to the angular displacement of an object relative to a frame of reference. Typically, orientation is described by the rotation that causes this angular displacement from a starting orientation. In this example, orientation is defined as the rotation that takes a quantity in a parent reference frame to a child reference frame. Orientation is usually given as a quaternion, rotation matrix, set of Euler angles, or rotation vector. It is useful to think about orientation as a frame rotation: the child reference frame is rotated relative to the parent frame.

Consider an example where the child reference frame is rotated 30 degrees around the vector [1/3 2/3 2/3].

```
figure;
dr.draw3DOrientation(gca, [1/3 2/3 2/3], 30);
```

Frame Rotation of 30° Around the Vector [1/3 2/3 2/3]
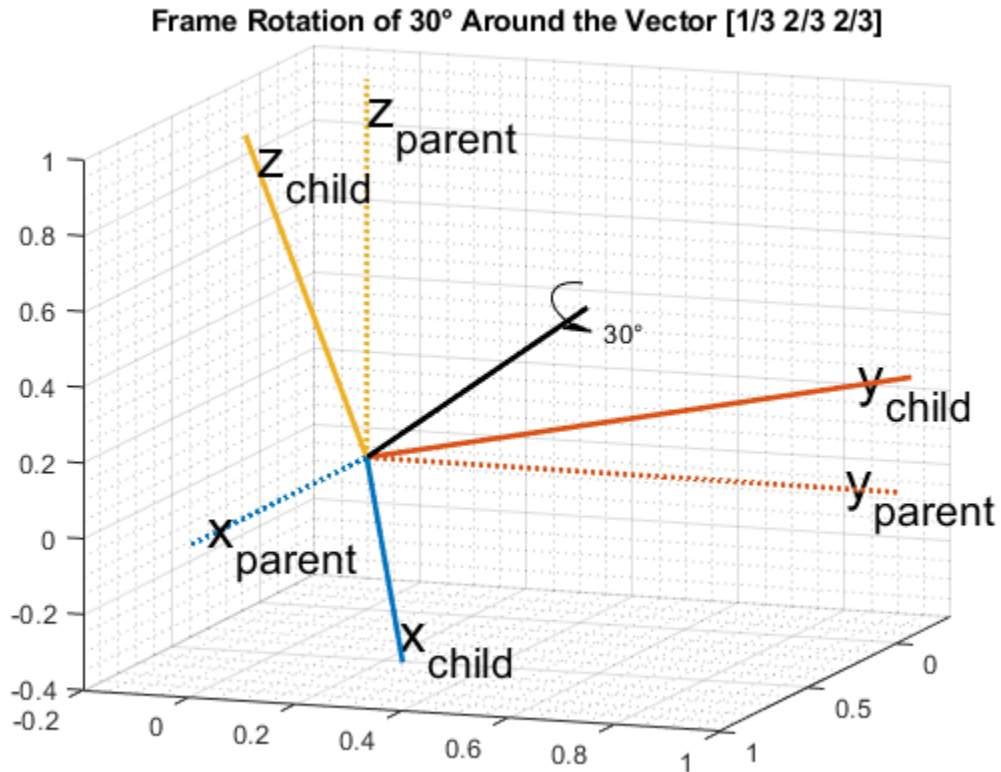
Quaternions

Quaternions are numbers of the form

$$a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$$

where

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

and $a, b, c,$ and d are real numbers. In the rest of this example, the four numbers $a, b, c,$ and d are referred to as the *parts* of the quaternion.

Quaternions for Rotations and Orientation

The axis and the angle of rotation are encapsulated in the quaternion parts. For a unit vector axis of rotation [x, y, z], and rotation angle α, the quaternion describing this rotation is

$$\cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)(x\mathbf{i} + y\mathbf{j} + z\mathbf{k})$$

Note that to describe a rotation using a quaternion, the quaternion must be a *unit quaternion*. A unit quaternion has a norm of 1, where the norm is defined as

$$norm(q) = \sqrt{a^2 + b^2 + c^2 + d^2}$$

5-35

There are a variety of ways to construct a quaternion in MATLAB, for example:

```
q1 = quaternion(1,2,3,4)
```

```
q1 =

  quaternion


    1 + 2i + 3j + 4k
```

Arrays of quaternions can be made in the same way:

```
quaternion([1 10; -1 1], [2 20; -2 2], [3 30; -3 3], [4 40; -4 4])
```

```
ans =

  2x2 quaternion array


     1 +  2i +  3j +  4k     10 + 20i + 30j + 40k
    -1 -  2i -  3j -  4k      1 +  2i +  3j +  4k
```

Arrays with four columns can also be used to construct quaternions, with each column representing a quaternion part:

```
qmgk = quaternion(magic(4))
```

```
qmgk =

  4x1 quaternion array


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

Quaternions can be indexed and manipulated just like any other array:

```
qmgk(3)
```

```
ans =

  quaternion


     9 +  7i +  6j + 12k
```

```
reshape(qmgk,2,2)
```

```
ans =

  2x2 quaternion array


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

```
[q1; q1]
```

```
ans =

  2x1 quaternion array


     1 + 2i + 3j + 4k
     1 + 2i + 3j + 4k
```

Quaternion Math

Quaternions have well-defined arithmetic operations. Addition and subtraction are similar to complex numbers: parts are added/subtracted independently. Multiplication is more complicated because of the earlier equation:

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

This means that multiplication of quaternions is not commutative. That is, $pq \neq qp$ for quaternions p and q. However, every quaternion has a multiplicative inverse, so quaternions can be divided. Arrays of the `quaternion` class can be added, subtracted, multiplied, and divided in MATLAB.

```
q = quaternion(1,2,3,4);
p = quaternion(-5,6,-7,8);
```

Addition

```
p + q
```

```
ans =

  quaternion


    -4 +  8i -  4j + 12k
```

Subtraction

```
p - q
```

```
ans =

  quaternion
```

```
    -6 +   4i -  10j +   4k
```

Multiplication

p*q

```
ans =

  quaternion

   -28 -  56i -  30j +  20k
```

Multiplication in the reverse order (note the different result)

q*p

```
ans =

  quaternion

   -28 +  48i -  14j -  44k
```

Right division of p by q is equivalent to $p(q^{-1})$.

p./q

```
ans =

  quaternion

      0.6 +   2.2667i + 0.53333j - 0.13333k
```

Left division of q by p is equivalent to $p^{-1}q$.

p.\q

```
ans =

  quaternion

    0.10345 +   0.2069i +      0j - 0.34483k
```

The conjugate of a quaternion is formed by negating each of the non-real parts, similar to conjugation for a complex number:

```
conj(p)

ans =

  quaternion

   -5 - 6i + 7j - 8k
```

Quaternions can be normalized in MATLAB:

```
pnormed = normalize(p)

pnormed =

  quaternion

   -0.37905 + 0.45486i - 0.53067j + 0.60648k

norm(pnormed)

ans =

    1
```

Point and Frame Rotations with Quaternions

Quaternions can be used to rotate points in a static frame of reference, or to rotate the frame of reference itself. The `rotatepoint` function rotates a point $v = (v_x, v_y, v_z)$ using a quaternion q through the following equation:

$$pv_{quat}p^*$$

where v_{quat} is

$$v_{quat} = 0 + v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k}$$

and p^* indicates quaternion conjugation. Note the above quaternion multiplication results in a quaternion with the real part, a, equal to 0. The b, c, and d parts of the result form the rotated point (b, c, d).

Consider the example of point rotation from above. The point (0.7, 0.5) was rotated 30 degrees around the Z-axis. In three dimensions this point has a 0 Z-coordinate. Using the axis-angle formulation, a quaternion can be constructed using [0 0 1] as the axis of rotation.

```
ang = deg2rad(30);
q = quaternion(cos(ang/2), 0, 0, sin(ang/2));
pt = [0.7, 0.5, 0];  % Z-coordinate is 0 in the X-Y plane
ptrot = rotatepoint(q, pt)
```

```
ptrot =

    0.3562    0.7830          0
```

Similarly, the `rotateframe` function takes a quaternion q and point v to compute

$$p^* v_{quat} p$$

Again the above quaternion multiplication results in a quaternion with 0 real part. The (b, c, d) parts of the result form the coordinate of the point v in the new, rotated reference frame. Using the `quaternion` class:

```
ptframerot = rotateframe(q, pt)
```

```
ptframerot =

    0.8562    0.0830          0
```

A quaternion and its conjugate have opposite effects because of the symmetry in the point and frame rotation equations. Rotating by the conjugate "undoes" the rotation.

```
rotateframe(conj(q), ptframerot)
```

```
ans =

    0.7000    0.5000          0
```

Because of the symmetry of the equations, this code performs the same rotation.

```
rotatepoint(q, ptframerot)
```

```
ans =

    0.7000    0.5000          0
```

Other Rotation Representations

Often rotations and orientations are described using alternate means: Euler angles, rotation matrices, and/or rotation vectors. All of these interoperate with quaternions in MATLAB.

Euler angles are frequently used because they are easy to interpret. Consider a frame of reference rotated by 30 degrees around the Z-axis, then 20 degrees around the Y-axis, and then -50 degrees around the X-axis. Note here, and throughout, the rotations around each axis are *intrinsic*: each subsequent rotation is around the newly created set of axes. In other words, the second rotation is around the "new" Y-axis created by the first rotation, not around the original Y-axis.

```
figure;
euld = [30 20 -50];
dr.drawEulerRotation(gca, euld);
```

Frame Rotation using Euler Angles of 30°Around Z-axis, 20°Around Y-axis, -50°Around X-axis
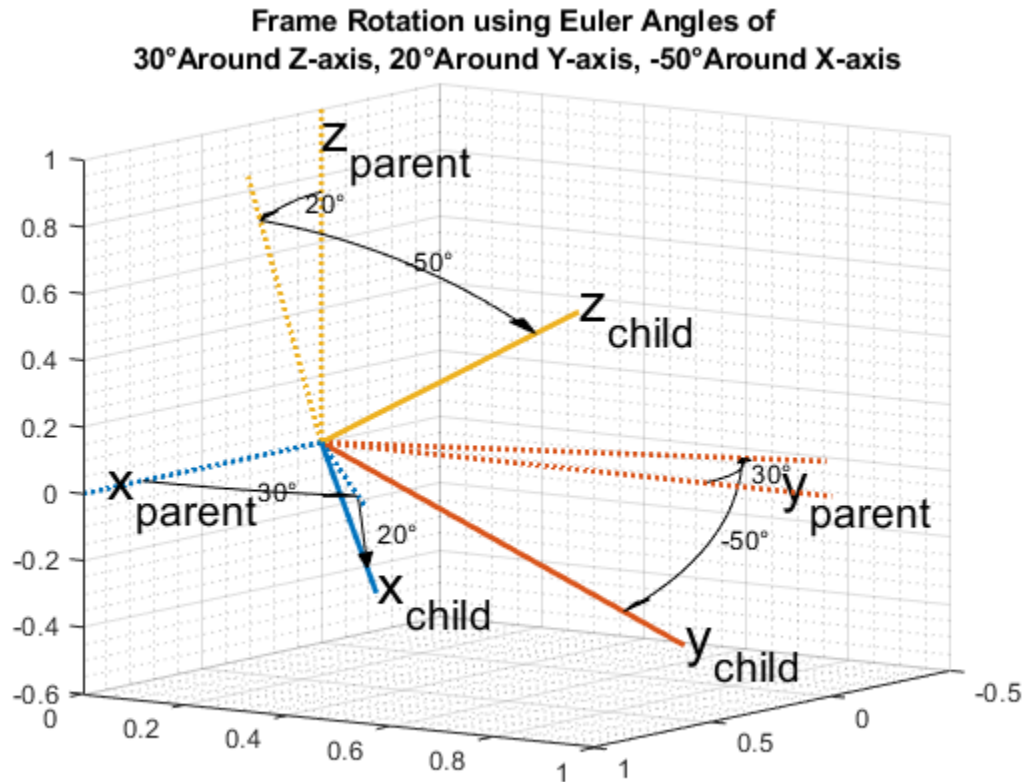
To build a quaternion from these Euler angles for the purpose of frame rotation, use the `quaternion` constructor. Since the order of rotations is around the Z-axis first, then around the new Y-axis, and finally around the new X-axis, use the `'ZYX'` flag.

```
qeul = quaternion(deg2rad(euld), 'euler', 'ZYX', 'frame')

qeul =

  quaternion


      0.84313 -  0.44275i + 0.044296j +  0.30189k
```

The `'euler'` flag indicates that the first argument is in radians. If the argument is in degrees, use the `'eulerd'` flag.

```
qeuld = quaternion(euld, 'eulerd', 'ZYX', 'frame')

qeuld =

  quaternion


      0.84313 -  0.44275i + 0.044296j +  0.30189k
```

To convert back to Euler angles:

```
rad2deg(euler(qeul, 'ZYX', 'frame'))
```

```
ans =

   30.0000   20.0000  -50.0000
```

Equivalently, the `eulerd` method can be used.

```
eulerd(qeul, 'ZYX', 'frame')
```

```
ans =

   30.0000   20.0000  -50.0000
```

Alternatively, this same rotation can be represented as a rotation matrix:

```
rmat = rotmat(qeul, 'frame')
```

```
rmat =

    0.8138    0.4698   -0.3420
   -0.5483    0.4257   -0.7198
   -0.1926    0.7733    0.6040
```

The conversion back to quaternions is similar:

```
quaternion(rmat, 'rotmat', 'frame')
```

```
ans =

  quaternion

      0.84313 -  0.44275i + 0.044296j +  0.30189k
```

Just as a quaternion can be used for either point or frame rotation, it can be converted to a rotation matrix (or set of Euler angles) specifically for point or frame rotation. The rotation matrix for point rotation is the transpose of the matrix for frame rotation. To convert between rotation representations, it is necessary to specify `'point'` or `'frame'`.

The rotation matrix for the point rotation section of this example is:

```
rotmatPoint = rotmat(q, 'point')
```

```
rotmatPoint =

    0.8660   -0.5000         0
    0.5000    0.8660         0
```

```
        0         0    1.0000
```

To find the location of the rotated point, right-multiply `rotmatPoint` by the transposed array `pt`.

```
rotmatPoint * (pt')
```

```
ans =

    0.3562
    0.7830
         0
```

The rotation matrix for the frame rotation section of this example is:

```
rotmatFrame = rotmat(q, 'frame')
```

```
rotmatFrame =

    0.8660    0.5000         0
   -0.5000    0.8660         0
         0         0    1.0000
```

To find the location of the point in the rotated reference frame, right-multiply `rotmatFrame` by the transposed array `pt`.

```
rotmatFrame * (pt')
```

```
ans =

    0.8562
    0.0830
         0
```

A rotation vector is an alternate, compact rotation encapsulation. A rotation vector is simply a three-element vector that represents the unit length axis of rotation scaled-up by the angle of rotation in radians. There is no frame-ness or point-ness associated with a rotation vector. To convert to a rotation vector:

```
rv = rotvec(qeul)
```

```
rv =

   -0.9349    0.0935    0.6375
```

To convert to a quaternion:

```
quaternion(rv, 'rotvec')
```

```
ans =
```

```
quaternion


     0.84313 -  0.44275i + 0.044296j +  0.30189k
```

Distance

One advantage of quaternions over Euler angles is the lack of discontinuities. Euler angles have discontinuities that vary depending on the convention being used. The `dist` function compares the effect of rotation by two different quaternions. The result is a number in the range of 0 to `pi`. Consider two quaternions constructed from Euler angles:

```
eul1 = [0, 10, 0];
eul2 = [0, 15, 0];
qdist1 = quaternion(deg2rad(eul1), 'euler', 'ZYX', 'frame');
qdist2 = quaternion(deg2rad(eul2), 'euler', 'ZYX', 'frame');
```

Subtracting the Euler angles, you can see there is no rotation around the Z-axis or X-axis.

```
eul2 - eul1
```

```
ans =

     0     5     0
```

The difference between these two rotations is five degrees around the Y-axis. The `dist` shows the difference as well.

```
rad2deg(dist(qdist1, qdist2))
```

```
ans =

    5.0000
```

For Euler angles such as `eul1` and `eul2`, computing angular distance is trivial. A more complex example, which spans an Euler angle discontinuity, is:

```
eul3 = [0, 89, 0];
eul4 = [180, 89, 180];
qdist3 = quaternion(deg2rad(eul3), 'euler', 'ZYX', 'frame');
qdist4 = quaternion(deg2rad(eul4), 'euler', 'ZYX', 'frame');
```

Though `eul3` and `eul4` represent nearly the same orientation, simple Euler angle subtraction gives the impression that these two orientations are very far apart.

```
euldiff = eul4 - eul3
```

```
euldiff =

   180     0   180
```

Using the `dist` function on the quaternions shows that there is only a two-degree difference in these rotations:

```
euldist = rad2deg(dist(qdist3, qdist4))

euldist =

    2.0000
```

A quaternion and its negative represent the same rotation. This is not obvious from subtracting quaternions, but the `dist` function makes it clear.

```
qpos = quaternion(-cos(pi/4), 0 ,0, sin(pi/4))

qpos =

  quaternion

    -0.70711 +       0i +       0j + 0.70711k

qneg = -qpos

qneg =

  quaternion

     0.70711 +       0i +       0j - 0.70711k

qdiff = qpos - qneg

qdiff =

  quaternion

    -1.4142 +      0i +      0j + 1.4142k

dist(qpos, qneg)

ans =

     0
```

Supported Functions

The `quaternion` class lets you effectively describe rotations and orientations in MATLAB. The full list of quaternion-supported functions can be found with the `methods` function:

```
methods('quaternion')
```

```
Methods for class quaternion:

angvel            ismatrix          prod
cat               isnan             quaternion
classUnderlying   isrow             rdivide
compact           isscalar          reshape
conj              isvector          rotateframe
ctranspose        ldivide           rotatepoint
disp              length            rotmat
dist              log               rotvec
double            meanrot           rotvecd
eq                minus             single
euler             mtimes            size
eulerd            ndims             slerp
exp               ne                times
horzcat           norm              transpose
iscolumn          normalize         uminus
isempty           numel             validateattributes
isequal           parts             vertcat
isequaln          permute
isfinite          plus
isinf             power

Static methods:

ones              zeros
```

Lowpass Filter Orientation Using Quaternion SLERP

This example shows how to use spherical linear interpolation (SLERP) to create sequences of quaternions and lowpass filter noisy trajectories. SLERP is a commonly used computer graphics technique for creating animations of a rotating object.

SLERP Overview

Consider a pair of quaternions q_0 and q_1. Spherical linear interpolation allows you to create a sequence of quaternions that vary smoothly between q_0 and q_1 with a constant angular velocity. SLERP uses an interpolation parameter `h` that can vary between 0 and 1 and determines how close the output quaternion is to either q_0 or q_1.

The original formulation of quaternion SLERP was given by Ken Shoemake [1] as:

$$Slerp(q_0, q_1, h) = q_1(q_1^{-1}q_2)^h$$

An alternate formulation with sinusoids (used in the `slerp` function implementation) is:

$$Slerp(q_0, q_1, h) = \frac{\sin((1-h)\theta)}{\sin\theta}q_0 + \frac{\sin(h\theta)}{\sin\theta}q_1$$

where θ is the dot product of the quaternion parts. Note that $\theta = dist(q_0, q_1)/2$.

SLERP vs Linear Interpolation of Quaternion Parts

Consider the following example. Build two quaternions from Euler angles.

```
q0 = quaternion([-80 10 0], 'eulerd', 'ZYX', 'frame');
q1 = quaternion([80 70 70], 'eulerd', 'ZYX', 'frame');
```

To find a quaternion 30 percent of the way from `q0` to `q1`, specify the `slerp` parameter as 0.3.

```
p30 = slerp(q0, q1, 0.3);
```

To view the interpolated quaternion's Euler angle representation, use the `eulerd` function.

```
eulerd(p30, 'ZYX', 'frame')
```

```
ans =

  -56.6792   33.2464   -9.6740
```

To create a smooth trajectory between `q0` and `q1`, specify the `slerp` interpolation parameter as a vector of evenly spaced numbers between 0 and 1.

```
dt = 0.01;
h = (0:dt:1).';
trajSlerped = slerp(q0, q1, h);
```

Compare the results of the SLERP algorithm with a trajectory between `q0` and `q1`, using simple linear interpolation (LERP) of each quaternion part.

```
partsLinInterp = interp1( [0;1], compact([q0;q1]), h, 'linear');
```

Note that linear interpolation does not give unit quaternions, so they must be normalized.

```
trajLerped = normalize(quaternion(partsLinInterp));
```

Compute the angular velocities from each approach.

```
avSlerp = helperQuat2AV(trajSlerped, dt);
avLerp = helperQuat2AV(trajLerped, dt);
```

Plot both sets of angular velocities. Notice that the angular velocity for SLERP is constant, but it varies for linear interpolation.

```
sp = HelperSlerpPlotting;
sp.plotAngularVelocities(avSlerp, avLerp);
```
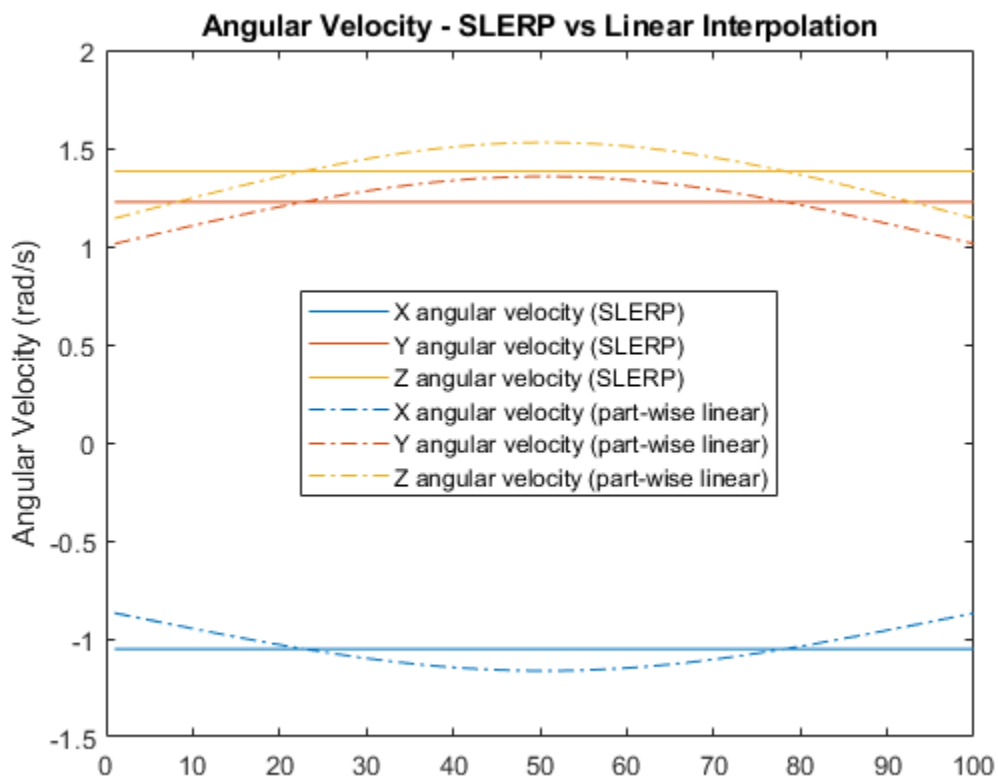

SLERP produces a smooth rotation at a constant rate.

Lowpass Filtering with SLERP

SLERP can also be used to make more complex functions. Here, SLERP is used to lowpass filter a noisy trajectory.

Rotational noise can be constructed by forming a quaternion from a noisy rotation vector.

```
rcurr = rng(1);
sigma = 1e-1;
noiserv = sigma .* ( rand(numel(h), 3) - 0.5);
```

```
qnoise = quaternion(noiserv, 'rotvec');
rng(rcurr);
```

To corrupt the trajectory `trajSlerped` with noise, incrementally rotate the trajectory with the noise vector `qnoise`.

```
trajNoisy = trajSlerped .* qnoise;
```

You can smooth real-valued signals using a single pole filter of the form:

$$y_k = y_{k-1} + \alpha(x_k - y_{k-1})$$

This formula essentially says that the new filter state y_k should be moved toward the current input x_k by a step size that is proportional to the distance between the current input and the current filter state y_{k-1}.

The spirit of this approach informs how a quaternion sequence can be lowpass filtered. To do this, both the `dist` and `slerp` functions are used.

The `dist` function returns a measurement in radians of the difference in rotation applied by two quaternions. The range of the `dist` function is the half-open interval [0, pi).

The `slerp` function is used to steer the filter state towards the current input. It is steered more towards the input when the difference between the input and current filter state has a large `dist`, and less toward the input when `dist` gives a small value. The interpolation parameter to `slerp` is in the closed-interval [0,1], so the output of `dist` must be re-normalized to this range. However, the full range of [0,1] for the interpolation parameter gives poor performance, so it is limited to a smaller range `hrange` centered at `hbias`.

```
hrange = 0.4;
hbias = 0.4;
```

Limit `low` and `high` to the interval [0, 1].

```
low  = max(min(hbias - (hrange./2), 1), 0);
high = max(min(hbias + (hrange./2), 1), 0);
hrangeLimited = high - low;
```

Initialize the filter and preallocate outputs.

```
y = trajNoisy(1); % initial filter state
qout = zeros(size(y), 'like', y); % preallocate filter output
qout(1) = y;
```

Filter the noisy trajectory, sample-by-sample.

```
for ii=2:numel(trajNoisy)
    x = trajNoisy(ii);
    d = dist(y, x);

    % Renormalize dist output to the range [low, high]
    hlpf = (d./pi).*hrangeLimited + low;
    y = slerp(y,x,hlpf);
    qout(ii) = y;
end

f = figure;
```

```
sp.plotEulerd(f, trajNoisy, 'o');
sp.plotEulerd(f, trajSlerped, 'k-.', 'LineWidth', 2);
sp.plotEulerd(f, qout, '-', 'LineWidth', 2);
sp.addAnnotations(f, hrange, hbias);
```
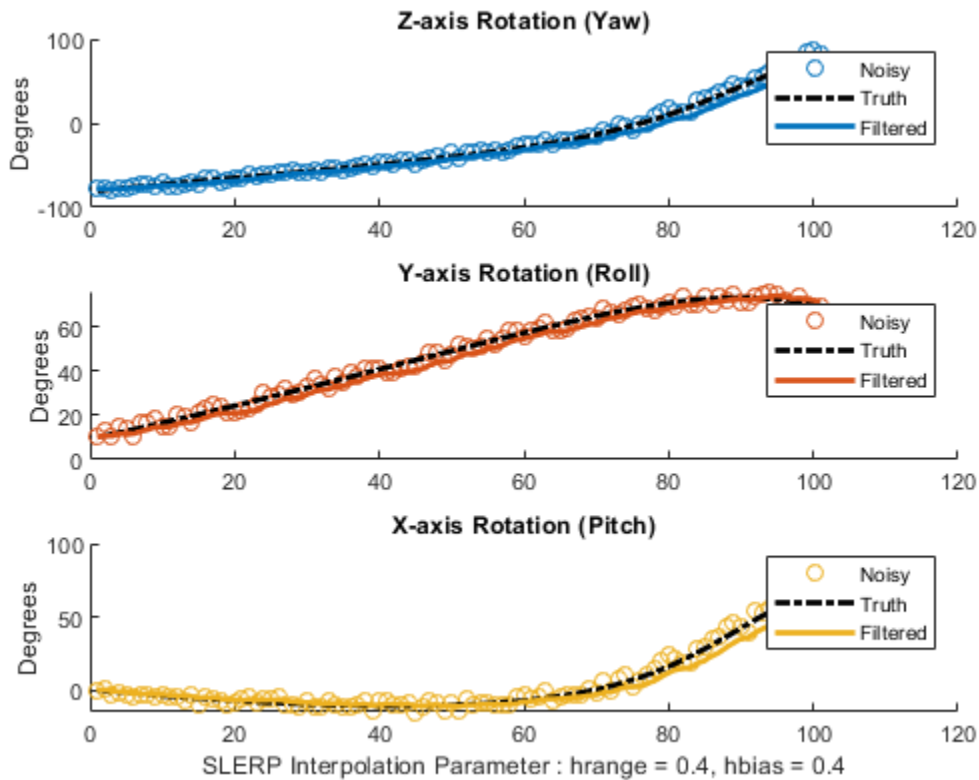

Conclusion

SLERP can be used for creating both short trajectories between two orientations and for smoothing or lowpass filtering. It has found widespread use in a variety of industries.

References

1 Shoemake, Ken. "Animating Rotation with Quaternion Curves." *ACM SIGRAPH Computer Graphics 19*, no 3 (1985):245-54, doi:10.1145/325165.325242

Introduction to Simulating IMU Measurements

This example shows how to simulate inertial measurement unit (IMU) measurements using the `imuSensor` System object. An IMU can include a combination of individual sensors, including a gyroscope, an accelerometer, and a magnetometer. You can specify properties of the individual sensors using `gyroparams`, `accelparams`, and `magparams`, respectively.

In the following plots, unless otherwise noted, only the x-axis measurements are shown.

Default Parameters

The default parameters for the gyroscope model simulate an ideal signal. Given a sinusoidal input, the gyroscope output should match exactly.

```
params = gyroparams

% Generate N samples at a sampling rate of Fs with a sinusoidal frequency
% of Fc.
N = 1000;
Fs = 100;
Fc = 0.25;

t = (0:(1/Fs):((N-1)/Fs)).';
acc = zeros(N, 3);
angvel = zeros(N, 3);
angvel(:,1) = sin(2*pi*Fc*t);

imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Ideal Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')


params =

  gyroparams with properties:

    MeasurementRange: Inf         rad/s
          Resolution: 0           (rad/s)/LSB
        ConstantBias: [0 0 0]     rad/s
    AxesMisalignment: [0 0 0]     %

        NoiseDensity: [0 0 0]     (rad/s)/√Hz
       BiasInstability: [0 0 0]   rad/s
          RandomWalk: [0 0 0]     (rad/s)*√Hz

             TemperatureBias: [0 0 0]    (rad/s)/°C
    TemperatureScaleFactor: [0 0 0]      %/°C
         AccelerationBias: [0 0 0]       (rad/s)/(m/s²)
```
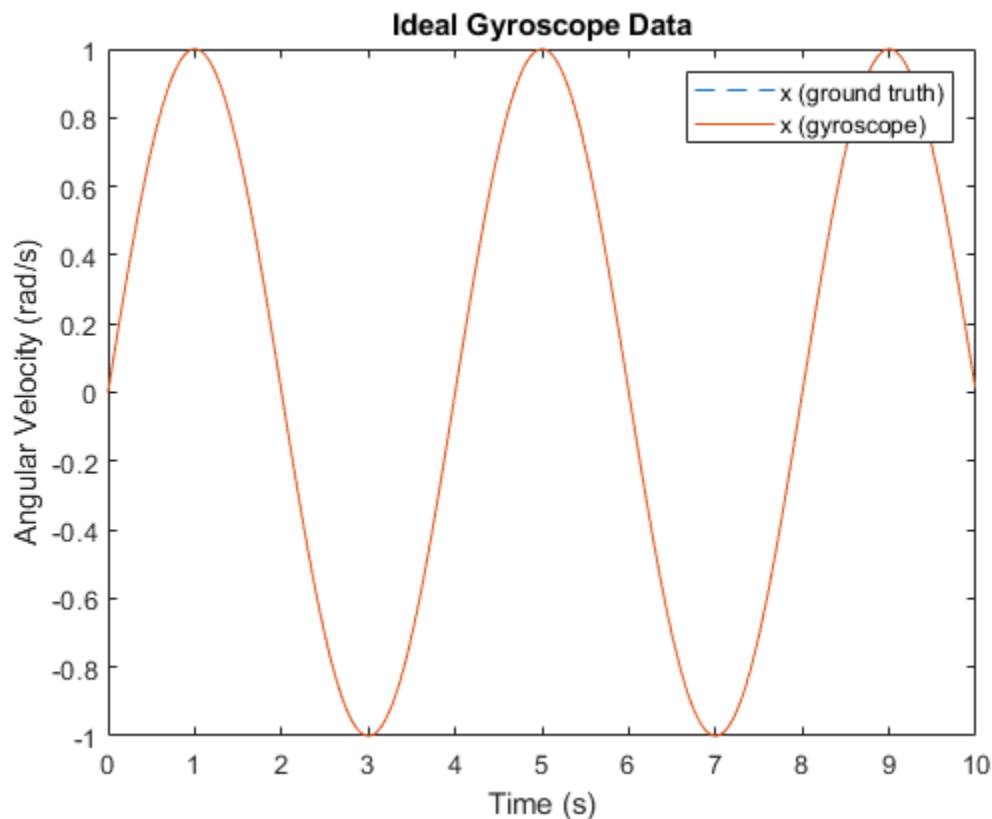
Hardware Parameter Tuning

The following parameters model hardware limitations or defects. Some can be corrected through calibration.

`MeasurementRange` determines the maximum absolute value reported by the gyroscope. Larger absolute values are saturated. The effect is shown by setting the measurement range to a value smaller than the amplitude of the sinusoidal ground-truth angular velocity.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.MeasurementRange = 0.5; % rad/s

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Saturated Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```

Saturated Gyroscope Data
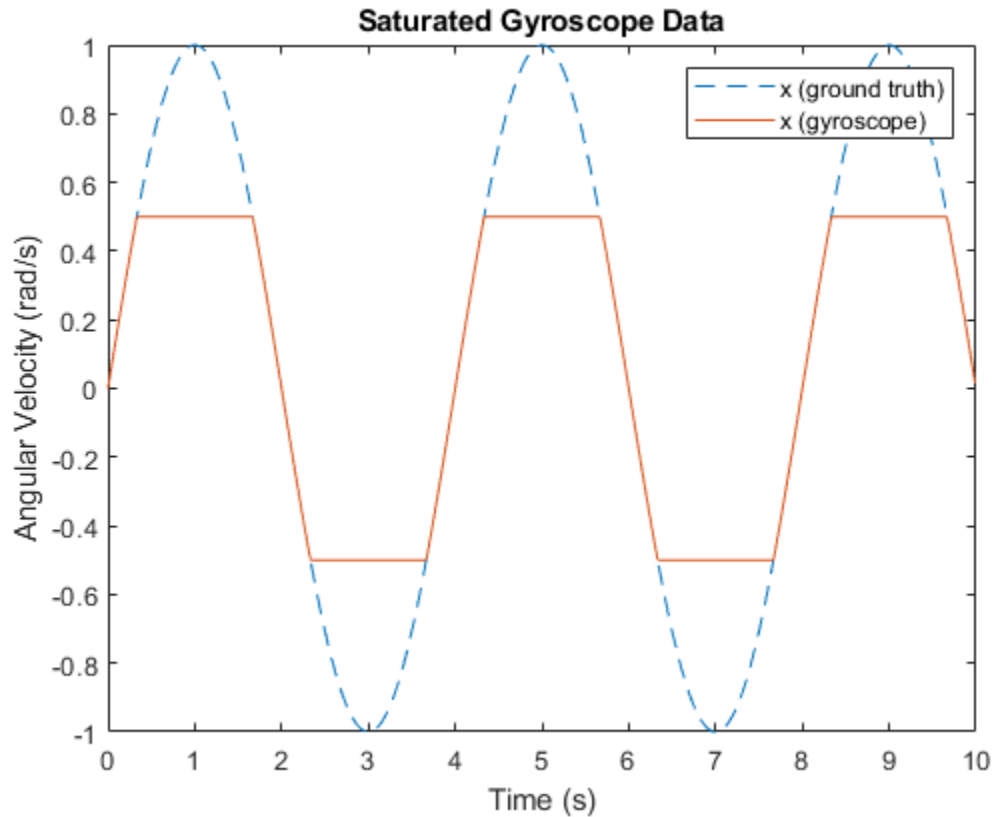
`Resolution` affects the step size of the digital measurements. Use this parameter to model the quantization effects from the analog-to-digital converter (ADC). The effect is shown by increasing the parameter to a much larger value than is typical.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.Resolution = 0.5; % (rad/s)/LSB

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Quantized Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```
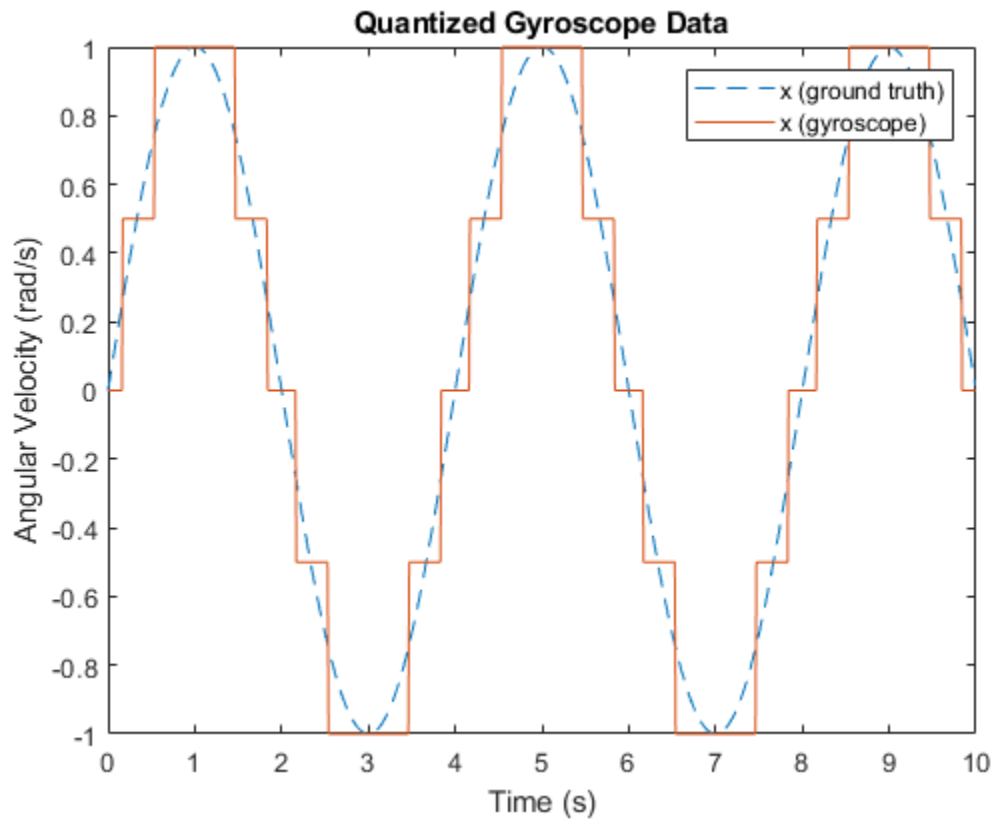
AxesMisalignment is the amount of skew in the sensor axes. This skew normally occurs when the sensor is mounted to the PCB and can be corrected through calibration. The effect is shown by skewing the x-axis slightly and plotting both the x-axis and y-axis.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.AxesMisalignment = [2 0 0]; % percent

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1:2), '--', t, gyroData(:,1:2))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Misaligned Gyroscope Data')
legend('x (ground truth)', 'y (ground truth)', ...
    'x (gyroscope)', 'y (gyroscope)')
```
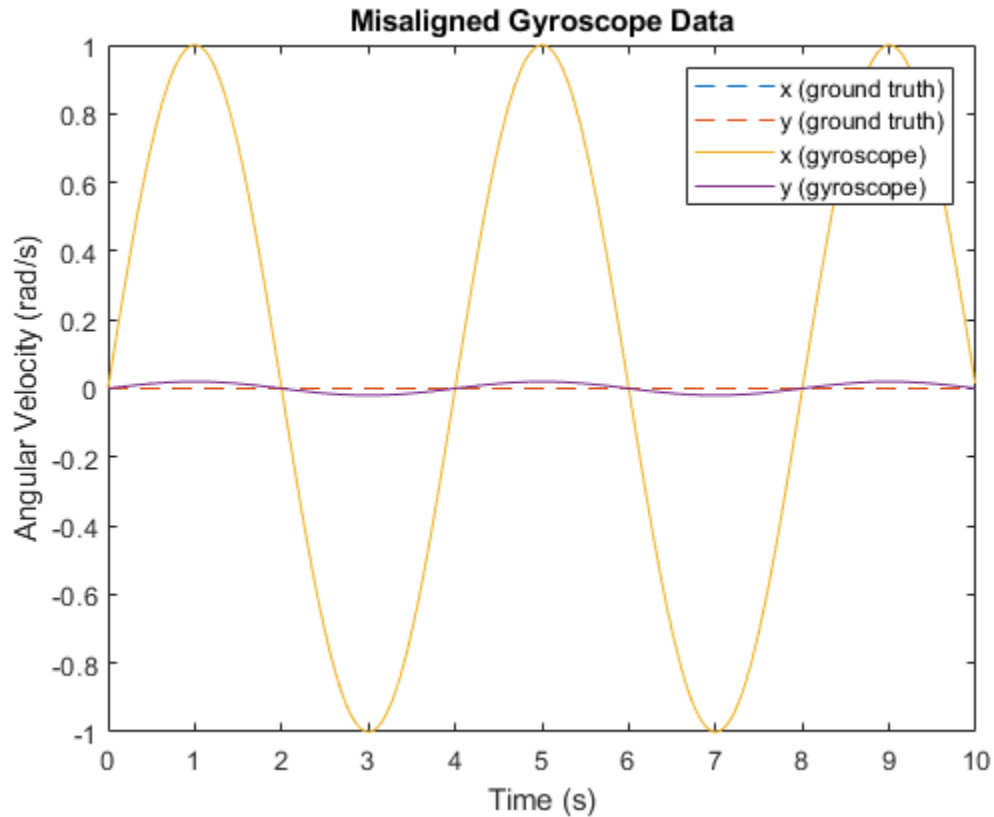
ConstantBias occurs in sensor measurements due to hardware defects. Since this bias is not caused by environmental factors, such as temperature, it can be corrected through calibration.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.ConstantBias = [0.4 0 0]; % rad/s

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Biased Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```

Biased Gyroscope Data
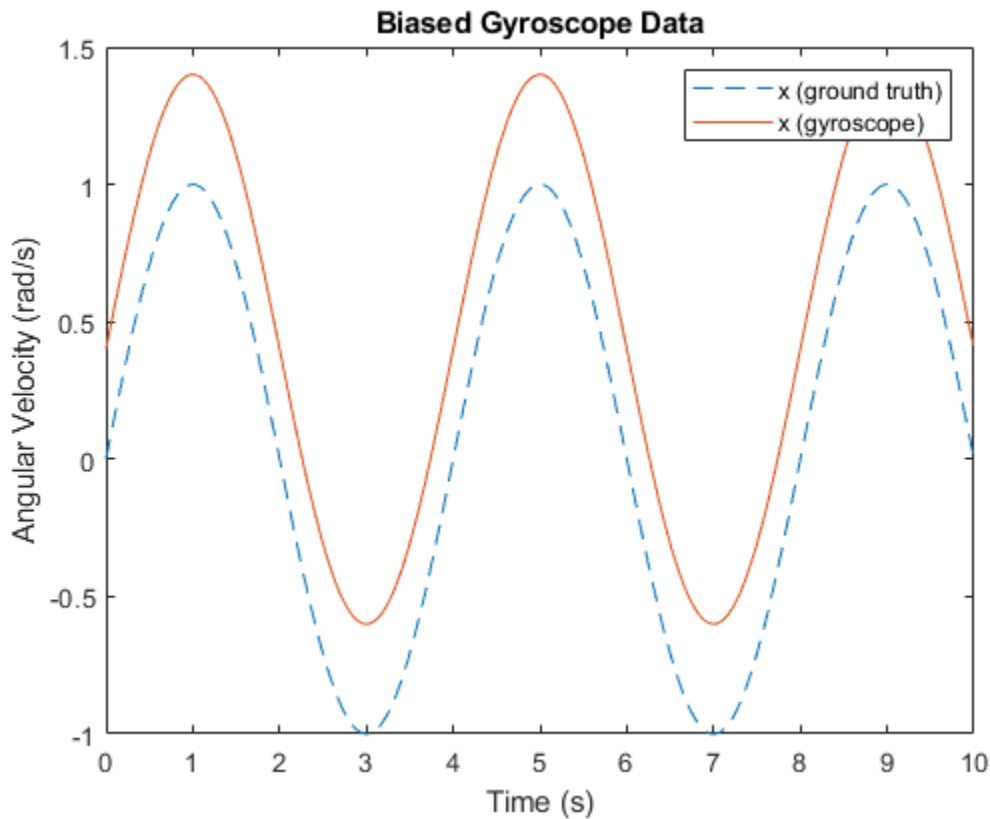
Random Noise Parameter Tuning

The following parameters model random noise in sensor measurements. More information on these parameters can be found in the "Inertial Sensor Noise Analysis Using Allan Variance" on page 5-97 example.

`NoiseDensity` is the amount of white noise in the sensor measurement. It is sometimes called angle random walk for gyroscopes or velocity random walk for accelerometers.

```
rng('default')

imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.NoiseDensity = 1.25e-2; % (rad/s)/sqrt(Hz)

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('White Noise Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```
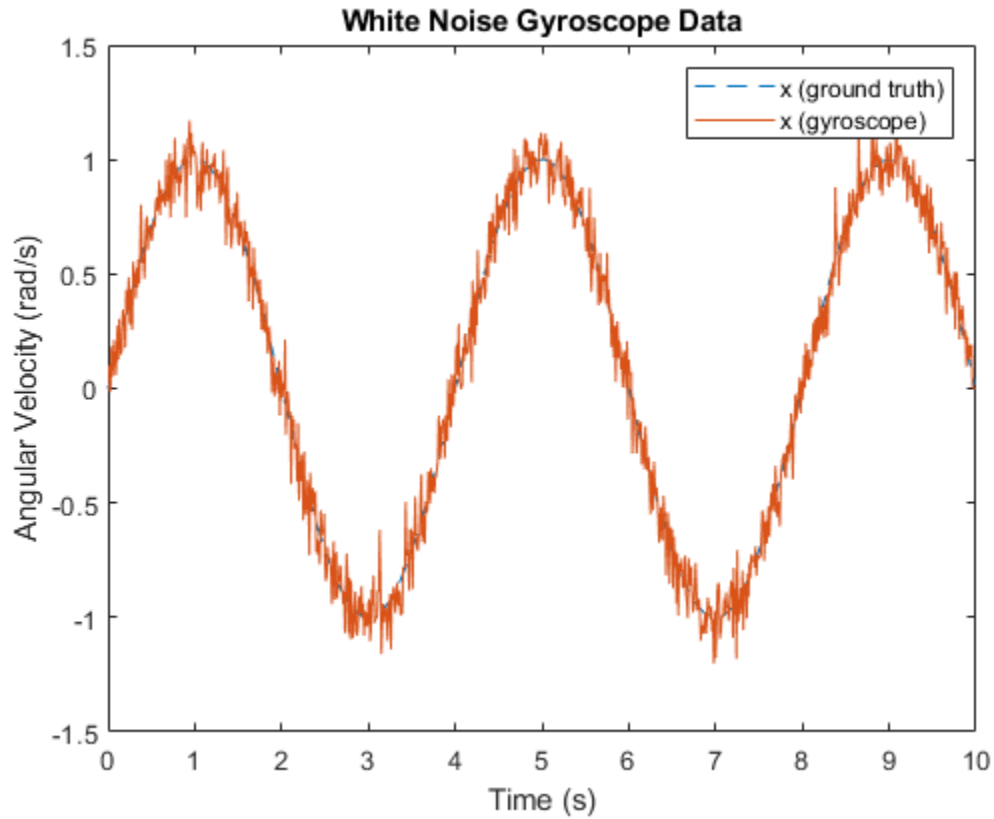
BiasInstability is the amount of pink or flicker noise in the sensor measurement.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.BiasInstability = 2.0e-2; % rad/s

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Bias Instability Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```

Bias Instability Gyroscope Data
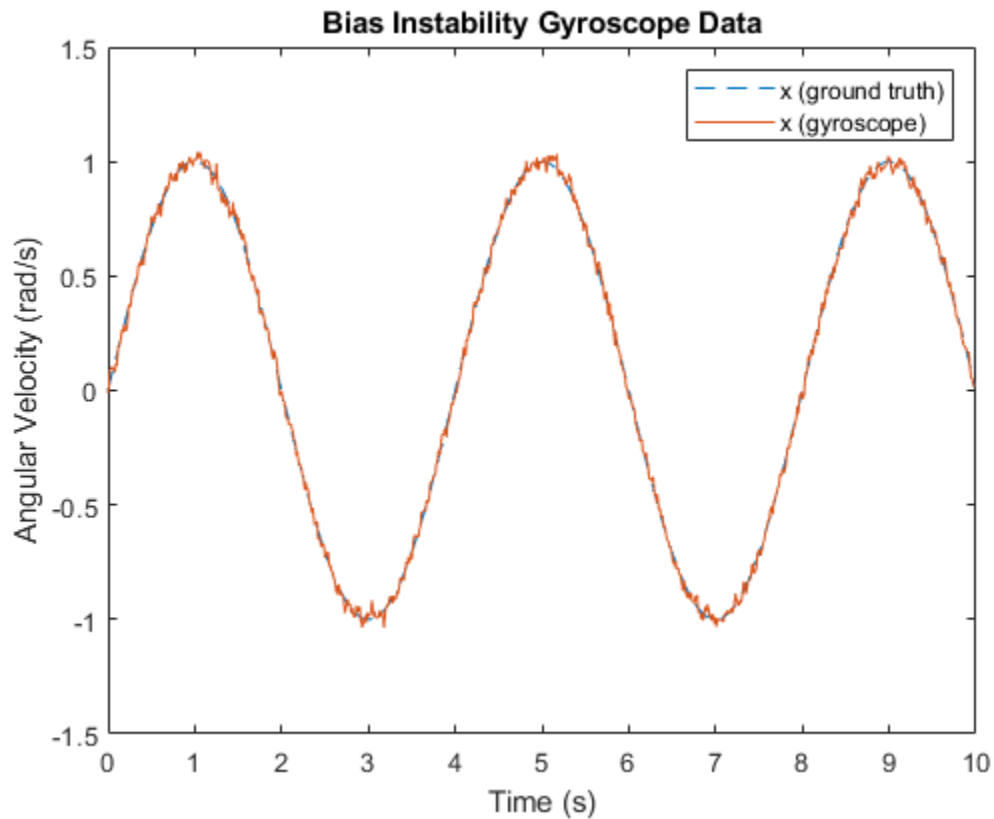
`RandomWalk` is the amount of Brownian noise in the sensor measurement. It is sometimes called rate random walk for gyroscopes or acceleration random walk for accelerometers.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.RandomWalk = 9.1e-2; % (rad/s)*sqrt(Hz)

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Random Walk Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```
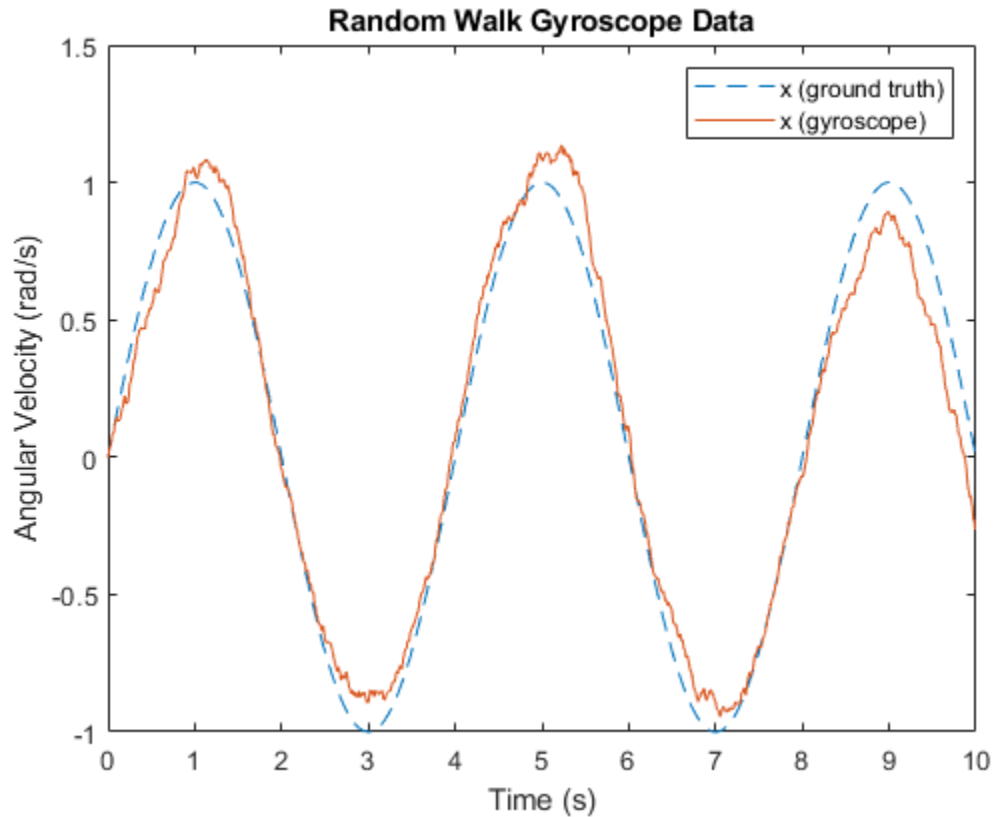
Environmental Parameter Tuning

The following parameters model noise that arises from changes to the environment of the sensor.

`TemperatureBias` is the bias added to sensor measurements due to temperature difference from the default operating temperature. Most sensor datasheets list the default operating temperature as 25 degrees Celsius. This bias is shown by setting the parameter to a non-zero value and setting the operating temperature to a value above 25 degrees Celsius.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.TemperatureBias = 0.06; % (rad/s)/(degrees C)
imu.Temperature = 35;

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Temperature-Biased Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```
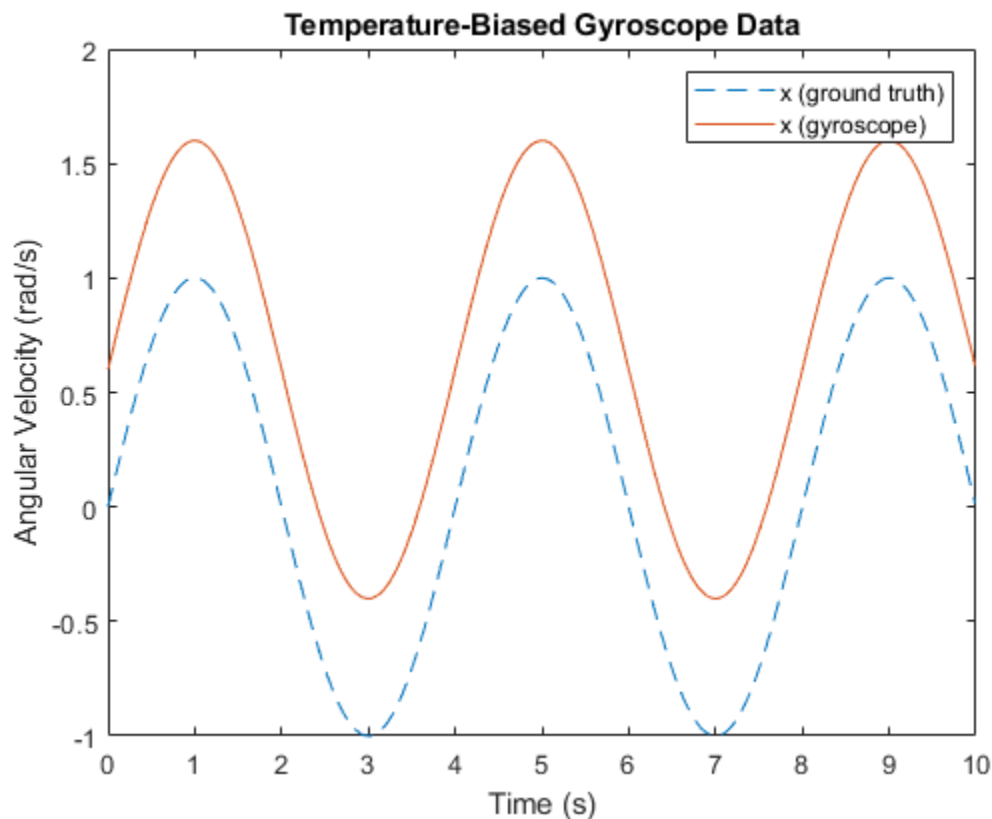
TemperatureScaleFactor is the error in the sensor scale factor due to changes in the operating temperature. This causes errors in the scaling of the measurement; in other words smaller ideal values have less error than larger values. This error is shown by linearly increasing the temperature.

```matlab
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.TemperatureScaleFactor = 3.2; % %/(degrees C)

standardTemperature = 25; % degrees C
temperatureSlope = 2; % (degrees C)/s

temperature = temperatureSlope*t + standardTemperature;

gyroData = zeros(N, 3);
for i = 1:N
    imu.Temperature = temperature(i);
    [~, gyroData(i,:)] = imu(acc(i,:), angvel(i,:));
end

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Temperature-Scaled Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```

Temperature-Scaled Gyroscope Data
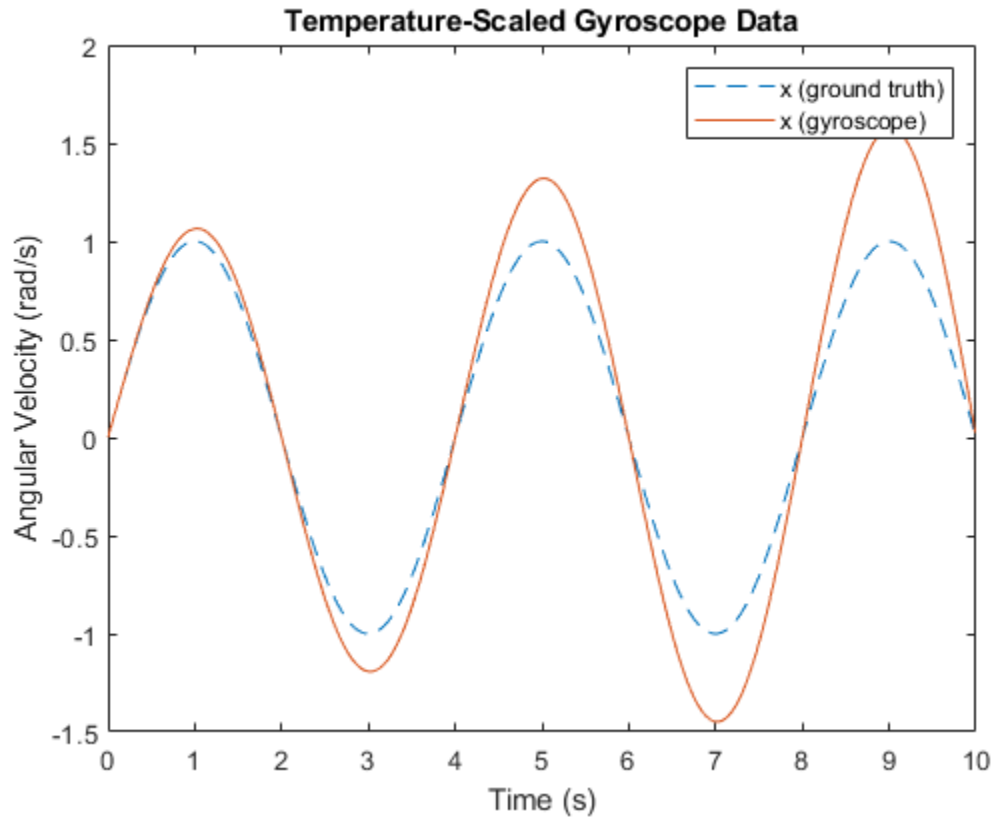
`AccelerationBias` is the bias added to the gyroscope measurement due to linear accelerations. This parameter is specific to the gyroscope. This bias is shown by setting the parameter to a non-zero value and using a non-zero input acceleration.

```
imu = imuSensor('SampleRate', Fs, 'Gyroscope', params);
imu.Gyroscope.AccelerationBias = 0.3; % (rad/s)/(m/s^2)

acc(:,1) = 1;

[~, gyroData] = imu(acc, angvel);

figure
plot(t, angvel(:,1), '--', t, gyroData(:,1))
xlabel('Time (s)')
ylabel('Angular Velocity (rad/s)')
title('Acceleration-Biased Gyroscope Data')
legend('x (ground truth)', 'x (gyroscope)')
```

Acceleration-Biased Gyroscope Data
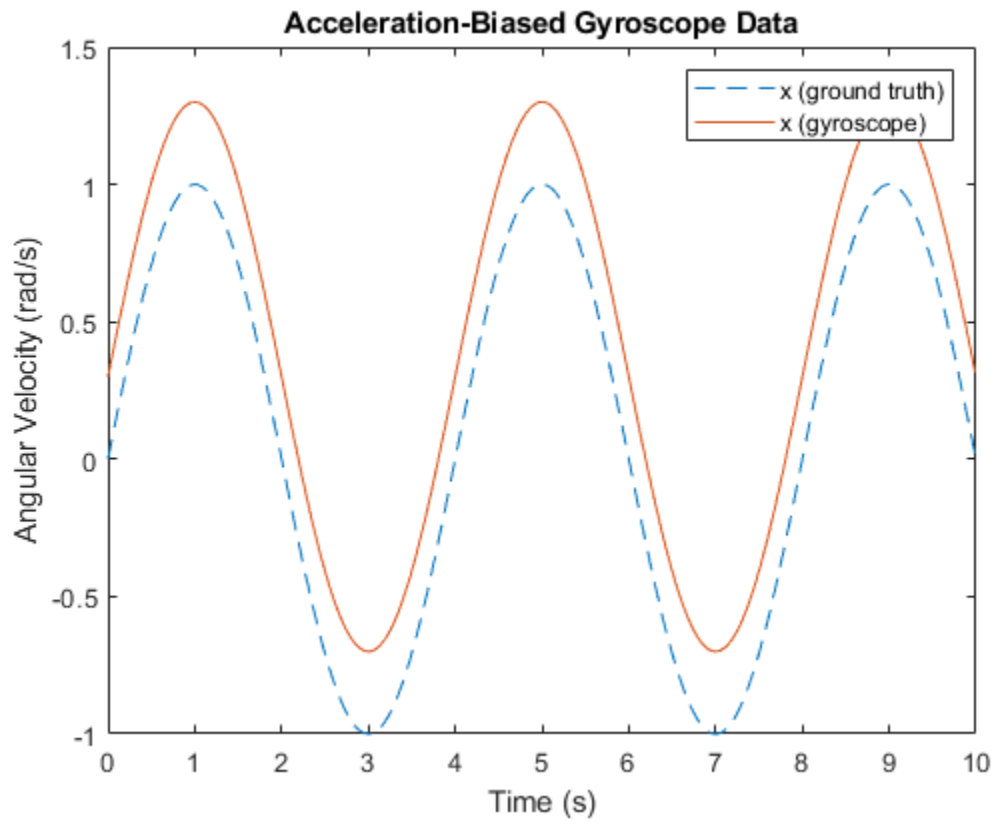
# Introduction to Using the Global Nearest Neighbor Tracker

This example shows how to configure and use the global nearest neighbor (GNN) tracker.

Motivation

The `trackerGNN` is a global nearest neighbor (GNN), single-hypothesis tracker. The `trackerGNN` allows you to:

1 Choose the assignment algorithm to associate detections with tracks.

2 Use either history-based or score-based track logic for confirmation and deletion of tracks.

3 Use any kind of tracking filter, including an interacting multiple model filter.

4 Connect the tracker to scanning and managed sensors that update only a subset of the tracks managed by the tracker.

5 Predict tracks into the future without modifying their internal state. This allows you to display the predicted state of the tracks or to provide track predictions to a sensor resource manager.

Construct and Use the `trackerGNN`

You can construct the `trackerGNN` and choose one of the assignment algorithms. By default, the `trackerGNN` uses the `'Munkres'` algorithm, which guarantees an optimal assignment, but may take more time to compute. You can use `'Auction'` or `'Jonker-Volgenant'` or provide a `'Custom'` function of your own. In this example, you choose the `'Auction'` algorithm.

```
tracker = trackerGNN('Assignment','Auction')
```

```
tracker =

  trackerGNN with properties:

                TrackerIndex: 0
      FilterInitializationFcn: 'initcvekf'
                  Assignment: 'Auction'
         AssignmentThreshold: [30 Inf]
                MaxNumTracks: 100
               MaxNumSensors: 20

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

          HasCostMatrixInput: false
  HasDetectableTrackIDsInput: false
              StateParameters: [1x1 struct]

                    NumTracks: 0
           NumConfirmedTracks: 0
```

The main way of using the `trackerGNN` is by calling it with new detections at each simulation step. A detection is an `objectDetection` input or a struct with similar fields. You must specify the time of the detection and its measurement. The other properties have default values. For example:

```
detections = {objectDetection(0,[1;2;3]); % Using default values on the detection ...
    objectDetection(0, [10;0;0], 'ObjectClassID', 2)}; % Using a non-default object class
disp(detections{1})
```

```
  objectDetection with properties:

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

```
time = 0;
[confirmedTracks, tentativeTracks] = tracker(detections, time);
disp(confirmedTracks)
disp(tentativeTracks)
```

```
  objectTrack with properties:

              TrackID: 2
             BranchID: 0
          SourceIndex: 0
           UpdateTime: 0
                  Age: 1
                State: [6x1 double]
       StateCovariance: [6x6 double]
       StateParameters: [1x1 struct]
         ObjectClassID: 2
            TrackLogic: 'History'
       TrackLogicState: [1 0 0 0 0]
           IsConfirmed: 1
             IsCoasted: 0
        IsSelfReported: 1
       ObjectAttributes: [1x1 struct]
```

```
  objectTrack with properties:

              TrackID: 1
             BranchID: 0
          SourceIndex: 0
           UpdateTime: 0
                  Age: 1
                State: [6x1 double]
       StateCovariance: [6x6 double]
       StateParameters: [1x1 struct]
         ObjectClassID: 0
            TrackLogic: 'History'
       TrackLogicState: [1 0 0 0 0]
           IsConfirmed: 0
             IsCoasted: 0
        IsSelfReported: 1
       ObjectAttributes: [1x1 struct]
```

Two types of tracks are created: confirmed and tentative. A confirmed track is a track that is considered to be an estimation of a real target, while a tentative track may still be a false target. The

IsConfirmed flag distinguishes between the two. The track created by the second detection has a nonzero ObjectClassID field and is immediately confirmed, because the sensor that reported it has been able to classify it and thus it is considered a real target. Alternatively, a track can be confirmed if there is enough evidence of its existence. In the history-based confirmation logic used here, if the track has been assigned 2 detections out of 3 it will be confirmed. This is controlled by the ConfirmationThreshold property. For example, the next detection is assigned to the tentative track and confirms it:

```
detections = {objectDetection(1,[1.1;2.2;3.3])};
time = time + 1; % Time must increase from one update of the tracker to the next
confirmedTracks = tracker(detections,time);
confirmedTracks(1)
```

```
ans =

  objectTrack with properties:

            TrackID: 1
           BranchID: 0
        SourceIndex: 0
         UpdateTime: 1
                Age: 2
              State: [6x1 double]
     StateCovariance: [6x6 double]
     StateParameters: [1x1 struct]
       ObjectClassID: 0
          TrackLogic: 'History'
      TrackLogicState: [1 1 0 0 0]
         IsConfirmed: 1
           IsCoasted: 0
       IsSelfReported: 1
      ObjectAttributes: [1x1 struct]
```

Use a Score-Based Confirmation and Deletion Logic

In many cases, the history-based confirmation and deletion logic is considered too simplistic, as it does not take into account statistical metrics. These metrics include the sensor's probability of detection and false alarm rate, the likelihood of new targets to appear, or the distance between a detection and the estimated state of the track assigned to it. A score-based confirmation and deletion logic takes into account such metrics and provides a more suitable statistical test.

To convert the tracker to a score-based confirmation and deletion logic, first release the tracker and then set the tracker's TrackLogic to 'Score':

```
release(tracker)
tracker.TrackLogic = 'Score'
```

```
tracker =

  trackerGNN with properties:

              TrackerIndex: 0
       FilterInitializationFcn: 'initcvekf'
                 Assignment: 'Auction'
```

```
            AssignmentThreshold: [30 Inf]
                  MaxNumTracks: 100
                 MaxNumSensors: 20

                    TrackLogic: 'Score'
         ConfirmationThreshold: 20
             DeletionThreshold: -7
          DetectionProbability: 0.9000
                FalseAlarmRate: 1.0000e-06
                        Volume: 1
                          Beta: 1

            HasCostMatrixInput: false
     HasDetectableTrackIDsInput: false
               StateParameters: [1x1 struct]

                     NumTracks: 0
            NumConfirmedTracks: 0
```

Notice that the confirmation and deletion thresholds have changed to scalar values, which represent the score used to confirm and delete a track. In addition, several more properties are now used to provide the parameters for the score-based confirmation and deletion.

Now, update the tracker to see the tracks confirmation.

```matlab
detections = {objectDetection(0,[1;2;3]); % Using default values on the detection ...
    objectDetection(0, [10;0;0], 'ObjectClassID', 2)}; % Using a non-default object class
time = 0;
tracker(detections, time); % Same as the first step above
detections = {objectDetection(1,[1.1;2.2;3.3])};
time = time + 1; % Time must increase from one update of the tracker to the next
[confirmedTracks, tentativeTracks] = tracker(detections,time);
confirmedTracks
```

```
confirmedTracks =

  objectTrack with properties:

             TrackID: 2
            BranchID: 0
         SourceIndex: 0
          UpdateTime: 1
                 Age: 2
               State: [6x1 double]
      StateCovariance: [6x6 double]
      StateParameters: [1x1 struct]
        ObjectClassID: 2
           TrackLogic: 'Score'
      TrackLogicState: [11.4076 13.7102]
          IsConfirmed: 1
            IsCoasted: 1
       IsSelfReported: 1
     ObjectAttributes: [1x1 struct]
```

Because the confirmed track was not assigned to any detection in this update, its score decreased. You can see that by looking at the `TrackLogicState` field and seeing that the first element, the current score, is lower than the second element, the maximum score. If the track continues to decrease relative to the maximum score, by more than the `DeletionThreshold` value, the track is deleted.

tentativeTracks

```
tentativeTracks =

  objectTrack with properties:

             TrackID: 1
            BranchID: 0
         SourceIndex: 0
          UpdateTime: 1
                 Age: 2
               State: [6x1 double]
      StateCovariance: [6x6 double]
      StateParameters: [1x1 struct]
        ObjectClassID: 0
           TrackLogic: 'Score'
       TrackLogicState: [17.7217 17.7217]
          IsConfirmed: 0
            IsCoasted: 0
       IsSelfReported: 1
      ObjectAttributes: [1x1 struct]
```

If the tracks are not assigned to any detections, they will first be coasted and after a few 'misses' they will be deleted. To see that, call the tracker with no detections:

```
for i = 1:3
    time = time + 1;
    [~,~,allTracks] = tracker({},time)
end
```

```
allTracks =

  2x1 objectTrack array with properties:

    TrackID
    BranchID
    SourceIndex
    UpdateTime
    Age
    State
    StateCovariance
    StateParameters
    ObjectClassID
    TrackLogic
    TrackLogicState
    IsConfirmed
    IsCoasted
    IsSelfReported
    ObjectAttributes
```

```
allTracks =

  2x1 objectTrack array with properties:

    TrackID
    BranchID
    SourceIndex
    UpdateTime
    Age
    State
    StateCovariance
    StateParameters
    ObjectClassID
    TrackLogic
    TrackLogicState
    IsConfirmed
    IsCoasted
    IsSelfReported
    ObjectAttributes


allTracks =

  objectTrack with properties:

              TrackID: 1
             BranchID: 0
          SourceIndex: 0
           UpdateTime: 4
                  Age: 5
                State: [6x1 double]
      StateCovariance: [6x6 double]
      StateParameters: [1x1 struct]
        ObjectClassID: 0
           TrackLogic: 'Score'
      TrackLogicState: [10.8139 17.7217]
           IsConfirmed: 0
             IsCoasted: 1
        IsSelfReported: 1
      ObjectAttributes: [1x1 struct]
```

The second track was deleted because it was not assigned any detections in 4 updates. This caused its score to fall by more than 7, the value of the `DeletionThreshold`. The first track is still not deleted, but its score is now lower and close to the deletion threshold.

Use Any Tracking Filter

The trackerGNN supports any tracking filter that implements the tracking filter interface. The selection of filter initialization function is defined using the `FilterInitializationFcn` property of the trackerGNN. This provides the following flexibility:

1 You can use any filter initialization function available in the product. Some examples include `initcvekf` (default), `initcvkf` , `initcvukf` , `initcvckf` , `initcaekf` , etc.

2 You can write your own filter initialization function and use any tracking filter. These include `trackingABF`, `trackingEKF`, `trackingKF`, `trackingUKF`, `trackingCKF`, `trackingPF`, `trackingMSCEKF`, `trackingGSF`, and `trackingIMM`.

3 You can write a tracking filter that inherits and implements the interface defined by the abstract `matlabshared.tracking.internal.AbstractTrackingFilter` class.

The following example shows how to use an interacting motion model (IMM) filter that has 3 types of motion models: constant velocity, constant acceleration and constant turn rate.

Modify the tracker to use an IMM filter

```
release(tracker) % Release the tracker
tracker.FilterInitializationFcn = 'initekfimm'


tracker =

  trackerGNN with properties:

                 TrackerIndex: 0
       FilterInitializationFcn: 'initekfimm'
                   Assignment: 'Auction'
          AssignmentThreshold: [30 Inf]
                 MaxNumTracks: 100
                MaxNumSensors: 20

                   TrackLogic: 'Score'
          ConfirmationThreshold: 20
             DeletionThreshold: -7
          DetectionProbability: 0.9000
                FalseAlarmRate: 1.0000e-06
                       Volume: 1
                         Beta: 1

             HasCostMatrixInput: false
     HasDetectableTrackIDsInput: false
               StateParameters: [1x1 struct]

                    NumTracks: 0
            NumConfirmedTracks: 0
```

Next, update the tracker with a detection and observe the three motion models that comprise it. You can see which model is used by looking at the `StateTransitionFcn` of each filter.

```
% Update the tracker with a single detection to get a single track
detection = {objectDetection(0, [1;2;3], 'ObjectClassID', 2)};
time = 0;
tracker(detection, time);
```

Use the `getTrackFilterProperties` method to view the `TrackingFilters` property. It returns a cell array that contains the `TrackingFilters` property: `{filter1;filter2;filter3}`

```
filters = getTrackFilterProperties(tracker,1,'TrackingFilters');
for i = 1:numel(filters{1})
    disp(filters{1}{i})
end
```

```
trackingEKF with properties:

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

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

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

trackingEKF with properties:

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

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

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

trackingEKF with properties:

                      State: [7x1 double]
            StateCovariance: [7x7 double]

           StateTransitionFcn: @constturn
    StateTransitionJacobianFcn: @constturnjac
                ProcessNoise: [4x4 double]
       HasAdditiveProcessNoise: 0

               MeasurementFcn: @ctmeas
       MeasurementJacobianFcn: @ctmeasjac
            MeasurementNoise: [3x3 double]
  HasAdditiveMeasurementNoise: 1
```

Interface with Scanning and Managed Sensors

By default, the tracker assumes that each step updates all the tracks managed by the tracker in a coverage area. This is not the case when sensors have limited coverage, and scan a small area, or when they are managed and cued to scan certain areas out of the total coverage area. If that is the case, the sensors should let the tracker know that some tracks were not covered by the sensors at that step. Otherwise, the tracker assumes that the tracks were supposed to be detected and will count a 'miss' against them, leading to their premature deletion.

The following example shows how the sensors signify that the track will not be detected, and how the track is not deleted.

Create a tracker that allows feedback from the sensors.

```
release(tracker) % Release the tracker
tracker.FilterInitializationFcn = 'initcvekf';
tracker.HasDetectableTrackIDsInput = true % Allows the tracker to get input about the track detec
% Update the tracker with a single detection to get a single track
detection = {objectDetection(0, [1;2;3], 'ObjectClassID', 2)};
time = 0;
trackIDs = []; % Initially, there are no tracks, so trackIDs has zero rows
track = tracker(detection, time, trackIDs)
% Update the tracker 2 more times without any detections. Let the tracker
% know that the track was not detectable by any sensor. Note how the
% TrackLogicState, shown as [currentScore, maxScore], does not change even
% though the track is not detected.
for i=1:2
    time = time + 1;
    trackIDs = [1, 0]; % Zero probability of detection means the track score should not decrease
    track = tracker({}, time, trackIDs) % No detections
end


tracker =

  trackerGNN with properties:

                   TrackerIndex: 0
         FilterInitializationFcn: 'initcvekf'
                     Assignment: 'Auction'
            AssignmentThreshold: [30 Inf]
                   MaxNumTracks: 100
                  MaxNumSensors: 20

                      TrackLogic: 'Score'
            ConfirmationThreshold: 20
                DeletionThreshold: -7
             DetectionProbability: 0.9000
                   FalseAlarmRate: 1.0000e-06
                          Volume: 1
                            Beta: 1

                HasCostMatrixInput: false
        HasDetectableTrackIDsInput: true
                  StateParameters: [1x1 struct]

                       NumTracks: 0
               NumConfirmedTracks: 0


track =

  objectTrack with properties:

                TrackID: 1
                BranchID: 0
             SourceIndex: 0
              UpdateTime: 0
                     Age: 1
                   State: [6x1 double]
```

```
        StateCovariance: [6x6 double]
        StateParameters: [1x1 struct]
          ObjectClassID: 2
             TrackLogic: 'Score'
         TrackLogicState: [13.7102 13.7102]
            IsConfirmed: 1
              IsCoasted: 0
         IsSelfReported: 1
       ObjectAttributes: [1x1 struct]


track =

  objectTrack with properties:

               TrackID: 1
              BranchID: 0
           SourceIndex: 0
            UpdateTime: 1
                   Age: 2
                 State: [6x1 double]
       StateCovariance: [6x6 double]
       StateParameters: [1x1 struct]
         ObjectClassID: 2
            TrackLogic: 'Score'
        TrackLogicState: [13.7102 13.7102]
           IsConfirmed: 1
             IsCoasted: 1
        IsSelfReported: 1
      ObjectAttributes: [1x1 struct]


track =

  objectTrack with properties:

               TrackID: 1
              BranchID: 0
           SourceIndex: 0
            UpdateTime: 2
                   Age: 3
                 State: [6x1 double]
       StateCovariance: [6x6 double]
       StateParameters: [1x1 struct]
         ObjectClassID: 2
            TrackLogic: 'Score'
        TrackLogicState: [13.7102 13.7102]
           IsConfirmed: 1
             IsCoasted: 1
        IsSelfReported: 1
      ObjectAttributes: [1x1 struct]
```

As seen, the track score did not decrease and the track was not deleted by the tracker, even though it was not detected in 5 updates.

Predict the Tracks to a Certain Time

The last enhancement allows you to predict the tracks into the future without changing their internal state. There are two common use cases for this:

1   Displaying the predicted tracks on a display.

2   Passing the predicted tracks to a sensor system so that the sensor system can cue a search
    pattern to detect them.

You use the `predictTracksToTime` method to get the predicted tracks.

Update the tracker with more detections

```
time = time + 1;
detections = {objectDetection(time, [4,2,3]); ...
    objectDetection(time, [10;0;0])};
track = tracker(detections, time, trackIDs);
disp('State of track #1 at time 3:')
disp(track.State)

% Predict tracks to different time steps:
predictedTrack1 = predictTracksToTime(tracker,1, time+0.5); % Predict track number 1 half a secon
disp('State of track #1 at time 3.5:')
predictedTrack1.State
% Predict all the confirmed tracks 2 seconds to the future
predictedConfirmedTracks = predictTracksToTime(tracker, 'Confirmed', time+2);
disp('State of track #1 at time 5:')
predictedConfirmedTracks.State

% Predict all the tracks 0.3 seconds to the future
disp('State of all the tracks at time 3.3:')
predictedTracks = predictTracksToTime(tracker, 'all', time+0.3);
predictedTracks.State
```

```
State of track #1 at time 3:
    3.9967
    1.0030
    2.0000
         0
    3.0000
         0

State of track #1 at time 3.5:

ans =

    4.4982
    1.0030
    2.0000
         0
    3.0000
         0

State of track #1 at time 5:

ans =
```

```
        6.0027
        1.0030
        2.0000
             0
        3.0000
             0
```

State of all the tracks at time 3.3:

```
ans =

        4.2976
        1.0030
        2.0000
             0
        3.0000
             0


ans =

        10
         0
         0
         0
         0
         0
```

You can use the `predictTracksToTime` method to visualize the predicted state of the tracks.

```matlab
% First, use a |theaterPlot| and a |trackPlotter| to plot the tracks.
thPlot = theaterPlot('XLimits',[-20 20], 'Ylimits', [-20 20]);
trPlotter = trackPlotter(thPlot, 'DisplayName', 'Predicted Track');
posSelector = [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 1 0];
velSelector = [0 1 0 0 0 0; 0 0 0 1 0 0; 0 0 0 0 0 1];

% Then, plot the predicted tracks every 0.1 seconds
for t = time+(0.1:0.1:5)
    predictedTracks = predictTracksToTime(tracker, 'Confirmed', t);
    [pos,cov] = getTrackPositions(predictedTracks,posSelector);
    vel = getTrackVelocities(predictedTracks,velSelector);
    plotTrack(trPlotter,pos,vel,cov);
    drawnow
end
```
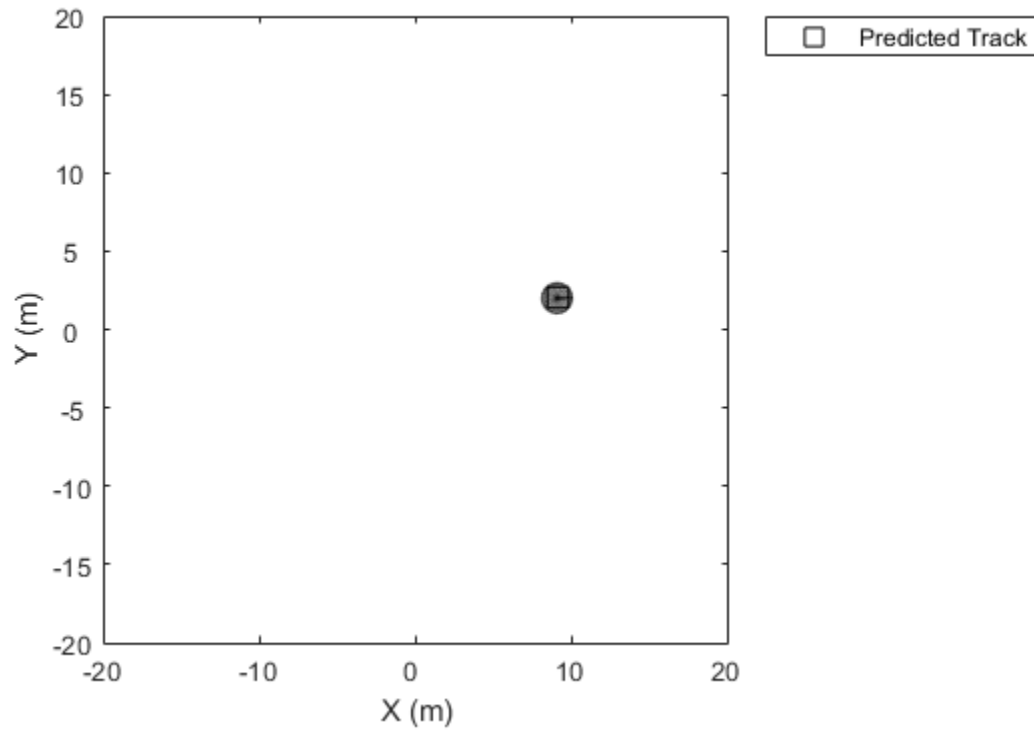
Summary

In this example, you created a `trackerGNN` and used it to track multiple targets. You modified the tracker to use various assignment algorithms, two types of confirmation and deletion logic, and various tracking filters. In addition, you saw how to interface the tracker with a scanning radar and how to get the track predictions for display or sensor management.

Introduction to Track Logic

This example shows how to define and use confirmation and deletion logic that are based on history or score. It introduces the `trackHistoryLogic` and `trackScoreLogic` objects and shows how to use them as stand-alone objects and how to use them as part of the `trackerGNN`.

Introduction

The tracker maintains a list of tracks, or estimates of the target states in the area of interest. If a detection cannot be assigned to any track already maintained by the tracker, the tracker initiates a new track. In most cases, it is unclear whether the new track represents a true target or a false one. At first, a track is created with *tentative* status. If enough evidence is obtained, the track becomes *confirmed*. Similarly, if no detections are assigned to a track, the track is *coasted* (predicted without correction), but after a few missed updates, the tracker deletes the track.

There are two main ways of confirming and deleting tracks used in the literature:

1   History-based: the tracker counts the number of detections assigned to a track in several recent updates. If enough detections are assigned, the track is confirmed. If the track is not assigned to any detection for enough updates, it is deleted. This type of logic is often referred to as *M-out-of-N* or *Last-N*, meaning that out of *N* updates, the track must be detected at least _M _ times for it to be confirmed.

2   Score-based: the tracker calculates the *likelihood* that a track is of a real target. Instead of likelihood, we use the *score*, defined as the log of the likelihood. A high positive track score means that the track is very likely to be of a real target. A very negative track score means that the track is likely to be false. As a result, we can set a threshold for confirming a track if the score is high enough. If the score is low, or falls enough from the maximum score, the track is deleted.

In the following sections, you can see how to define and use the two types of objects that implement the history-based and score-based track logic.

Use a History-Based Track Logic to Confirm and Delete a Track

The simplest type of track logic is based on history. This type of logic counts how many times the track is detected (or missed) in the recent *N* updates. You want to confirm a track after 3 detections in 5 updates (3-out-of-5), and delete it after 6 consecutive misses. First, create the `trackHistoryLogic` object that maintains the track history by defining the confirmation and deletion thresholds.

```
historyLogic = trackHistoryLogic('ConfirmationThreshold', [3 5], 'DeletionThreshold', 6)

historyLogic =
  trackHistoryLogic with properties:

    ConfirmationThreshold: [3 5]
        DeletionThreshold: [6 6]
                  History: [0 0 0 0 0 0]
```

To illustrate this, in the first 5 updates, the track is detected every other update. The following shows how the track logic gets confirmed after exactly 3 out of 5 hits. Use the `init` method to initialize the object with the first hit. Then use either the `hit` or `miss` methods to indicate whether the track logic is updated by a hit or a miss, respectively.

The `checkConfirmation` method is used to check if the track can be confirmed based on its history. Use the `output` method to get the track history, which is a logical array of length `Nmax = max(Nconf, Ndel)`. In this example, `Nmax` is 6.

```
wasInitialized = false; % Has the object been initialized yet?
for i = 1:5
    detectedFlag = logical(mod(i,2)); % Only odd updates are true
    if detectedFlag && ~wasInitialized
        init(historyLogic)
        wasInitialized = true;
    elseif detectedFlag && wasInitialized
        hit(historyLogic)
    else
        miss(historyLogic)
    end
    history = output(historyLogic);
    confFlag = checkConfirmation(historyLogic);
    disp(['Track history is: ', num2str(history),'.  Confirmation Flag is: ',num2str(confFlag)])
end
```

```
Track history is: 1  0  0  0  0  0.  Confirmation Flag is: 0
Track history is: 0  1  0  0  0  0.  Confirmation Flag is: 0
Track history is: 1  0  1  0  0  0.  Confirmation Flag is: 0
Track history is: 0  1  0  1  0  0.  Confirmation Flag is: 0
Track history is: 1  0  1  0  1  0.  Confirmation Flag is: 1
```

Now, suppose that the track is not detected for several updates. After the sixth update it should be deleted. Use the `checkDeletion` method to check if the track was deleted.

```
for i = 1:6
    miss(historyLogic); % Every update the track is not detected
    history = output(historyLogic);
    deleteFlag = checkDeletion(historyLogic);
    disp(['Track history is: ', num2str(history),'.  Deletion Flag is: ',num2str(deleteFlag)])
end
```

```
Track history is: 0  1  0  1  0  1.  Deletion Flag is: 0
Track history is: 0  0  1  0  1  0.  Deletion Flag is: 0
Track history is: 0  0  0  1  0  1.  Deletion Flag is: 0
Track history is: 0  0  0  0  1  0.  Deletion Flag is: 0
Track history is: 0  0  0  0  0  1.  Deletion Flag is: 0
Track history is: 0  0  0  0  0  0.  Deletion Flag is: 1
```

Use a Score-Based Track Logic to Confirm and Delete a Track

In many cases, it is not enough to know if a track is assigned a detection. You may need to account for the likelihood that the assignment is correct. There is also a need to know how likely the detection is that of a real target, based on its *detection probability*, or how likely it is to be false, based on its *false alarm rate*. Additionally, if the track is new, you need to account for the rate, *beta* , at which new targets are likely to be detected in a unit volume.

Use the `trackScoreLogic` object to create a score-based track confirmation and deletion logic. Define the `ConfirmationThreshold` and `DeletionThreshold` as two scalar values.

When a track is assigned a detection, you update the track logic with a hit and in most cases the track score increases. The `ConfirmationThreshold` defines the minimum score that is required to confirm a track.

The track score decreases when a track is not assigned a detection. The `DeletionThreshold` is used to define how much we allow the score to decrease from the maximum score before deleting the track.

```
scoreLogic = trackScoreLogic('ConfirmationThreshold', 25, 'DeletionThreshold', -5)

scoreLogic =
  trackScoreLogic with properties:

    ConfirmationThreshold: 25
        DeletionThreshold: -5
                    Score: 0
                 MaxScore: 0
```

```
pd = 0.9;    % Probability of detection
pfa = 1e-6; % Probability of false alarm
volume = 1; % The volume of a sensor detection bin
beta = 0.1; % New target rate in a unit volume
wasInitialized = false; % Has the object been initialized yet?
```

Confirming a track using score-based logic is very similar to confirming a track using history-based logic. The main difference is that now additional parameters are taken into account that include the statistical performance of the sensor as well as the residual distance of the track relative to the detection.

You use the `init` , `hit` , and `miss` methods to initialize on first hit, update the `trackScoreLogic` with a subsequent hit, or update with a miss, respectively.

You use the `checkConfirmation` method to check if the track should be confirmed. You use the `output` method to get the current score and maximum score as a `[currentScore, maxScore]` array.

```
r = rng(2018); % Set the random seed for repeatable results
numSteps1 = 6;
scores = zeros(numSteps1,2);
for i = 1:numSteps1
    l = 0.05 + 0.05 * rand; % likelihood of the measurement
    detectedFlag = logical(mod(i,2)); % Only even updates are true in this example
    if detectedFlag && ~wasInitialized
        init(scoreLogic, volume, beta);
        wasInitialized = true;
    elseif detectedFlag && wasInitialized
        hit(scoreLogic, volume, l);
    else
        miss(scoreLogic);
    end
    scores(i,:) = output(scoreLogic);
    confFlag = checkConfirmation(scoreLogic);
    disp(['Score and MaxScore: ', num2str(scores(i,:)),'.  Confirmation Flag is: ',num2str(confF
end
```

```
Score and MaxScore: 11.4076      11.4076.  Confirmation Flag is: 0
Score and MaxScore: 9.10498      11.4076.  Confirmation Flag is: 0
Score and MaxScore: 20.4649      20.4649.  Confirmation Flag is: 0
Score and MaxScore: 18.1624      20.4649.  Confirmation Flag is: 0
Score and MaxScore: 29.2459      29.2459.  Confirmation Flag is: 1
Score and MaxScore: 26.9433      29.2459.  Confirmation Flag is: 1
```

```matlab
rng(r); % Return the random seed to its previous setting
```

Notice how the track score increases with every successful update and decreases with every missed detection. Once the track score is higher than the ConfirmationThreshold, the checkConfirmation function returns true, meaning that the track is confirmed now.

As in the history-based track logic, if the track is not assigned to any detection, it should eventually be deleted. The `DeletionThreshold` property is used to determine when the track score has decreased enough so that the track should be deleted. Note that the score does not have to fall below the `DeletionThreshold`, only that the *difference* between the current score and the maximum score obtained by the track should go below the threshold. The reason is that if a track obtains a high score after many successful updates, and if we used the absolute score, it would take too many missed updates to delete the track. The following shows a track being deleted after three misses.

```matlab
numSteps2 = 3;
scores(end+1:end+numSteps2,:) = zeros(numSteps2,2);
for i = 1:numSteps2
    miss(scoreLogic);
    deleteFlag = checkDeletion(scoreLogic);
    scores(numSteps1+i,:) = output(scoreLogic);
    disp(['Score and MaxScore: ', num2str(scores(numSteps1+i,:)),'.  Deletion Flag is: ',num2str
end
```

```
Score and MaxScore: 24.6407      29.2459.  Deletion Flag is: 0
Score and MaxScore: 22.3381      29.2459.  Deletion Flag is: 1
Score and MaxScore: 20.0355      29.2459.  Deletion Flag is: 1
```

```matlab
deletionScore = scores(:,2) + scoreLogic.DeletionThreshold;

stairs(scores)
hold on
plot([1,numSteps1+numSteps2],[scoreLogic.ConfirmationThreshold scoreLogic.ConfirmationThreshold]
stairs(deletionScore,'--')
title('Track Score and Maximum Score')
legend('Score','MaxScore','ConfirmationThrehsold','DeletionThreshold','Location','best')
```
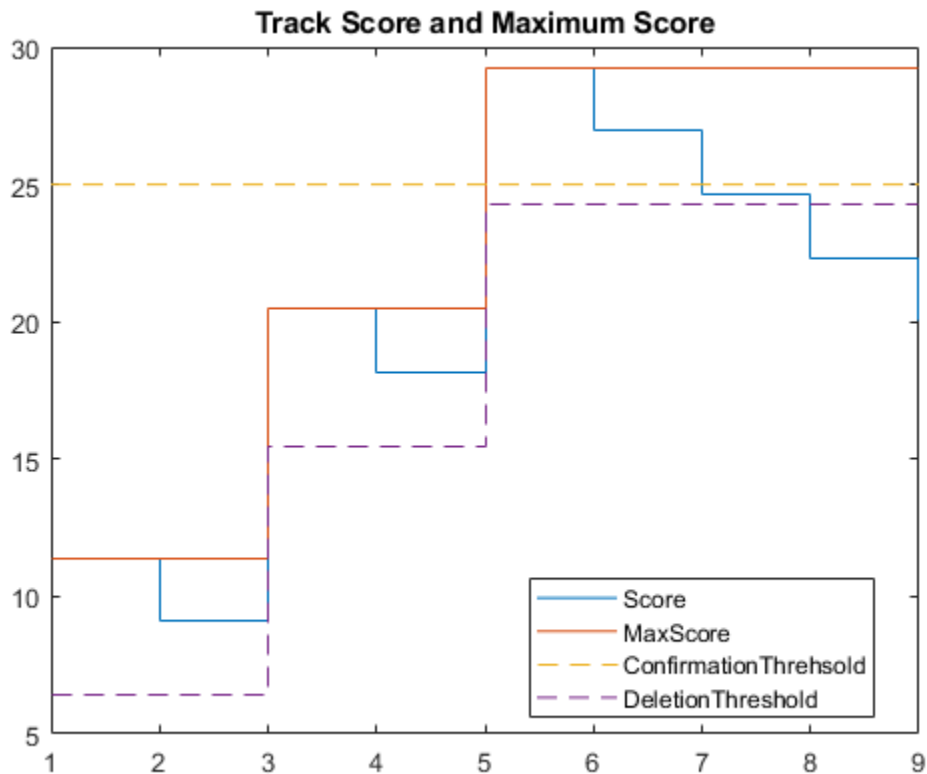
If you want to delay the track deletion until more misses happen, simply change the
`DeletionThreshold` to a more negative value.

Use a Track Logic in a Tracker

Typically, track logic is used inside a tracker. Configure the `trackerGNN` to use history-based track
logic with confirmation after 3 successes in 5 updates and deletion after 5 misses in 6 updates.

```
tracker = trackerGNN('Assignment', 'Auction', 'ConfirmationThreshold', [3 5], 'DeletionThreshold
tracker =
  trackerGNN with properties:

                TrackerIndex: 0
      FilterInitializationFcn: 'initcvekf'
                  Assignment: 'Auction'
         AssignmentThreshold: [30 Inf]
               MaxNumTracks: 100
              MaxNumSensors: 20

                  TrackLogic: 'History'
        ConfirmationThreshold: [3 5]
           DeletionThreshold: [5 6]

          HasCostMatrixInput: false
   HasDetectableTrackIDsInput: false
             StateParameters: [1x1 struct]
```

```
              NumTracks: 0
     NumConfirmedTracks: 0
```

The following loop updates the tracker with a detection at every odd numbered update. After 5 updates, the tracker confirms the track. You can look at the track history and the track confirmation flag after each update by examining the `TrackLogicState` and `IsConfirmed` fields of the track output.

```
for i = 1:5
    detectedFlag = logical(mod(i,2)); % Only odd updates are true
    if detectedFlag
        detection = {objectDetection(i,[1;2;3])};
    else
        detection = {};
    end
    [~,~,allTracks] = tracker(detection, i)
end

allTracks =
  objectTrack with properties:

             TrackID: 1
            BranchID: 0
         SourceIndex: 0
          UpdateTime: 1
                 Age: 1
               State: [6x1 double]
      StateCovariance: [6x6 double]
      StateParameters: [1x1 struct]
        ObjectClassID: 0
           TrackLogic: 'History'
      TrackLogicState: [1 0 0 0 0 0]
          IsConfirmed: 0
            IsCoasted: 0
       IsSelfReported: 1
      ObjectAttributes: [1x1 struct]


allTracks =
  objectTrack with properties:

             TrackID: 1
            BranchID: 0
         SourceIndex: 0
          UpdateTime: 2
                 Age: 2
               State: [6x1 double]
      StateCovariance: [6x6 double]
      StateParameters: [1x1 struct]
        ObjectClassID: 0
           TrackLogic: 'History'
      TrackLogicState: [0 1 0 0 0 0]
          IsConfirmed: 0
            IsCoasted: 1
       IsSelfReported: 1
      ObjectAttributes: [1x1 struct]
```

```
allTracks =
  objectTrack with properties:

              TrackID: 1
             BranchID: 0
          SourceIndex: 0
           UpdateTime: 3
                  Age: 3
                State: [6x1 double]
      StateCovariance: [6x6 double]
      StateParameters: [1x1 struct]
        ObjectClassID: 0
           TrackLogic: 'History'
      TrackLogicState: [1 0 1 0 0 0]
          IsConfirmed: 0
            IsCoasted: 0
       IsSelfReported: 1
     ObjectAttributes: [1x1 struct]


allTracks =
  objectTrack with properties:

              TrackID: 1
             BranchID: 0
          SourceIndex: 0
           UpdateTime: 4
                  Age: 4
                State: [6x1 double]
      StateCovariance: [6x6 double]
      StateParameters: [1x1 struct]
        ObjectClassID: 0
           TrackLogic: 'History'
      TrackLogicState: [0 1 0 1 0 0]
          IsConfirmed: 0
            IsCoasted: 1
       IsSelfReported: 1
     ObjectAttributes: [1x1 struct]


allTracks =
  objectTrack with properties:

              TrackID: 1
             BranchID: 0
          SourceIndex: 0
           UpdateTime: 5
                  Age: 5
                State: [6x1 double]
      StateCovariance: [6x6 double]
      StateParameters: [1x1 struct]
        ObjectClassID: 0
           TrackLogic: 'History'
      TrackLogicState: [1 0 1 0 1 0]
          IsConfirmed: 1
            IsCoasted: 0
       IsSelfReported: 1
```

```
        ObjectAttributes: [1x1 struct]
```

The following loop updates the tracker 5 more times with 5 misses. The track is deleted after 4 additional updates (the 9th update overall) because it has missed detections in the 4th, 6th, 7th, 8th and 9th updates (5 out of the last 6 updates).

```matlab
for i = 1:5
    detection = {};
    confirmedTrack = tracker(detection, i+5)
end

confirmedTrack =
  objectTrack with properties:

            TrackID: 1
           BranchID: 0
        SourceIndex: 0
         UpdateTime: 6
                Age: 6
              State: [6x1 double]
     StateCovariance: [6x6 double]
     StateParameters: [1x1 struct]
       ObjectClassID: 0
          TrackLogic: 'History'
      TrackLogicState: [0 1 0 1 0 1]
         IsConfirmed: 1
           IsCoasted: 1
       IsSelfReported: 1
      ObjectAttributes: [1x1 struct]


confirmedTrack =
  objectTrack with properties:

            TrackID: 1
           BranchID: 0
        SourceIndex: 0
         UpdateTime: 7
                Age: 7
              State: [6x1 double]
     StateCovariance: [6x6 double]
     StateParameters: [1x1 struct]
       ObjectClassID: 0
          TrackLogic: 'History'
      TrackLogicState: [0 0 1 0 1 0]
         IsConfirmed: 1
           IsCoasted: 1
       IsSelfReported: 1
      ObjectAttributes: [1x1 struct]


confirmedTrack =
  objectTrack with properties:

            TrackID: 1
           BranchID: 0
        SourceIndex: 0
```

```
             UpdateTime: 8
                    Age: 8
                  State: [6x1 double]
        StateCovariance: [6x6 double]
        StateParameters: [1x1 struct]
          ObjectClassID: 0
             TrackLogic: 'History'
        TrackLogicState: [0 0 0 1 0 1]
             IsConfirmed: 1
               IsCoasted: 1
          IsSelfReported: 1
        ObjectAttributes: [1x1 struct]


confirmedTrack =

  0x1 objectTrack array with properties:

    TrackID
    BranchID
    SourceIndex
    UpdateTime
    Age
    State
    StateCovariance
    StateParameters
    ObjectClassID
    TrackLogic
    TrackLogicState
    IsConfirmed
    IsCoasted
    IsSelfReported
    ObjectAttributes


confirmedTrack =

  0x1 objectTrack array with properties:

    TrackID
    BranchID
    SourceIndex
    UpdateTime
    Age
    State
    StateCovariance
    StateParameters
    ObjectClassID
    TrackLogic
    TrackLogicState
    IsConfirmed
    IsCoasted
    IsSelfReported
    ObjectAttributes
```

Summary

Trackers require a way to confirm tracks that are considered to be of true targets and delete tracks after they are not assigned any detections after a while. Two types of track confirmation and deletion logic were presented: a history-based track logic and a score-based track logic. The history-based track logic only considers whether a track is assigned detections in the recent past updates. The score-based track logic provides a statistical measure of how likely a track is to represent a real target. Both types of track logic objects can be used as stand-alone objects, but are typically used inside a tracker object.

Introduction to Tracking Scenario and Simulating Sensor Detections

This example introduces how to generate synthetic radar detections in a tracking scenario that simulates target motion and sensor detections. Specifically, this example shows:

- How to simulate the motion of targets using `trackingScenario`.
- How to generate synthetic sensor detections using a sensor object. You use three different approaches to generate detections — generate detections from a specific sensor, generate detections for all sensors mounted on a given platform, and generate all detections in the tracking scenario.
- How to use the various plotters offered by `theaterPlot` to visualize the scenario, sensor coverage, and detections.

The main benefit of using scenario generation and sensor simulation over sensor recording is the ability to create rare and potentially hazardous scenarios and test sensor fusion algorithms with these scenarios. These radar detections can be used to develop various tracking algorithms, such as `trackerGNN` and `trackerJPDA`.

Simulate a Moving Object and its Attributes

Create a Tracking Scenario with a Moving Target

The first step in generating simulated radar detections is creating a tracking scenario, in which the motion of one or more moving targets is simulated. To set up a tracking scenario, you first create a `trackingScenario` object, which serves as a simulation environment for all the moving targets added later.

```
rng(2020); % For repeatable results.
scene = trackingScenario('UpdateRate',2.5)

scene =
  trackingScenario with properties:

    IsEarthCentered: 0
         UpdateRate: 2.5000
     SimulationTime: 0
           StopTime: Inf
          IsRunning: 1
          Platforms: {}
```

By default, a tracking scenario runs from the `SimulationTime` to the `StopTime`. The step size is given by `UpdateRate` in Hz. Based on an update rate of 2.5 Hz, the step size of `scene` is 0.4 sec. Currently, there is no targets (defined using `platform`) defined in the scenario. When the `StopTime` is defined as `Inf`, the scenario runs until the motion of all the platforms in the scenario ends. Define a platform in the scenario named `target`:

```
target = platform(scene);
```

You can check the target is now defined in the scenario, `scene`.

```
scene.Platforms{1}
```

```
ans =
  Platform with properties:

        PlatformID: 1
           ClassID: 0
          Position: [0 0 0]
       Orientation: [0 0 0]
        Dimensions: [1×1 struct]
              Mesh: [1×1 extendedObjectMesh]
        Trajectory: [1×1 kinematicTrajectory]
     PoseEstimator: [1×1 insSensor]
          Emitters: {}
           Sensors: {}
        Signatures: {[1×1 rcsSignature]  [1×1 irSignature]  [1×1 tsSignature]}
```

By default, a dimensionless point object is created, but you can specify the object extent in length, width and height. Also, a trivial `kinematicTrajectory` is associated with the target. To create a trajectory for the target, you can use the `waypointTrajectory` object. You can set up a waypoint trajectory by specifying a series of waypoints of the target and the corresponding visiting time. You can also specify other trajectory properties such as velocity and course. See `waypointTrajectory` for more details.

The following code defines a racetrack path for a target with straight legs of 20 km and a turn radius of 2 km. The altitude of the trajectory is 3 km, which is defined as –3 km in the default North-East-Down coordinate frame used in this scenario.

```
h = -3; % Unit in km
waypoints = 1e3*[  1    1    0   -1   -1    0    1    1
                   0   10   12   10  -10  -12  -10    0
                   h    h    h    h    h    h    h    h]';
course = [ 90   90  180   270  270    0   90   90]';% unit in degree
timeOfArrival = 60*[  0   1.5  1.8  2.1  5.1  5.4  5.7  7.2]';
targetTrajectory = waypointTrajectory('Waypoints', waypoints, 'TimeOfArrival', timeOfArrival);
target.Trajectory = targetTrajectory;
```

Visualize and Run the Tracking Scenario

To visualize the scenario as it runs, create a `theaterPlot` and add a `trajectoryPlotter` and a `platformPlotter` to visualize the trajectory and the target, respectively.

```
tp = theaterPlot('XLimits',[-12 12]*1e3,'Ylimits',[-12 12]*1e3,'ZLimits',[-1e4 1e4]);
trajPlotter = trajectoryPlotter(tp,'DisplayName','Trajectory');
plotTrajectory(trajPlotter,{waypoints})
targetPlotter = platformPlotter(tp,'DisplayName','Target');
```

Run the tracking scenario and visualize the target motion in a bird's eye view.

```
while advance(scene) && ishghandle(tp.Parent)
    targetPose = pose(target,'true');
    plotPlatform(targetPlotter, targetPose.Position);
end
```
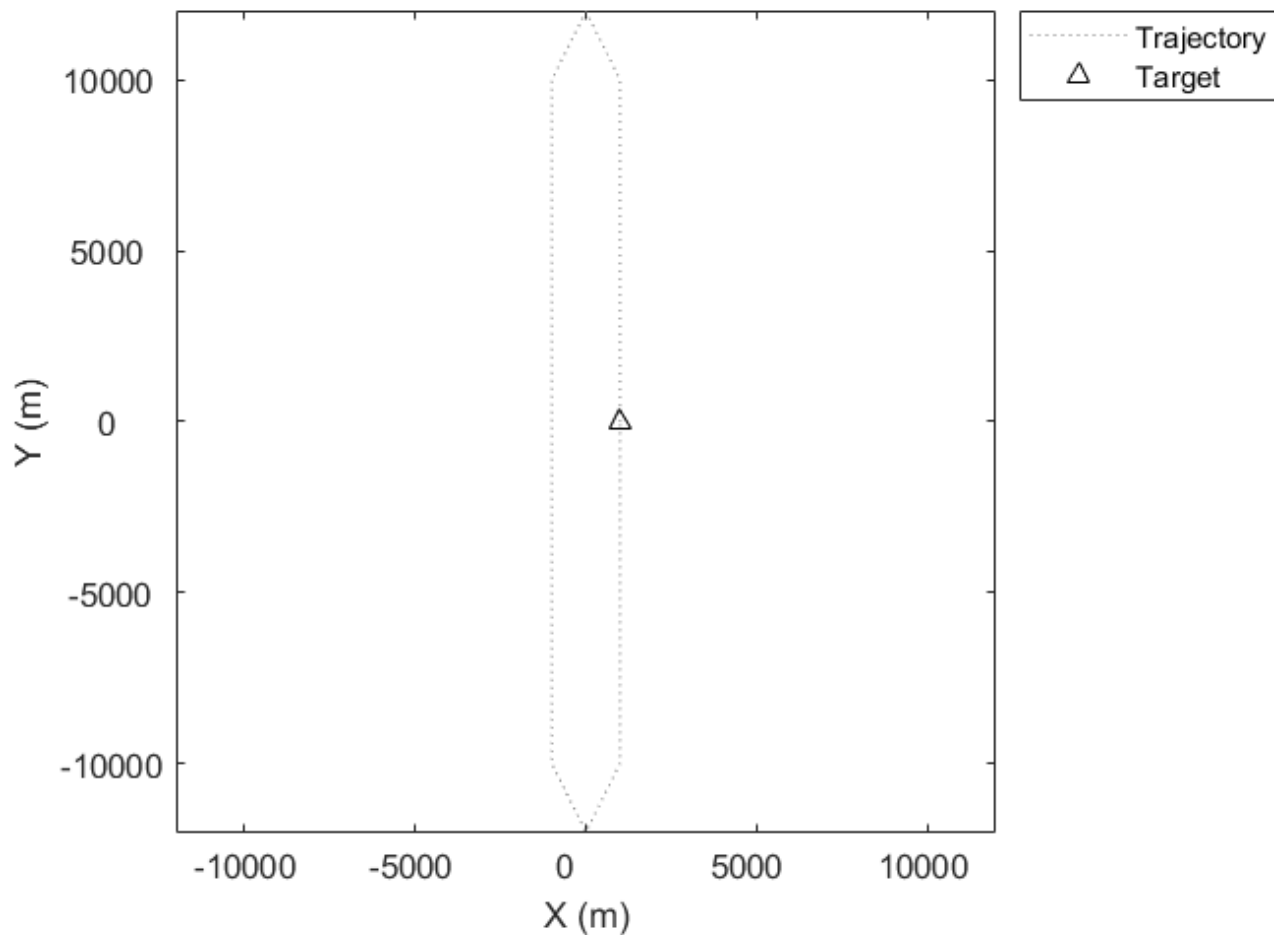
Generate Detections Using Radar Sensors

In a tracking scenario, you can generate detections of targets in three different approaches:

Model a Radar Sensor and Simulate its Detections

To add a sensor to the scenario, you first create a `Platform` on which the sensor is mounted. The `Platform` object also provides a `targetPoses` function which you can use to obtain the poses of targets relative to the platform.

Create a Tower and a Radar

Next, you create a tower platform with dimensions of 5 m, 5 m, and 30 m in length, width, and height, respectively. Position the tower to the left the target path. Set the z coordinate of the tower position to -15 m such that the tower bottom is on the horizontal plane of the scenario frame.

```
tower = platform(scene);
tower.Dimensions = struct ('Length',5,'Width',5,'Height',30,'OriginOffset', [0 0 0]);
tower.Trajectory.Position = [-1e4 0 -15];
```

Define a platform plotter to show the tower.

```
towerPlotter = platformPlotter(tp,'DisplayName','Tower','Marker','o','MarkerFaceColor',[0 0 0]);
plotPlatform(towerPlotter,tower.Trajectory.Position,tower.Dimensions,tower.Trajectory.Orientation
```

Create a monostatic scanning radar sensor. By default, a sensor object reports detections in the coordinate frame of the platform on which the sensor is mounted. You can change the output coordinate frame using the `DetectionsCoordinates` property of `mononstaticRadarSensor`. The available output frames are `'Body'`(default), `'Scenario'`, `'Sensor rectangular'`, and `'Sensor spherical'`.

```
radar1 = monostaticRadarSensor(1,'UpdateRate',2.5,...
    'MountingLocation',[0 0 -15],'FieldOfView',[4;45],...
    'MechanicalScanLimits',[-60 60;0 0],'HasElevation',true,'FalseAlarmRate',1e-7)

radar1 =
  monostaticRadarSensor with properties:

                 SensorIndex: 1
                  UpdateRate: 2.5000
                    ScanMode: 'Mechanical'

            MountingLocation: [0 0 -15]
              MountingAngles: [0 0 0]

          MaxUnambiguousRange: 100000
    MaxUnambiguousRadialSpeed: 200
                  FieldOfView: [2×1 double]
          MaxMechanicalScanRate: [2×1 double]
          MechanicalScanLimits: [2×2 double]
               MechanicalAngle: [2×1 double]
                    LookAngle: [2×1 double]

          DetectionProbability: 0.9000
                FalseAlarmRate: 1.0000e-07

  Show all properties
```

Mount the sensor on the tower platform.

```
tower.Sensors = radar1;
```

Generate and Simulate Detections

You can visualize the radar coverage area and its scanning beam using the `coveragePlotter`, `coverageConfig`, and `plotCoverage` functions. Create a `detectionPlotter` to visualize detections generated by the radar.

```
radar1Plotter = coveragePlotter(tp,'DisplayName','Radar1 Beam','Color','b');
detPlotter = detectionPlotter(tp,'DisplayName','Detections',...
    'Marker','o','MarkerFaceColor',[1 0 0]);
```

Restart the scenario, generate the detections, and plot the detections. You can observe several false detections.

```
restart(scene);
while advance(scene) && ishghandle(tp.Parent)

    view(70,40); % Comment this to show the bird's eye view;
```

5-89

```matlab
time = scene.SimulationTime;

% Obtain the target pose and plot it.
poseTarget = pose(target,'true');
plotPlatform(targetPlotter,poseTarget.Position);

% Obtain the target pose expressed in the tower's coordinate frame.
poseInTower = targetPoses(tower);

% Obtain the radar detections using the radar object.
[detections, numDets] = radar1(poseInTower,time);

% Extract detection positions and transform them to the scenario frame.
detPos = zeros(numDets,3);
for i=1:numDets
    detPos(i,:) = tower.Trajectory.Position + detections{i}.Measurement';
end

% Plot detections.
if ~isempty(detPos)
    plotDetection(detPlotter,detPos);
end

% Plot the radar beam and coverage area.
plotCoverage(radar1Plotter,coverageConfig(scene));

end
```
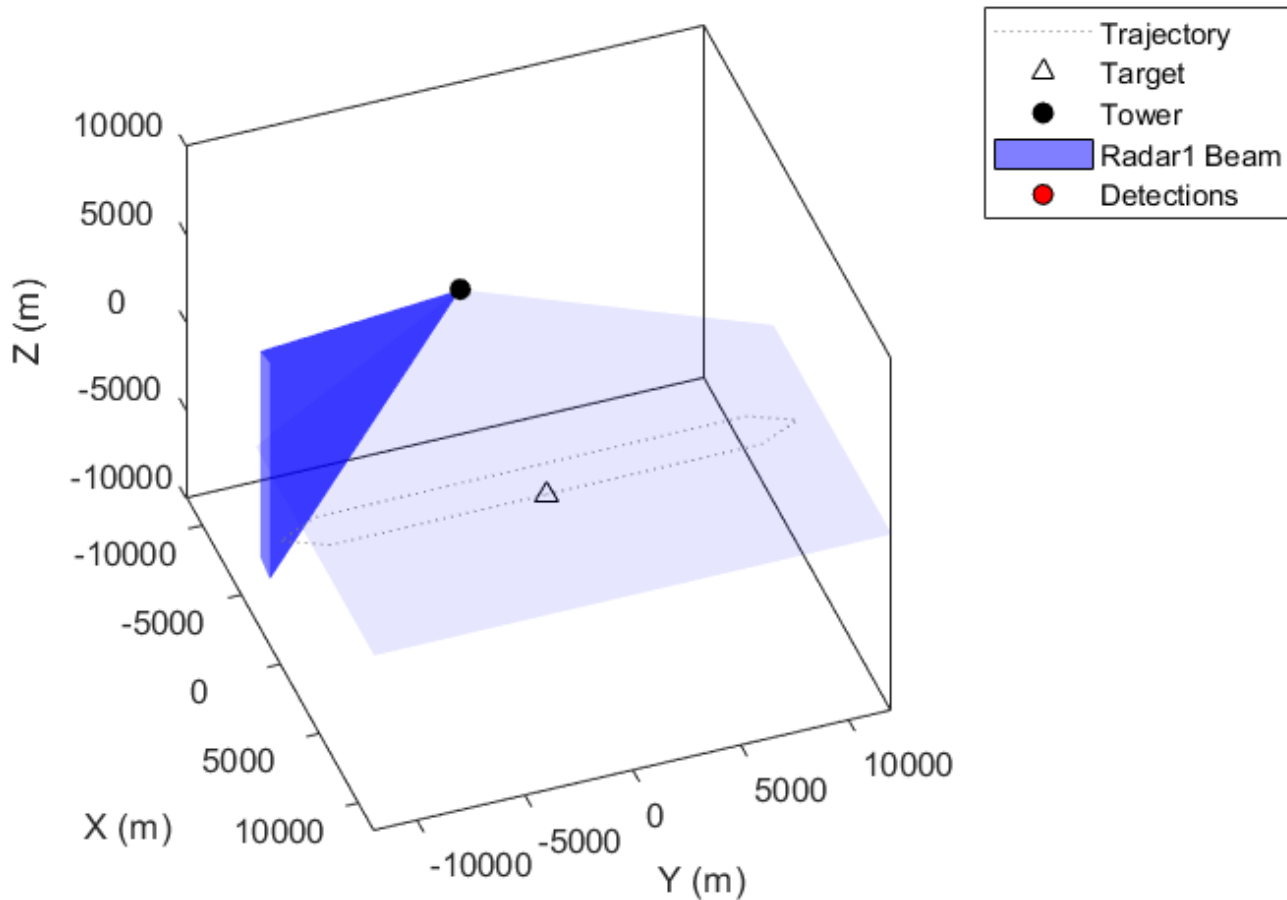
Generate Detections for All the Sensors Mounted on a Platform

Alternately, you can generate detections for all sensors on a given platform using the `detect` function. To illustrate this approach, you add a second radar sensor to the tower platform. You also create a coverage plotter for the second radar sensor.

```
radar2 = monostaticRadarSensor(2,'Rotator','UpdateRate',2.5, ...
    'MountingLocation',[0 0 -15], 'FieldOfView',[4; 45], ...
    'HasElevation',true,'FalseAlarmRate',1e-7);
tower.Sensors{2} = radar2;
radar2Plotter = coveragePlotter(tp,'DisplayName','Radar 2 Beam', 'Color','g');
```

Restart the scenario, generate the detections based on the tower, and plot the detections. As shown in the simulation, both the two radars generate detections of the target.

```
restart(scene);
while advance(scene) && ishghandle(tp.Parent)

    time = scene.SimulationTime;

    % Obtain the target pose and plot it.
```

```
poseTarget = pose(target,'true');
plotPlatform(targetPlotter,poseTarget.Position);

% Plot the coverage of sensors.
configs = coverageConfig(scene);
plotCoverage(radar1Plotter,configs(1));
plotCoverage(radar2Plotter,configs(2));

% Generate the detections.
[towerDetections, numDets] = detect(tower,time);

% Extract detection positions and transform them to the scenario frame.
detPos = NaN(numDets,3);
for i=1:numDets
    detPos(i,:) = tower.Trajectory.Position + towerDetections{i}.Measurement';
end

% Plot detections.
if numDets
    plotDetection(detPlotter,detPos);
end

end
```
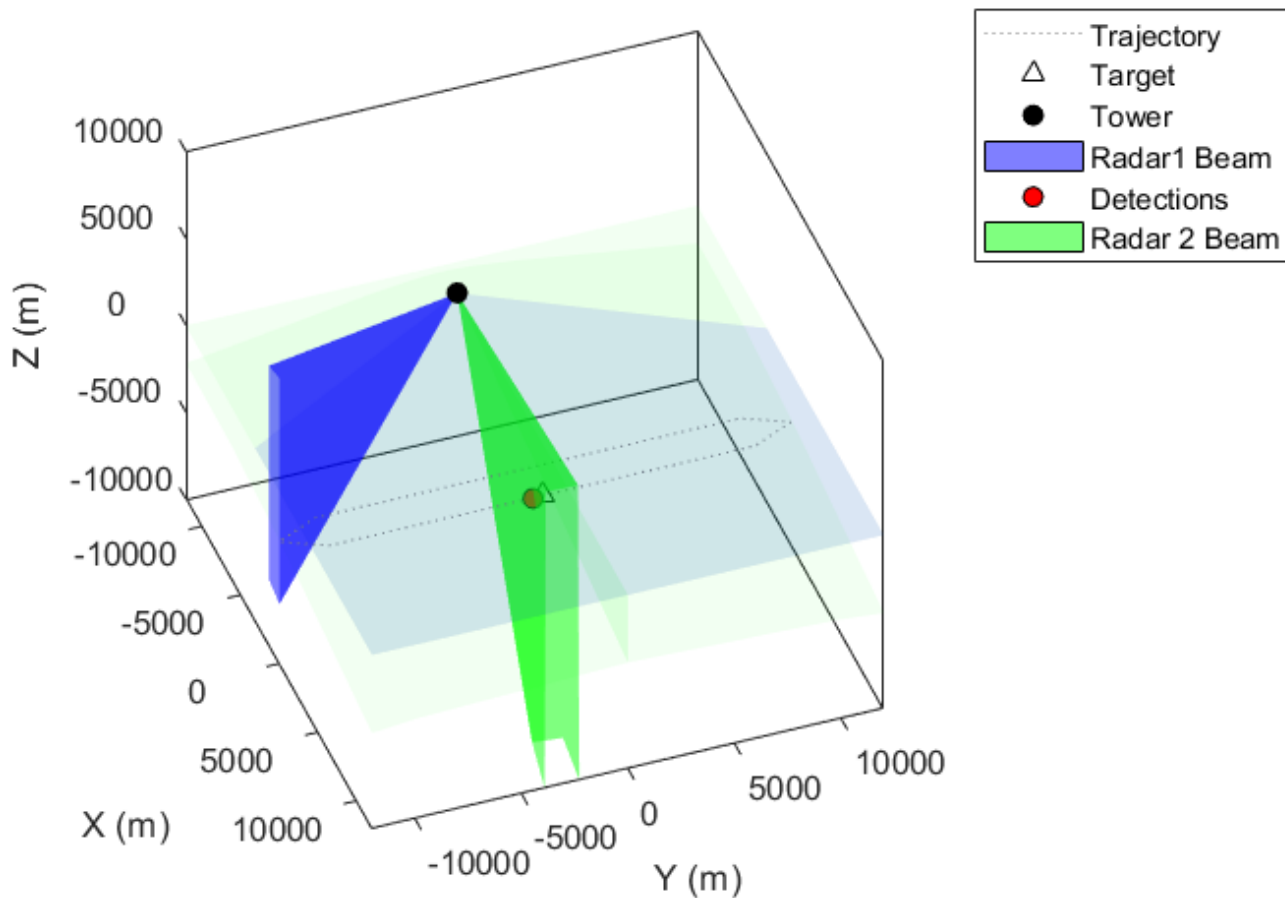
Generate Detections from All the Sensors in a Scenario

You can also generate detections from all sensors in the tracking scenario using the `detect` function of `trackingScenario`.

To illustrate this approach, you add a plane in the scenario and define its waypoint trajectory. The plane flies from southwest to northeast at a height of 2.9 km. Create a platform plotter for the plane.

```
plane = platform(scene);
planeTrajectory = waypointTrajectory('Waypoints',1e3*[-10 -10 -2.9; 12 12 -2.9],'TimeOfArrival',
plane.Trajectory = planeTrajectory;
planePlotter = platformPlotter(tp,'DisplayName','Plane','Marker','d','MarkerEdgeColor','k');
```

Mount a staring radar on the plane with a field of view of 50 degrees. Create a plotter for the radar.

```
radar3 = monostaticRadarSensor(3,'No scanning','UpdateRate',2.5,'FieldOfView',[60,20], ...
    'HasElevation',true,'FalseAlarmRate',1e-7);
plane.Sensors = radar3;
radar3Plotter = coveragePlotter(tp,'DisplayName','Radar 3 Beam', 'Color','y');
```

Set the reporting frames of all the three radars to the scenario frame. You need to enable inertial navigation system (INS) before setting the detection coordinate to the scenario frame.

```
release(radar1);
radar1.HasINS = true;
radar1.DetectionCoordinates = 'Scenario';
release(radar2);
radar2.HasINS = true;
radar2.DetectionCoordinates = 'Scenario';
radar3.HasINS = true;
radar3.DetectionCoordinates = 'Scenario';
```

Restart and run the scenario, generate all the detections in the scenario, and plot the detections.

```
% Hide the published figure. Show a pre-recorded animation instead.
f = tp.Parent.Parent;
if numel(dbstack) > 5
    f.Visible = 'on';
    f.Visible = 'off';
end

restart(scene);
while advance(scene) && ishghandle(tp.Parent)

    % Obtain the target pose and plot it.
    poseTarget = pose(target,'true');
    plotPlatform(targetPlotter,poseTarget.Position);

    % Obtain the plane pose and plot it.
    posePlane = pose(plane,'true');
    plotPlatform(planePlotter,posePlane.Position);

    % Plot the coverage of sensors.
    configs = coverageConfig(scene);
    plotCoverage(radar1Plotter,configs(1));
    plotCoverage(radar2Plotter,configs(2));
    plotCoverage(radar3Plotter,configs(3));

    % Generate the detections.
    scenarioDetections = detect(scene);
    numDets = numel(scenarioDetections);

    % Extract detection positions in the scenario frame.
    detPos = NaN(numDets,3);
    for i=1:numDets
        detPos(i,:) = scenarioDetections{i}.Measurement';
    end

    % Plot detections.
    if numDets
        plotDetection(detPlotter,detPos);
    end

end
```
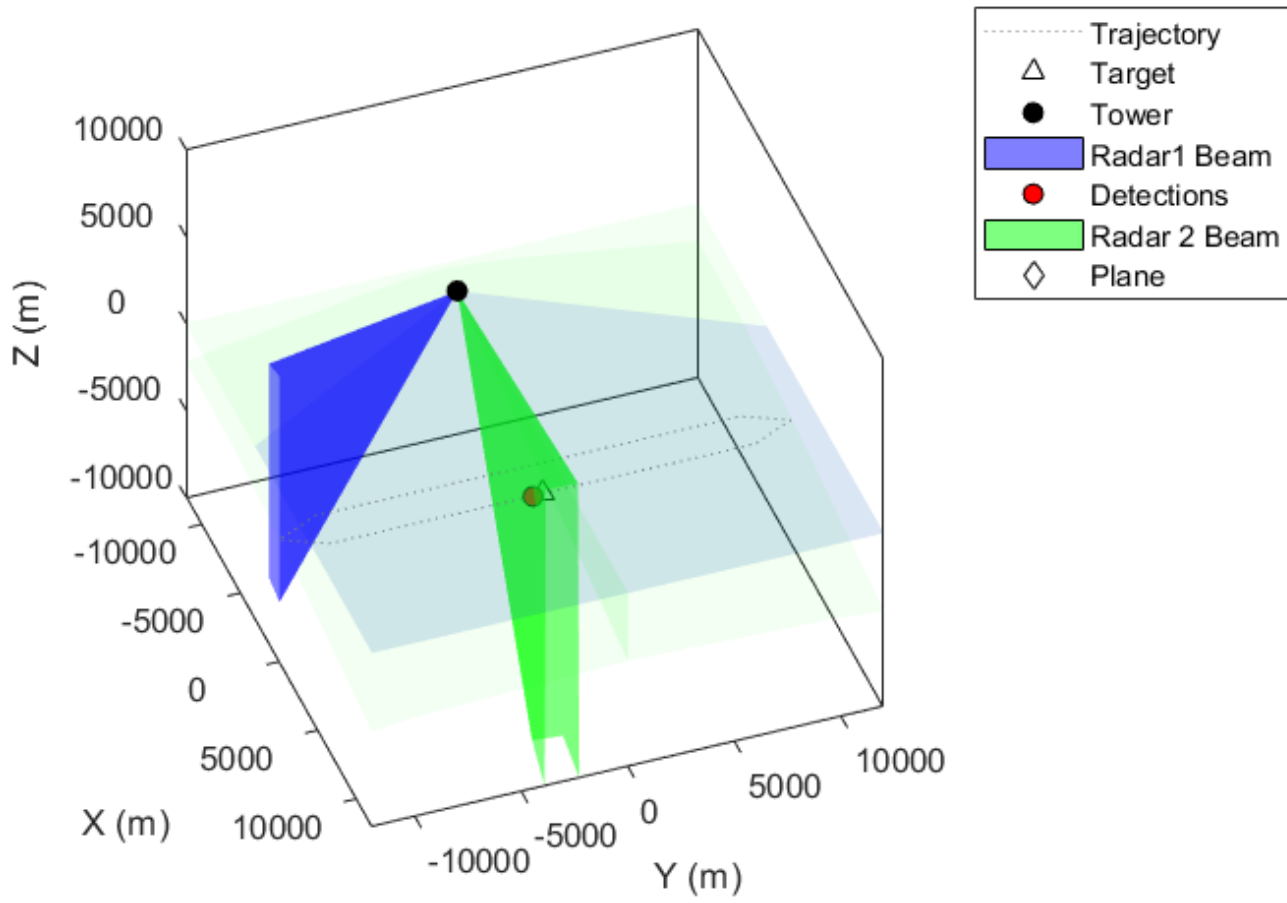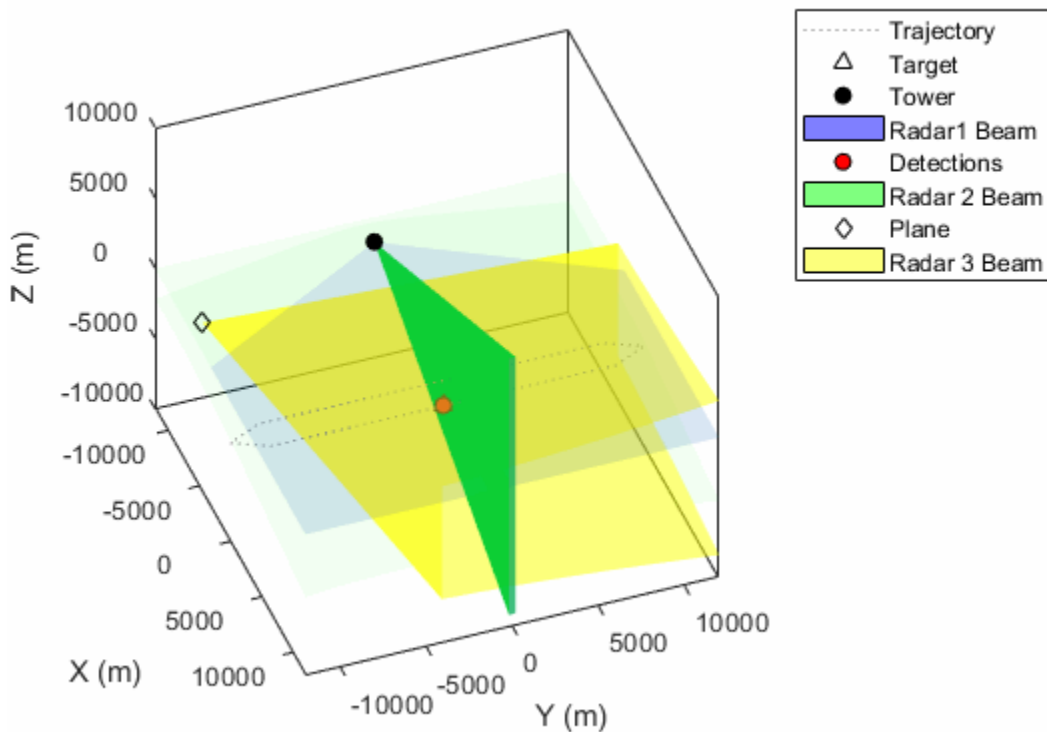
From the results, for a period of time, the radar on the plane can continuously detect the target. The radar on the plane also generate several false detections. The two radars on the tower can detect the plane.

Summary

This example shows you how to create a tracking scenario, simulate target motion, and simulate radar detections in three different approaches based on a sensor, a platform, and the whole scenario. It also shows you how to visualize the target trajectory, instantaneous positions, radar scan beams, and detections.

Inertial Sensor Noise Analysis Using Allan Variance

This example shows how to use the Allan variance to determine noise parameters of a MEMS gyroscope. These parameters can be used to model the gyroscope in simulation. The gyroscope measurement is modeled as:

$$\Omega(t) = \Omega_{Ideal}(t) + Bias_N(t) + Bias_B(t) + Bias_K(t)$$

The three noise parameters N (angle random walk), K (rate random walk), and B (bias instability) are estimated using data logged from a stationary gyroscope.

Background

Allan variance was originally developed by David W. Allan to measure the frequency stability of precision oscillators. It can also be used to identify various noise sources present in stationary gyroscope measurements. Consider L samples of data from a gyroscope with a sample time of τ_0. Form data clusters of durations $\tau_0, 2\tau_0, ..., m\tau_0, (m < (L-1)/2)$ and obtain the averages of the sum of the data points contained in each cluster over the length of the cluster. The Allan variance is defined as the two-sample variance of the data cluster averages as a function of cluster time. This example uses the overlapping Allan variance estimator. This means that the calculated clusters are overlapping. The estimator performs better than non-overlapping estimators for larger values of L.

Allan Variance Calculation

The Allan variance is calculated as follows:

Log L stationary gyroscope samples with a sample period τ_0. Let Ω be the logged samples.

```
% Load logged data from one axis of a three-axis gyroscope. This recording
% was done over a six hour period with a 100 Hz sampling rate.
load('LoggedSingleAxisGyroscope', 'omega', 'Fs')
t0 = 1/Fs;
```

For each sample, calculate the output angle θ:

$$\theta(t) = \int^t \Omega(t')dt'$$

For discrete samples, the cumulative sum multiplied by τ_0 can be used.

```
theta = cumsum(omega, 1)*t0;
```

Next, calculate the Allan variance:

$$\sigma^2(\tau) = \frac{1}{2\tau^2} < (\theta_{k+2m} - 2\theta_{k+m} + \theta_k)^2 >$$

where $\tau = m\tau_0$ and $<>$ is the ensemble average.

The ensemble average can be expanded to:

$$\sigma^2(\tau) = \frac{1}{2\tau^2(L - 2m)} \sum_{k=1}^{L-2m} (\theta_{k+2m} - 2\theta_{k+m} + \theta_k)^2$$

5-97

```
maxNumM = 100;
L = size(theta, 1);
maxM = 2.^floor(log2(L/2));
m = logspace(log10(1), log10(maxM), maxNumM).';
m = ceil(m); % m must be an integer.
m = unique(m); % Remove duplicates.

tau = m*t0;

avar = zeros(numel(m), 1);
for i = 1:numel(m)
    mi = m(i);
    avar(i,:) = sum( ...
        (theta(1+2*mi:L) - 2*theta(1+mi:L-mi) + theta(1:L-2*mi)).^2, 1);
end
avar = avar ./ (2*tau.^2 .* (L - 2*m));
```

Finally, the Allan deviation $\sigma(t) = \sqrt{\sigma^2(t)}$ is used to determine the gyroscope noise parameters.

```
adev = sqrt(avar);

figure
loglog(tau, adev)
title('Allan Deviation')
xlabel('\tau');
ylabel('\sigma(\tau)')
grid on
axis equal
```
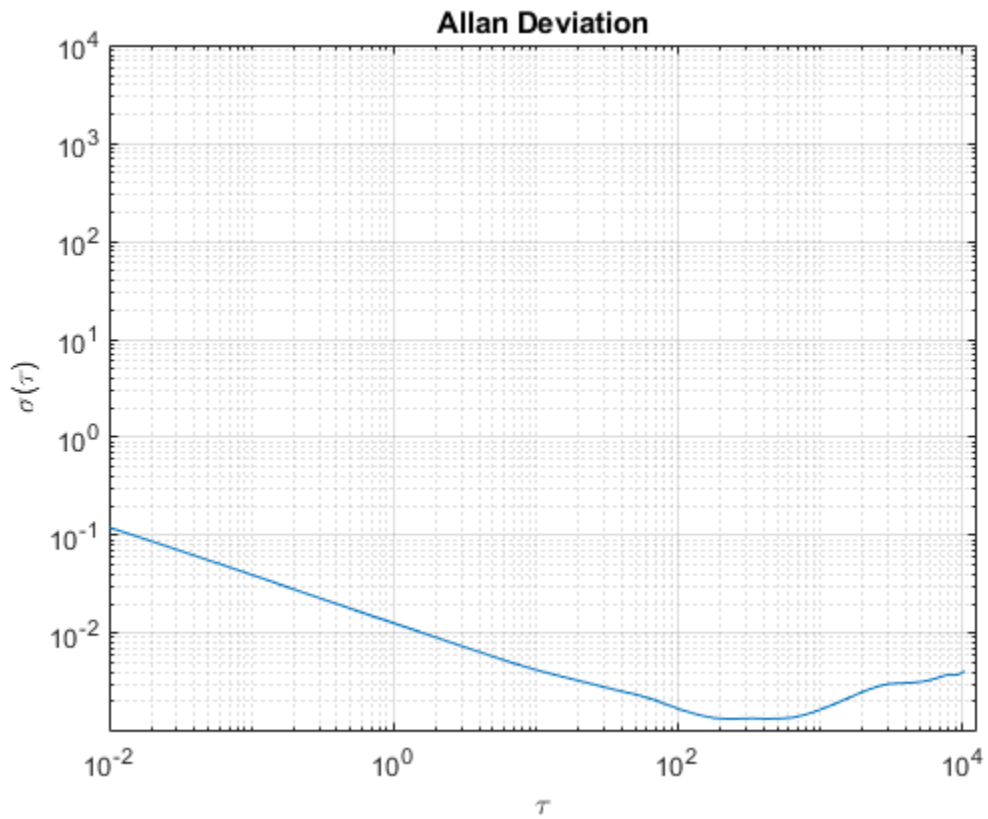
Allan Deviation

The Allan variance can also be calculated using the `allanvar` function.

```
[avarFromFunc, tauFromFunc] = allanvar(omega, m, Fs);
adevFromFunc = sqrt(avarFromFunc);

figure
loglog(tau, adev, tauFromFunc, adevFromFunc);
title('Allan Deviations')
xlabel('\tau')
ylabel('\sigma(\tau)')
legend('Manual Calculation', 'allanvar Function')
grid on
axis equal
```

Allan Deviations
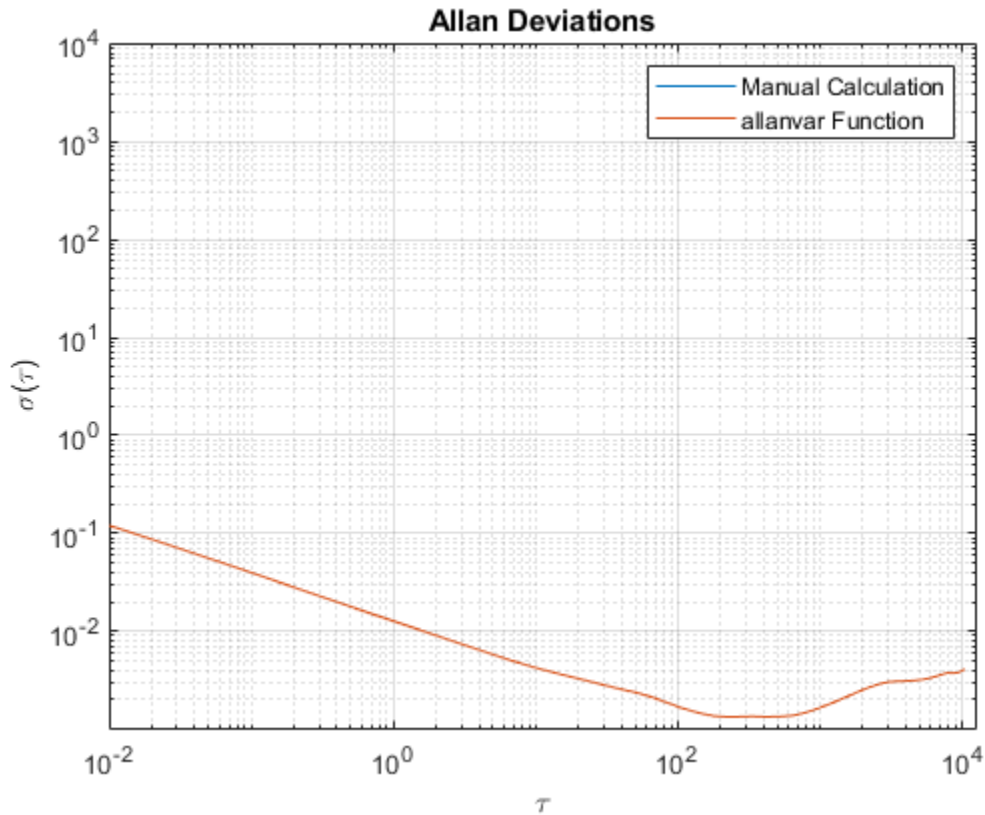
Noise Parameter Identification

To obtain the noise parameters for the gyroscope, use the following relationship between the Allan variance and the two-sided power spectral density (PSD) of the noise parameters in the original data set Ω. The relationship is:

$$\sigma^2(\tau) = 4 \int_0^\infty S_\Omega(f) \frac{sin^4(\pi f\tau)}{(\pi f\tau)^2} df$$

From the above equation, the Allan variance is proportional to the total noise power of the gyroscope when passed through a filter with a transfer function of $sin^4(x)/(x)^2$. This transfer function arises from the operations done to create and operate on the clusters.

Using this transfer function interpretation, the filter bandpass depends on τ. This means that different noise parameters can be identified by changing the filter bandpass, or varying τ.

Angle Random Walk

The angle random walk is characterized by the white noise spectrum of the gyroscope output. The PSD is represented by:

$$S_\Omega(f) = N^2$$

where

N = angle random walk coefficient

Substituting into the original PSD equation and performing integration yields:

$$\sigma^2(\tau) = \frac{N^2}{\tau}$$

The above equation is a line with a slope of -1/2 when plotted on a log-log plot of $\sigma(\tau)$ versus τ. The value of N can be read directly off of this line at $\tau = 1$.

```
% Find the index where the slope of the log-scaled Allan deviation is equal
% to the slope specified.
slope = -0.5;
logtau = log10(tau);
logadev = log10(adev);
dlogadev = diff(logadev) ./ diff(logtau);
[~, i] = min(abs(dlogadev - slope));

% Find the y-intercept of the line.
b = logadev(i) - slope*logtau(i);

% Determine the angle random walk coefficient from the line.
logN = slope*log(1) + b;
N = 10^logN

% Plot the results.
tauN = 1;
lineN = N ./ sqrt(tau);
figure
loglog(tau, adev, tau, lineN, '--', tauN, N, 'o')
title('Allan Deviation with Angle Random Walk')
xlabel('\tau')
ylabel('\sigma(\tau)')
legend('\sigma', '\sigma_N')
text(tauN, N, 'N')
grid on
axis equal


N =

    0.0126
```
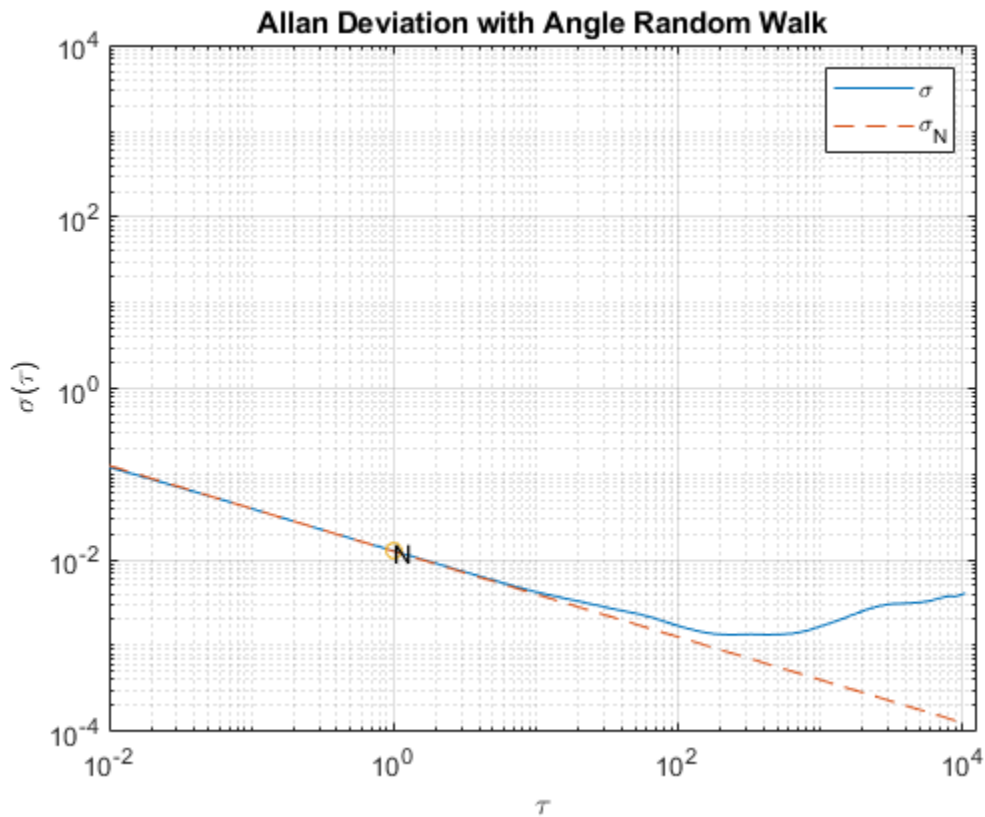
Rate Random Walk

The rate random walk is characterized by the red noise (Brownian noise) spectrum of the gyroscope output. The PSD is represented by:

$$S_\Omega(f) = (\frac{K}{2\pi})^2 \frac{1}{f^2}$$

where

K = rate random walk coefficient

Substituting into the original PSD equation and performing integration yields:

$$\sigma^2(\tau) = \frac{K^2 \tau}{3}$$

The above equation is a line with a slope of 1/2 when plotted on a log-log plot of $\sigma(\tau)$ versus τ. The value of K can be read directly off of this line at $\tau = 3$.

```
% Find the index where the slope of the log-scaled Allan deviation is equal
% to the slope specified.
slope = 0.5;
logtau = log10(tau);
logadev = log10(adev);
dlogadev = diff(logadev) ./ diff(logtau);
```

```matlab
[~, i] = min(abs(dlogadev - slope));

% Find the y-intercept of the line.
b = logadev(i) - slope*logtau(i);

% Determine the rate random walk coefficient from the line.
logK = slope*log10(3) + b;
K = 10^logK

% Plot the results.
tauK = 3;
lineK = K .* sqrt(tau/3);
figure
loglog(tau, adev, tau, lineK, '--', tauK, K, 'o')
title('Allan Deviation with Rate Random Walk')
xlabel('\tau')
ylabel('\sigma(\tau)')
legend('\sigma', '\sigma_K')
text(tauK, K, 'K')
grid on
axis equal
```

```
K =

   9.0679e-05
```


Allan Deviation with Rate Random Walk
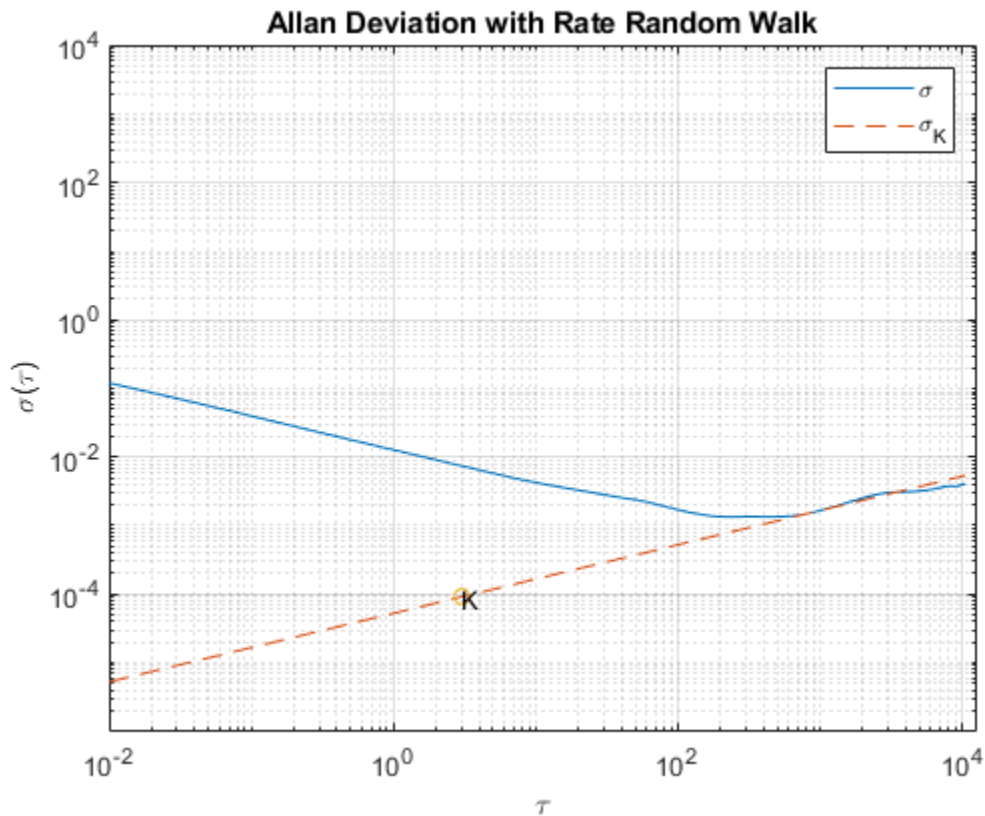
Bias Instability

The bias instability is characterized by the pink noise (flicker noise) spectrum of the gyroscope output. The PSD is represented by:

$$S_\Omega(f) = \begin{cases} (\frac{B^2}{2\pi})\frac{1}{f} & : f \le f_0 \\ 0 & : f > f_0 \end{cases}$$

where

B = bias instability coefficient

f_0 = cut-off frequency

Substituting into the original PSD equation and performing integration yields:

$$\sigma^2(\tau) = \frac{2B^2}{\pi}\left[\ln 2 + - \frac{\sin^3 x}{2x^2}(\sin x + 4x\cos x) + Ci(2x) - Ci(4x)\right]$$

where

$$x = \pi f_0 \tau$$

Ci = cosine-integral function

When τ is much longer than the inverse of the cutoff frequency, the PSD equation is:

$$\sigma^2(\tau) = \frac{2B^2}{\pi}\ln 2$$

The above equation is a line with a slope of 0 when plotted on a log-log plot of $\sigma(\tau)$ versus τ. The value of B can be read directly off of this line with a scaling of $\sqrt{\frac{2\ln 2}{\pi}} \approx 0.664$.

```
% Find the index where the slope of the log-scaled Allan deviation is equal
% to the slope specified.
slope = 0;
logtau = log10(tau);
logadev = log10(adev);
dlogadev = diff(logadev) ./ diff(logtau);
[~, i] = min(abs(dlogadev - slope));

% Find the y-intercept of the line.
b = logadev(i) - slope*logtau(i);

% Determine the bias instability coefficient from the line.
scfB = sqrt(2*log(2)/pi);
logB = b - log10(scfB);
B = 10^logB

% Plot the results.
tauB = tau(i);
lineB = B * scfB * ones(size(tau));
figure
loglog(tau, adev, tau, lineB, '--', tauB, scfB*B, 'o')
```

```matlab
title('Allan Deviation with Bias Instability')
xlabel('\tau')
ylabel('\sigma(\tau)')
legend('\sigma', '\sigma_B')
text(tauB, scfB*B, '0.664B')
grid on
axis equal
```

```
B =

    0.0020
```


Allan Deviation with Bias Instability
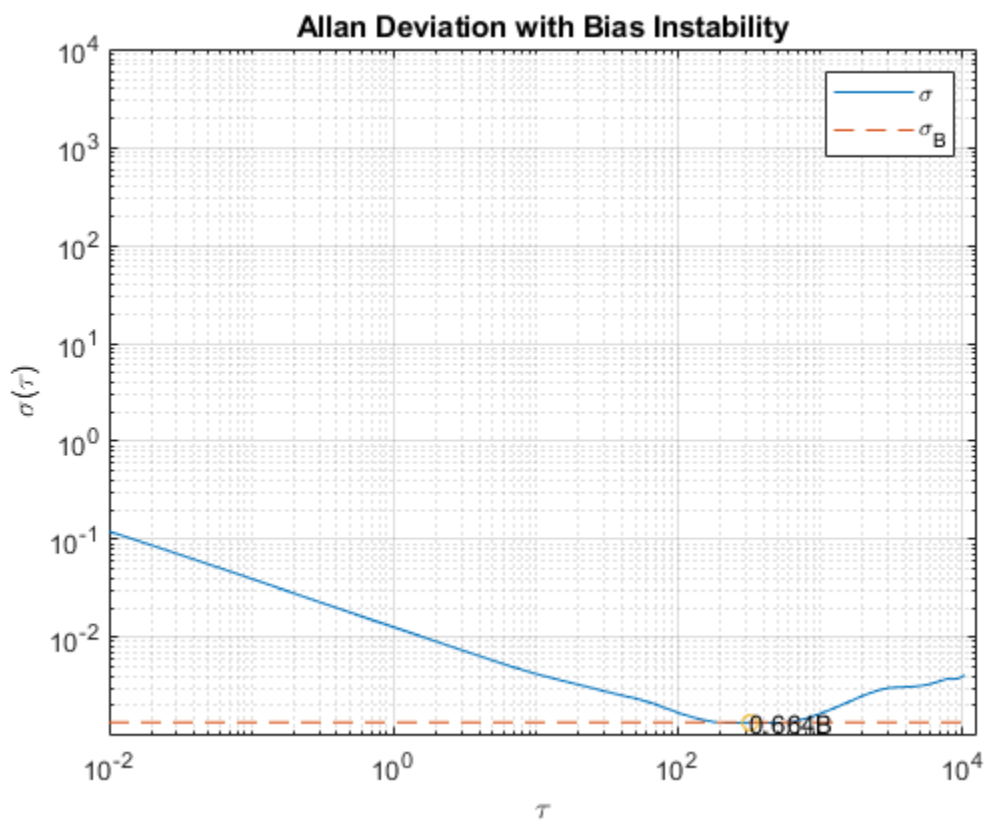
Now that all the noise parameters have been calculated, plot the Allan deviation with all of the lines used for quantifying the parameters.

```matlab
tauParams = [tauN, tauK, tauB];
params = [N, K, scfB*B];
figure
loglog(tau, adev, tau, [lineN, lineK, lineB], '--', ...
    tauParams, params, 'o')
title('Allan Deviation with Noise Parameters')
xlabel('\tau')
ylabel('\sigma(\tau)')
legend('\sigma', '\sigma_N', '\sigma_K', '\sigma_B')
text(tauParams, params, {'N', 'K', '0.664B'})
```

```
grid on
axis equal
```
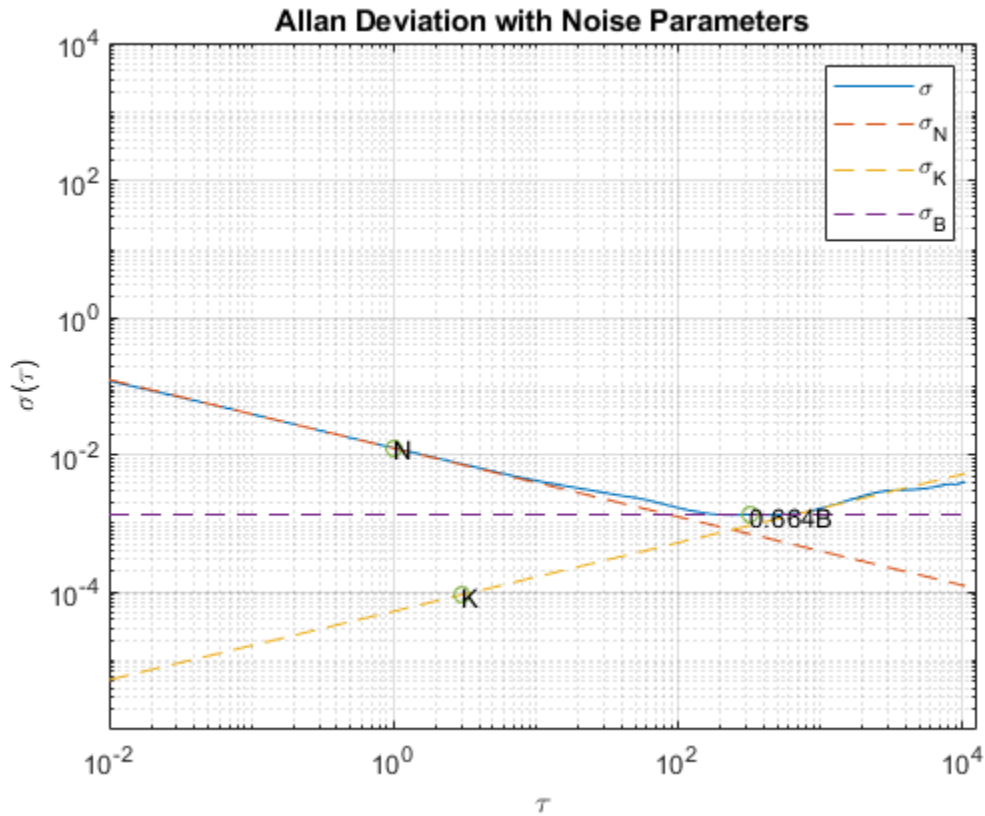

Gyroscope Simulation

Use the `imuSensor` object to simulate gyroscope measurements with the noise parameters identified above.

```
% Simulating the gyroscope measurements takes some time. To avoid this, the
% measurements were generated and saved to a MAT-file. By default, this
% example uses the MAT-file. To generate the measurements instead, change
% this logical variable to true.
generateSimulatedData = false;

if generateSimulatedData
    % Set the gyroscope parameters to the noise parameters determined
    % above.
    gyro = gyroparams('NoiseDensity', N, 'RandomWalk', K, ...
        'BiasInstability', B);
    omegaSim = helperAllanVarianceExample(L, Fs, gyro);
else
    load('SimulatedSingleAxisGyroscope', 'omegaSim')
end
```

Calculate the simulated Allan deviation and compare it to the logged data.

```
[avarSim, tauSim] = allanvar(omegaSim, 'octave', Fs);
adevSim = sqrt(avarSim);
```

```
adevSim = mean(adevSim, 2); % Use the mean of the simulations.

figure
loglog(tau, adev, tauSim, adevSim, '--')
title('Allan Deviation of HW and Simulation')
xlabel('\tau');
ylabel('\sigma(\tau)')
legend('HW', 'SIM')
grid on
axis equal
```
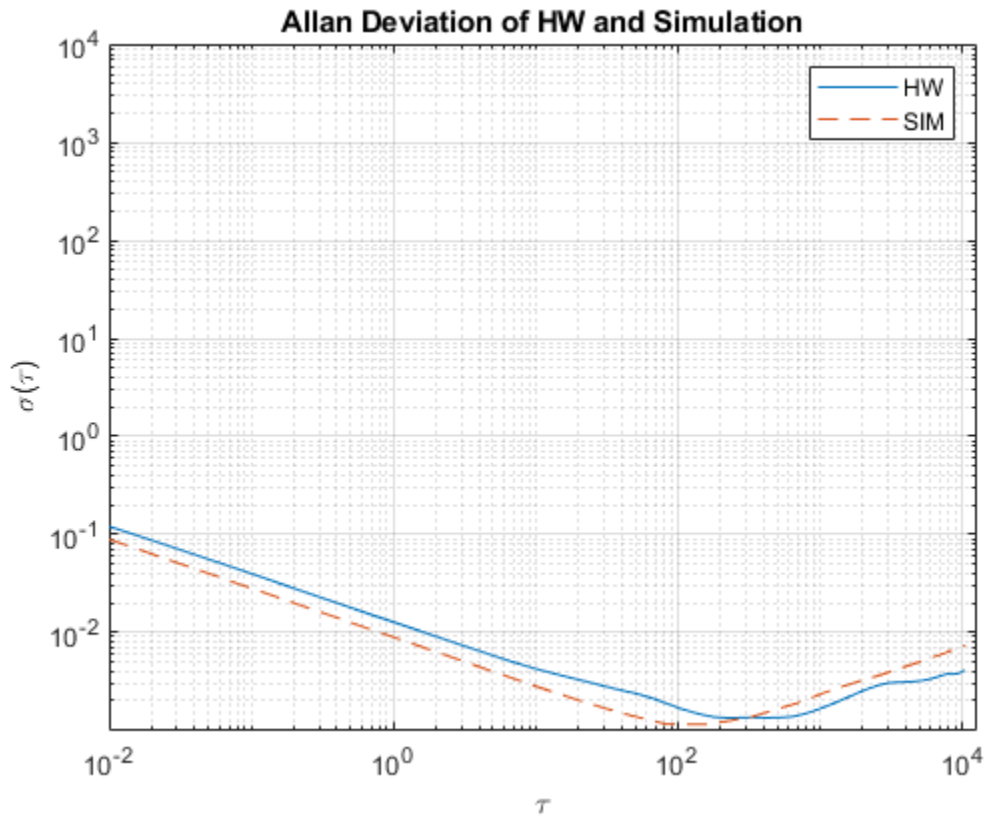

The plot shows that the gyroscope model created from the `imuSensor` generates measurements with similar Allan deviation to the logged data. The model measurements contain slightly less noise since the quantization and temperature-related parameters are not set using `gyroparams`. The gyroscope model can be used to generate measurements using movements that are not easily captured with hardware.

References

- IEEE Std. 647-2006 IEEE Standard Specification Format Guide and Test Procedure for Single-Axis Laser Gyros

Estimate Orientation Through Inertial Sensor Fusion

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

Orientation

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

For the following algorithms, the fixed, parent coordinate system used is North-East-Down (NED). NED is sometimes referred to as the global coordinate system or reference frame. In the NED reference frame, the X-axis points north, the Y-axis points east, and the Z-axis points downward. The X-Y plane of NED is considered to be the local tangent plane of the Earth. Depending on the algorithm, north may be either magnetic north or true north. The algorithms in this example use magnetic north.

If specified, the following algorithms can estimate orientation relative to East-North-Up (ENU) parent coordinate system instead of NED.

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

The orientation quantity computed is a rotation that takes quantities from the parent reference frame to the child reference frame. The rotation is represented by a quaternion or rotation matrix.

Types of Sensors

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

Sensor Data

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

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

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

viewer = HelperOrientationViewer;
```

Accelerometer-Magnetometer Fusion

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

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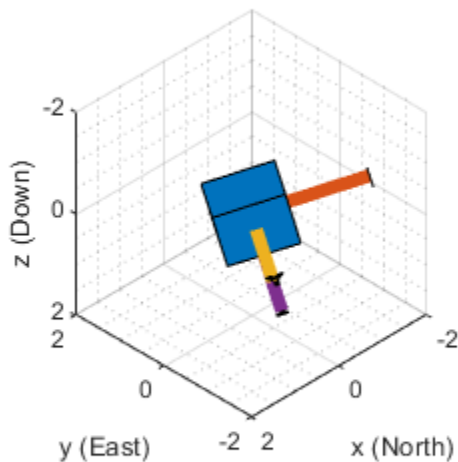

Note that the `ecompass` algorithm correctly finds the location of north. However, because the function is memoryless, the estimated motion is not smooth. The algorithm could be used as an initialization step in an orientation filter or some of the techniques presented in the "Lowpass Filter Orientation Using Quaternion SLERP" on page 5-47 could be used to smooth the motion.

Accelerometer-Gyroscope Fusion

The following objects estimate orientation using either an error-state Kalman filter or a complementary filter. The error-state Kalman filter is the standard estimation filter and allows for many different aspects of the system to be tuned using the corresponding noise parameters. The complementary filter can be used as a substitute for systems with memory constraints, and has minimal tunable parameters, which allows for easier configuration at the cost of finer tuning.

The `imufilter` and `complementaryFilter` System objects™ fuse accelerometer and gyroscope data. The `imufilter` uses an internal error-state Kalman filter and the `complementaryFilter` uses a complementary filter. The filters are capable of removing the gyroscope's bias noise, which drifts over time.

```
ifilt = imufilter('SampleRate', ld.Fs);
for ii=1:size(acc,1)
```

```
        qimu = ifilt(acc(ii,:), gyro(ii,:));
        viewer(qimu);
        pause(0.01);
    end
```
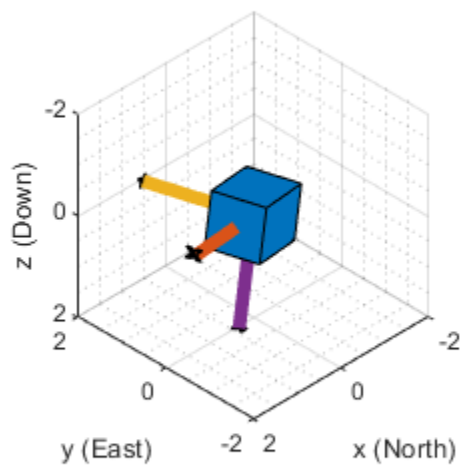


```
    % Disable magnetometer input.
    cfilt = complementaryFilter('SampleRate', ld.Fs, 'HasMagnetometer', false);
    for ii=1:size(acc,1)
        qimu = cfilt(acc(ii,:), gyro(ii,:));
        viewer(qimu);
        pause(0.01);
    end
```
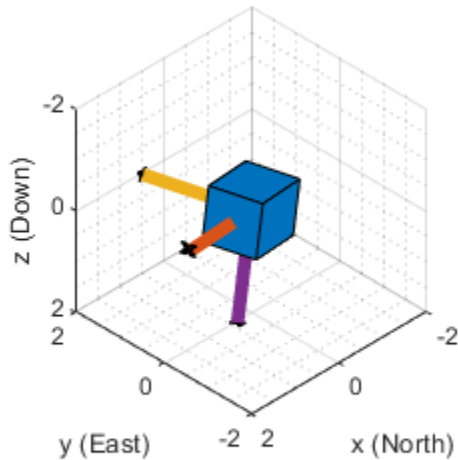
Although the `imufilter` and `complementaryFilter` algorithms produce significantly smoother estimates of the motion, compared to the `ecompass`, they do not correctly estimate the direction of north. The `imufilter` does not process magnetometer data, so it simply assumes the device's X-axis is initially pointing northward. The motion estimate given by `imufilter` is relative to the initial estimated orientation. The `complementaryFilter` makes the same assumption when the `HasMagnetometer` property is set to `false`.

Accelerometer-Gyroscope-Magnetometer Fusion

An attitude and heading reference system (AHRS) consists of a 9-axis system that uses an accelerometer, gyroscope, and magnetometer to compute orientation. The `ahrsfilter` and `complementaryFilter` System objects™ combine the best of the previous algorithms to produce a smoothly changing estimate of the device orientation, while correctly estimating the direction of north. The `complementaryFilter` uses the same complementary filter algorithm as before, with an extra step to include the magnetometer and improve the orientation estimate. Like `imufilter`, `ahrsfilter` algorithm also uses an error-state Kalman filter. In addition to gyroscope bias removal, the `ahrsfilter` has some ability to detect and reject mild magnetic jamming.

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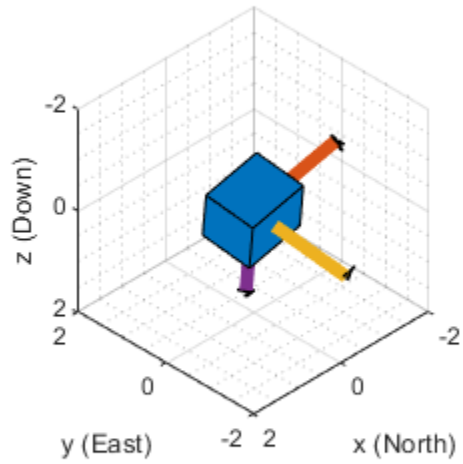
```
cfilt = complementaryFilter('SampleRate', ld.Fs);
for ii=1:size(acc,1)
    qahrs = cfilt(acc(ii,:), gyro(ii,:), mag(ii,:));
    viewer(qahrs);
    pause(0.01);
end
```
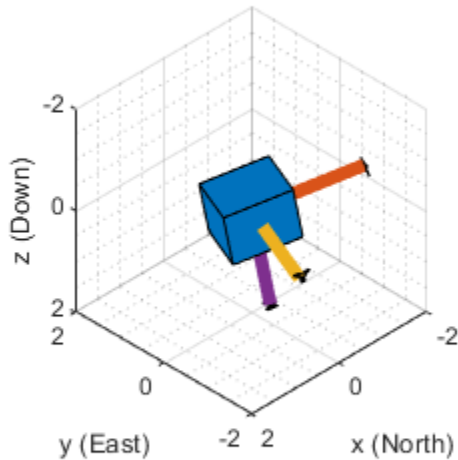
Tuning Filter Parameters

The `complementaryFilter`, `imufilter`, and `ahrsfilter` System objects™ all have tunable parameters. Tuning the parameters based on the specified sensors being used can improve performance.

The `complementaryFilter` parameters `AccelerometerGain` and `MagnetometerGain` can be tuned to change the amount each sensor's measurements impact the orientation estimate. When `AccelerometerGain` is set to `0`, only the gyroscope is used for the x- and y-axis orientation. When `AccelerometerGain` is set to `1`, only the accelerometer is used for the x- and y-axis orientation. When `MagnetometerGain` is set to `0`, only the gyroscope is used for the z-axis orientation. When `MagnetometerGain` is set to `1`, only the magnetometer is used for the z-axis orientation.

The `ahrsfilter` and `imufilter` System objects™ have more parameters that can allow the filters to more closely match specific hardware sensors. The environment of the sensor is also important to take into account. The `imufilter` parameters are a subset of the `ahrsfilter` parameters. The `AccelerometerNoise`, `GyroscopeNoise`, `MagnetometerNoise`, and `GyroscopeDriftNoise` are measurement noises. The sensors' datasheets help determine those values.

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

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

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

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

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

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

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

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

figure;
plot([squeeze(gravDistHigh), squeeze(gravDistLow)]);
title('Euclidean Distance to Gravity');
legend('LinearAccelerationNoise = 0.009', ...
    'LinearAccelerationNoise = 0.0009');
```
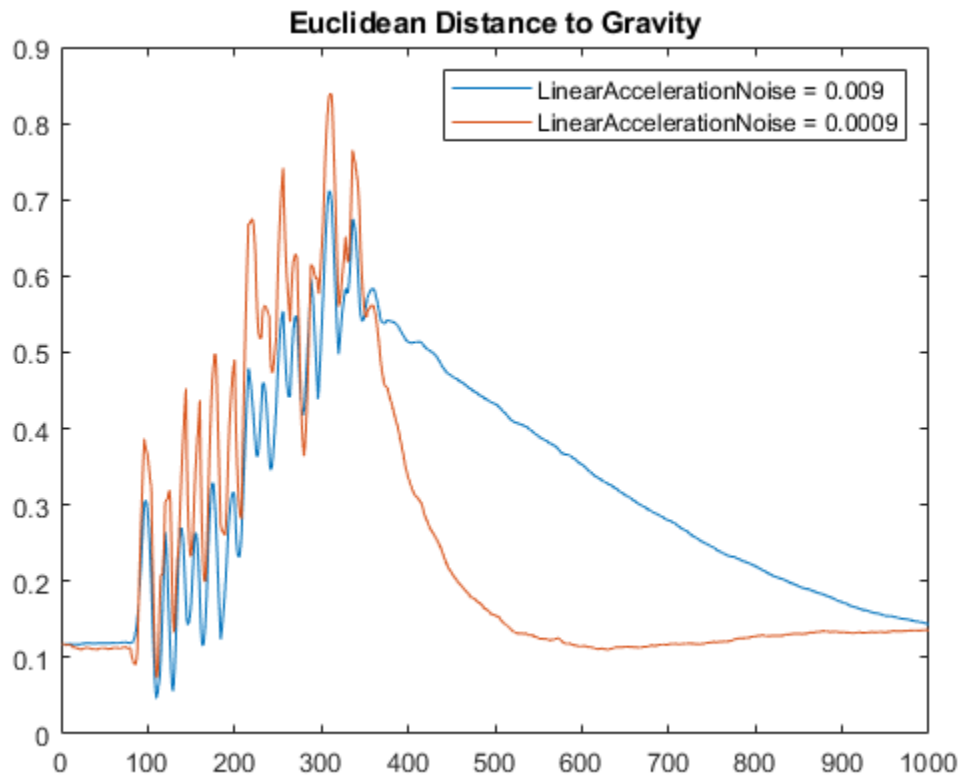
The `lowVarFilt` has a low `LinearAccelerationNoise`, so it expects to be in an environment with low linear acceleration. Therefore, it is more susceptible to linear acceleration, as illustrated by the large variations earlier in the plot. However, because it expects to be in an environment with a low linear acceleration, higher trust is placed in the accelerometer signal. As such, the orientation estimate converges quickly back to vertical once the shaking has ended. The converse is true for `highVarFilt`. The filter is less affected by shaking, but the orientation estimate takes longer to converge to vertical when the shaking has stopped.

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

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

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

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

figure;
timevec = 0:ld.Fs:(ld.Fs*numel(qpulse) - 1);
plot( timevec, eulerd(qpulse, 'ZYX', 'frame') );
title(['Stationary Trajectory Orientation Euler Angles' newline ...
    'Magnetic Jamming Response']);
legend('Z-rotation', 'Y-rotation', 'X-rotation');
ylabel('Degrees');
xlabel('Seconds');
```
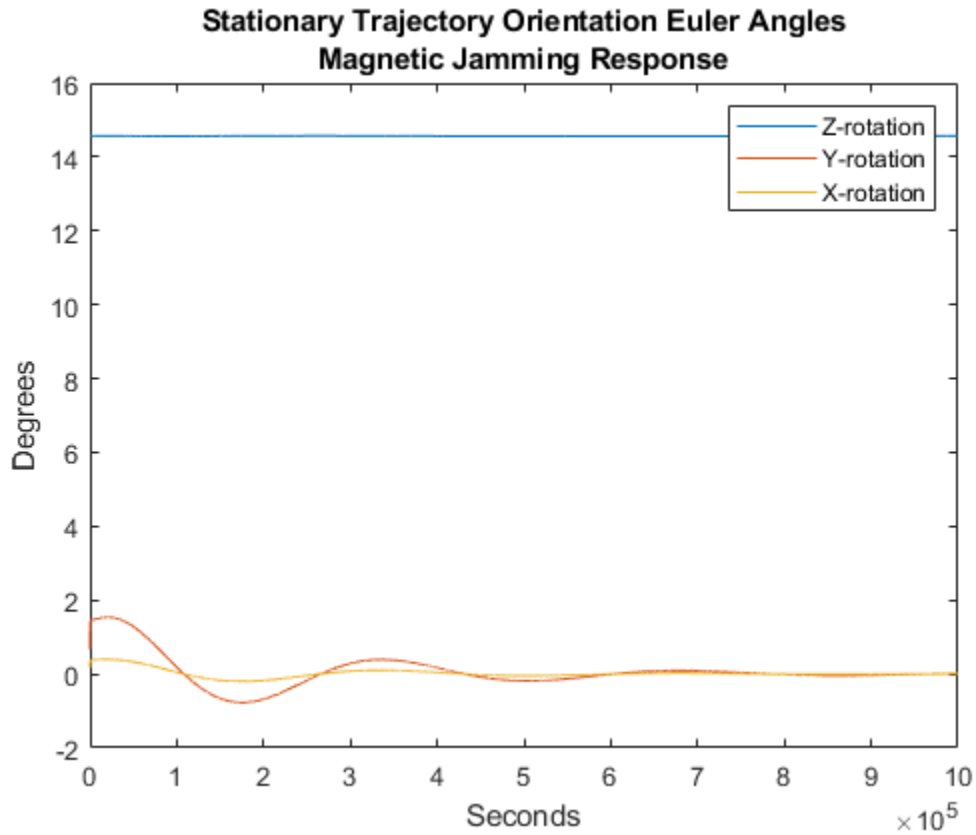
Note that the filter almost totally rejects this magnetic pulse as interference. Any magnetic field strength greater than four times the `ExpectedMagneticFieldStrength` is considered a jamming source and the magnetometer signal is ignored for those samples.

Conclusion

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

Estimate Orientation and Height Using IMU, Magnetometer, and Altimeter

This example shows how to fuse data from a 3-axis accelerometer, 3-axis gyroscope, 3-axis magnetometer (together commonly referred to as a MARG sensor for Magnetic, Angular Rate, and Gravity), and 1-axis altimeter to estimate orientation and height.

Simulation Setup

This simulation processes sensor data at multiple rates. The IMU (accelerometer and gyroscope) typically runs at the highest rate. The magnetometer generally runs at a lower rate than the IMU, and the altimeter runs at the lowest rate. Changing the sample rates causes parts of the fusion algorithm to run more frequently and can affect performance.

```matlab
% Set the sampling rate for IMU sensors, magnetometer, and altimeter.
imuFs = 100;
altFs = 10;
magFs = 25;
imuSamplesPerAlt = fix(imuFs/altFs);
imuSamplesPerMag = fix(imuFs/magFs);

% Set the number of samples to simulate.
N = 1000;

% Construct object for other helper functions.
hfunc = Helper10AxisFusion;
```

Define Trajectory

The sensor body rotates about all three axes while oscillating in position vertically. The oscillations increase in magnitude as the simulation continues.

```matlab
% Define the initial state of the sensor body
initPos = [0, 0, 0];        % initial position (m)
initVel = [0, 0, -1];       % initial linear velocity (m/s)
initOrient = ones(1, 'quaternion');

% Define the constant angular velocity for rotating the sensor body
% (rad/s).
angVel = [0.34 0.2 0.045];

% Define the acceleration required for simple oscillating motion of the
% sensor body.
fc = 0.2;
t = 0:1/imuFs:(N-1)/imuFs;
a = 1;
oscMotionAcc = sin(2*pi*fc*t);
oscMotionAcc = hfunc.growAmplitude(oscMotionAcc);

% Construct the trajectory object
traj = kinematicTrajectory('SampleRate', imuFs, ...
    'Velocity', initVel, ...
    'Position', initPos, ...
    'Orientation', initOrient);
```

Sensor Configuration

The accelerometer, gyroscope and magnetometer are simulated using `imuSensor`. The altimeter is modeled using the `altimeterSensor`. The values used in the sensor configurations correspond to real MEMS sensor values.

```
imu = imuSensor('accel-gyro-mag', 'SampleRate', imuFs);

% Accelerometer
imu.Accelerometer.MeasurementRange =  19.6133;
imu.Accelerometer.Resolution = 0.0023928;
imu.Accelerometer.ConstantBias = 0.19;
imu.Accelerometer.NoiseDensity = 0.0012356;

% Gyroscope
imu.Gyroscope.MeasurementRange = deg2rad(250);
imu.Gyroscope.Resolution = deg2rad(0.0625);
imu.Gyroscope.ConstantBias = deg2rad(3.125);
imu.Gyroscope.AxesMisalignment = 1.5;
imu.Gyroscope.NoiseDensity = deg2rad(0.025);

% Magnetometer
imu.Magnetometer.MeasurementRange = 1000;
imu.Magnetometer.Resolution = 0.1;
imu.Magnetometer.ConstantBias = 100;
imu.Magnetometer.NoiseDensity = 0.3/sqrt(50);

% altimeter
altimeter = altimeterSensor('UpdateRate', altFs, 'NoiseDensity', 2*0.1549);
```

Fusion Filter

Construct an `ahrs10filter` and configure.

```
fusionfilt = ahrs10filter;
fusionfilt.IMUSampleRate = imuFs;
```

Set initial values for the fusion filter.

```
initstate = zeros(18,1);
initstate(1:4) = compact(initOrient);
initstate(5) = initPos(3);
initstate(6) = initVel(3);
initstate(7:9) = imu.Gyroscope.ConstantBias/imuFs;
initstate(10:12) = imu.Accelerometer.ConstantBias/imuFs;
initstate(13:15) = imu.MagneticField;
initstate(16:18) = imu.Magnetometer.ConstantBias;
fusionfilt.State = initstate;
```

Initialize the state covariance matrix of the fusion filter. The ground truth is used for initial states, so there should be little error in the estimates.

```
icv = diag([1e-8*[1 1 1 1 1 1 1], 1e-3*ones(1,11)]);
fusionfilt.StateCovariance = icv;
```

Magnetometer and altimeter measurement noises are the observation noises associated with the sensors used by the internal Kalman filter in the `ahrs10filter`. These values would normally come from a sensor datasheet.

```
magNoise =  2*(imu.Magnetometer.NoiseDensity(1).^2)*imuFs;
altimeterNoise = 2*(altimeter.NoiseDensity).^2 * altFs;
```

Filter process noises are used to tune the filter to desired performance.

```
fusionfilt.AccelerometerNoise = [1e-1 1e-1 1e-4];
fusionfilt.AccelerometerBiasNoise = 1e-8;
fusionfilt.GeomagneticVectorNoise = 1e-12;
fusionfilt.MagnetometerBiasNoise = 1e-12;
fusionfilt.GyroscopeNoise = 1e-12;
```

Additional Simulation Option : Viewer

By default, this simulation plots the estimation errors at the end of the simulation. To view both the estimated position and orientation along with the ground truth as the simulation runs, set the `usePoseViewer` variable to `true`.

```
usePoseViewer = false;
```

Simulation Loop

```
q = initOrient;
firstTime = true;

actQ = zeros(N,1, 'quaternion');
expQ = zeros(N,1, 'quaternion');
actP = zeros(N,1);
expP = zeros(N,1);

for ii = 1: N
    % Generate a new set of samples from the trajectory generator
    accBody = rotateframe(q, [0 0 +oscMotionAcc(ii)]);
    omgBody = rotateframe(q, angVel);
    [pos, q, vel, acc] = traj(accBody, omgBody);

    % Feed the current position and orientation to the imuSensor object
    [accel, gyro, mag] = imu(acc, omgBody, q);
    fusionfilt.predict(accel, gyro);

    % Fuse magnetometer samples at the magnetometer sample rate
    if ~mod(ii,imuSamplesPerMag)
        fusemag(fusionfilt, mag, magNoise);
    end

    % Sample and fuse the altimeter output at the altimeter sample rate
    if ~mod(ii,imuSamplesPerAlt)
        altHeight = altimeter(pos);

        % Use the |fusealtimeter| method to update the fusion filter with
        % the altimeter output.
        fusealtimeter(fusionfilt,altHeight,altimeterNoise);
    end

    % Log the actual orientation and position
    [actP(ii), actQ(ii)] = pose(fusionfilt);

    % Log the expected orientation and position
    expQ(ii) = q;
```

```
    expP(ii) = pos(3);

    if usePoseViewer
        hfunc.view(actP(ii), actQ(ii),expP(ii), expQ(ii)); %#ok<*UNRCH>
    end

end
```

Plot Filter Performance

Plot the performance of the filter. The display shows the error in the orientation using quaternion distance and height error.

```
hfunc.plotErrs(actP, actQ, expP, expQ);
```
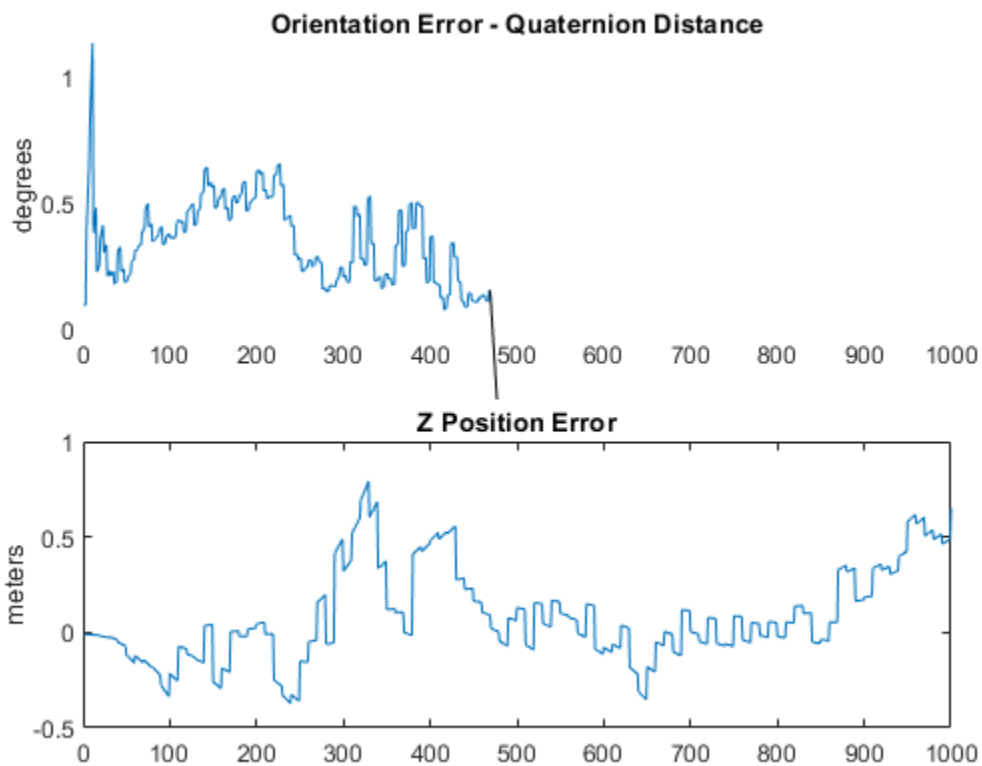

Conclusion

This example shows how to the `ahrs10filter` to perform 10-axis sensor fusion for height and orientation.

Scanning Radar Mode Configuration

This example shows you how to model different radar scan modes using the `monostaticRadarSensor`. This example shows how to configure the `monostaticRadarSensor` for several commonly used radar scan modes. With this model, you can simulate radars which mechanically scan, electronically scan, and which use both mechanical and electronic scanning. The scan limits in azimuth and elevation are configurable for both mechanical and electronic scan modes.

Mechanical Rotator

360 Degree Azimuth Scan

360 degree azimuth scan is a mode commonly found on both ground and airborne radars. This mode provides 360 degree surveillance by mechanically scanning the radar's antenna in azimuth. In this mode, radars typically employ a fan beam, which has a narrow field of view in azimuth, but covers a wide elevation span. These radars provide accurate range and azimuth measurements, but typically do not report elevation for the detected targets. A common example of a 360 degree azimuth scan radar is the airport surveillance radar.

Use the function `helperScanRadarModesExample` to create a `trackingScenario` with a platform to mount the radar and three targets. This function also sets up a `theaterPlot` to display the locations of the targets, the beam and boresight position for the radar, and the detections generated by the radar.

You can learn more about using `trackingScenario` and `theaterPlot` from the "Introduction to Tracking Scenario and Simulating Sensor Detections" on page 5-86 example.

```
% Initialize trackingScenario for simulation and theaterPlot for
% visualization
ts = trackingScenario;
[platform,fig] = helperScanRadarModesExample('Setup tracking scenario',ts);

% Set random seed for repeatable results
rng(2018)
```

Create a radar that mechanically rotates its antenna 360 degrees in azimuth. Mount it 15 meters above the platform with a yaw of -135 degrees from the platform's axes. Increase the azimuth field of view for better visibility in the displayed figures.

```
% Create a 360 degree mechanically rotating radar
radar = monostaticRadarSensor(1,'rotator');

% Locate the radar 15 meters above the platform
radar.MountingLocation = [0 0 -15];

% Rotate the radar so that it is yawed -135 degrees from the platform's axes
radar.MountingAngles(1) = -135;

% Set the radar's azimuth field of view to 5 degrees to display larger beams
radar.FieldOfView(1) = 5;

% Display configured radar
radar
```

```
radar =
```

```
    monostaticRadarSensor with properties:

                    SensorIndex: 1
                     UpdateRate: 1
                       ScanMode: 'Mechanical'

               MountingLocation: [0 0 -15]
                 MountingAngles: [-135 0 0]

             MaxUnambiguousRange: 100000
       MaxUnambiguousRadialSpeed: 200
                     FieldOfView: [2x1 double]
           MaxMechanicalScanRate: 75
            MechanicalScanLimits: [0 360]
                 MechanicalAngle: 0
                       LookAngle: 0

             DetectionProbability: 0.9000
                   FalseAlarmRate: 1.0000e-06

  Use get to show all properties
```

Configure the radar to mechanically scan at a rate of 2 rpm. In this mode, the radar schedules beams at each dwell that are spaced by the radar's azimuthal field of view. The radar's update rate is then computed from the desired scan rate and its azimuthal field of view.

```
rpm = 2;
fov = radar.FieldOfView;
scanrate = rpm*360/60;          % deg/s
updaterate = scanrate/fov(1)    % Hz
radar.UpdateRate = updaterate;
```

```
updaterate =

    2.4000
```

Use `trackingScenario` to simulate the motion of the targets in the scenario and generate detections using the mechanically rotating radar model. `helperScanRadarModesExample` is used to update the `theaterPlot` with the platform positions, radar beam and boresight location, and detections generated at each step of the simulation.

```
% Configure trackingScenario to advance at the radar's update rate.
ts.UpdateRate = radar.UpdateRate;

% Run simulation
figure(fig);
title('360 Degree Azimuth Scan');
while advance(ts) && ishghandle(fig)

    % Current simulation time
    simTime = ts.SimulationTime;

    % Current target positions
    targets = targetPoses(platform);
```

```
% Generate target detections at current scan position
dets = radar(targets,simTime);

% Update display
helperScanRadarModesExample('Update display',ts,platform,radar,dets);

% Take a snapshot of detection on inbound target
takeSnapshot = simTime> 5 && any(cellfun(@(d)d.ObjectAttributes{1}.TargetIndex,dets)==2);
snapped = helperScanRadarModesExample('Snapshot',fig,takeSnapshot);
if snapped
    close(fig);
end
end
```

360 Degree Azimuth Scan
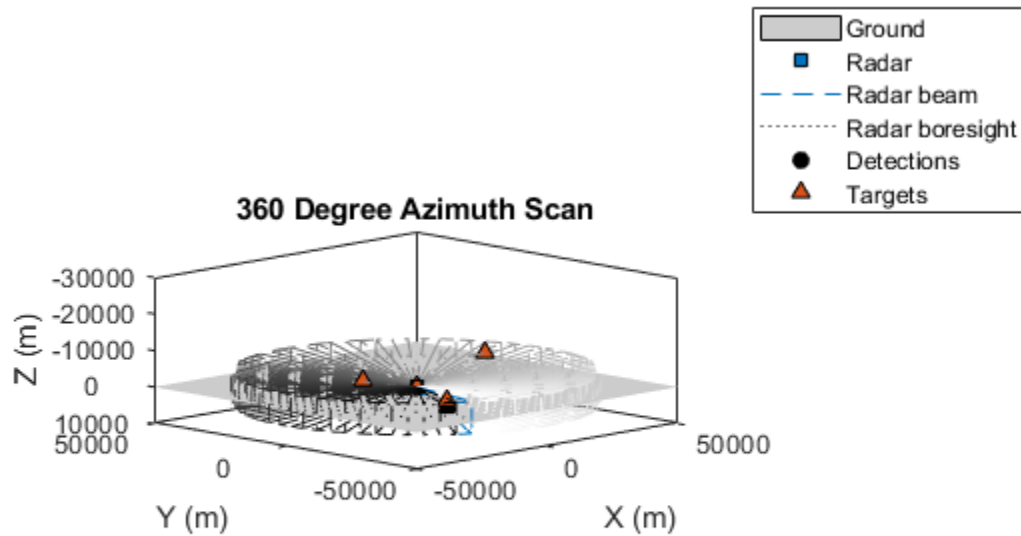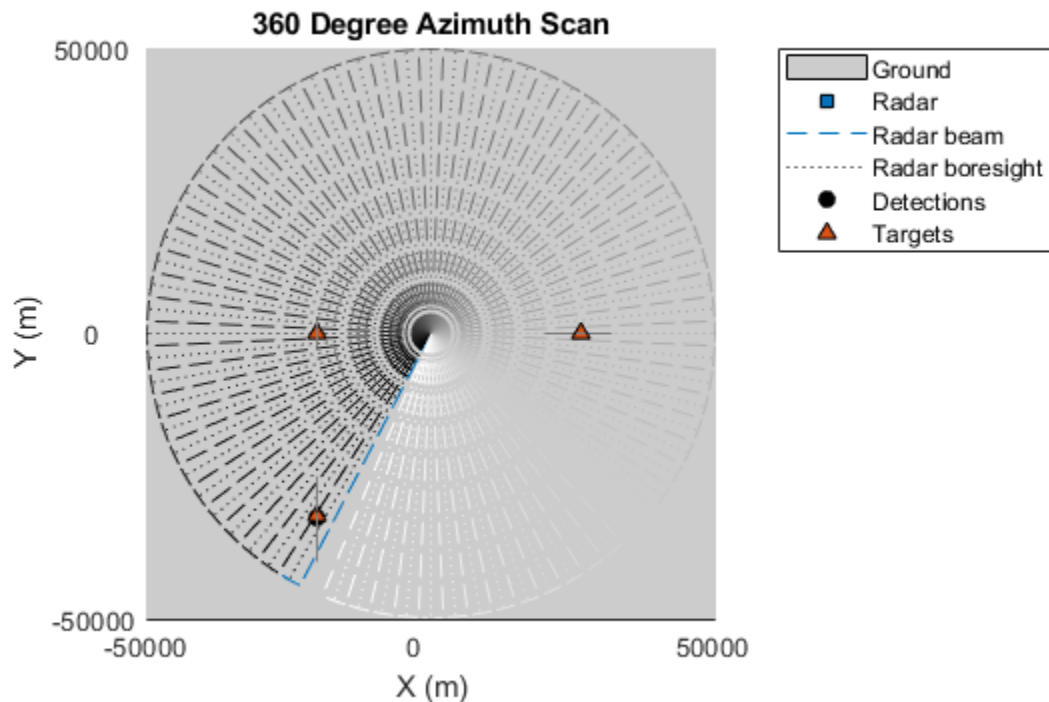
The preceding figures show the detection of the inbound target in 3-D and 2-D views. The inbound target is detected by the radar when its beam sweeps across its position. The radar's boresight is shown as a black, dotted line. The radar's current beam is shown as a blue, dashed line. The history of beam and boresight positions are shown in gray scale, with the more recent positions shown as black and the older positions fading to white.

360 Degree Azimuth Scan with Tilted Elevation Coverage

In the preceding section, the boresight of the radar's antenna (black, dotted line) was constrained to lie in the horizontal plane, resulting in half of the radar's beam directed below the horizontal plane. If you are modeling a ground-based radar, you may want to tilt the radar's boresight upwards so that only the area above the ground is surveyed by the radar. Conversely, for an airborne platform, you may desire to point the radar's beam downwards, to survey targets at altitudes below the radar's platform.

Tilt the boresight of the radar antenna so that none of the elevation span of the beam lies below the ground. To do this, enable elevation for the radar and set the elevation mechanical scan limits to search from the ground up to the full elevation field of view. Then set the elevation field of view to be slightly greater than the elevation spanned by the mechanical scan limits so that no raster scanning is performed by the radar (raster scanning is addressed in the next section).

As previously mentioned, the radar schedules beams that are spaced by the azimuth and elevation field of view to cover the entire mechanical scan limits. By setting the elevation field of view to be slightly larger than the mechanical scan limits in elevation, the radar places the beams in the middle of the mechanical scan limits.

This configuration was already set up when you created the radar using the 'rotator' configuration. All that you need to do is enable elevation.

```
release(radar);
radar.HasElevation = true;

% Confirm mechanical scan limits
radar.MechanicalScanLimits
```

```
ans =

     0    360
   -10      0
```

```
% Confirm elevation field of view is slightly larger than the elevation
% spanned by the scan limits so that raster scan is not performed
elSpan = diff(radar.MechanicalScanLimits(2,:))
isLarger = radar.FieldOfView(2)>elSpan
```

```
elSpan =

    10
```

```
isLarger =

  logical

   1
```

Use the `helperScanRadarModesExample` function to run the simulation as was done by using a while-loop in the preceding section.

```
helperScanRadarModesExample('Run simulation',ts,platform,radar,2);
```

**360 Degree Azimuth Scan
with Tilted Elevation Coverage**

Ground
Radar
Radar beam
Radar boresight
Detections
Targets

Z (m)

-30000
-20000
-10000
0
10000
50000

Y (m)

0
-50000

-50000
0

50000

50000

X (m)

360 Degree Azimuth Scan
with Tilted Elevation Coverage
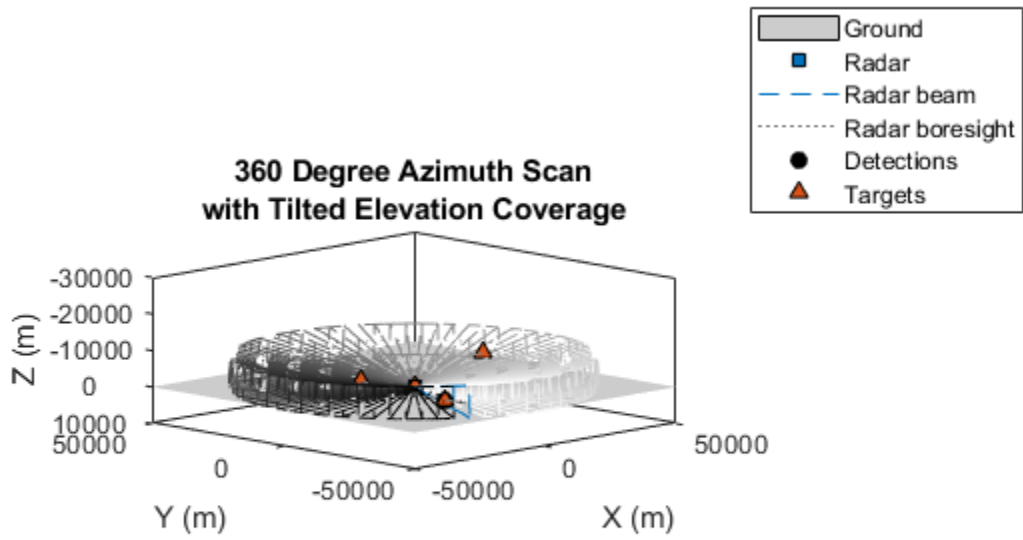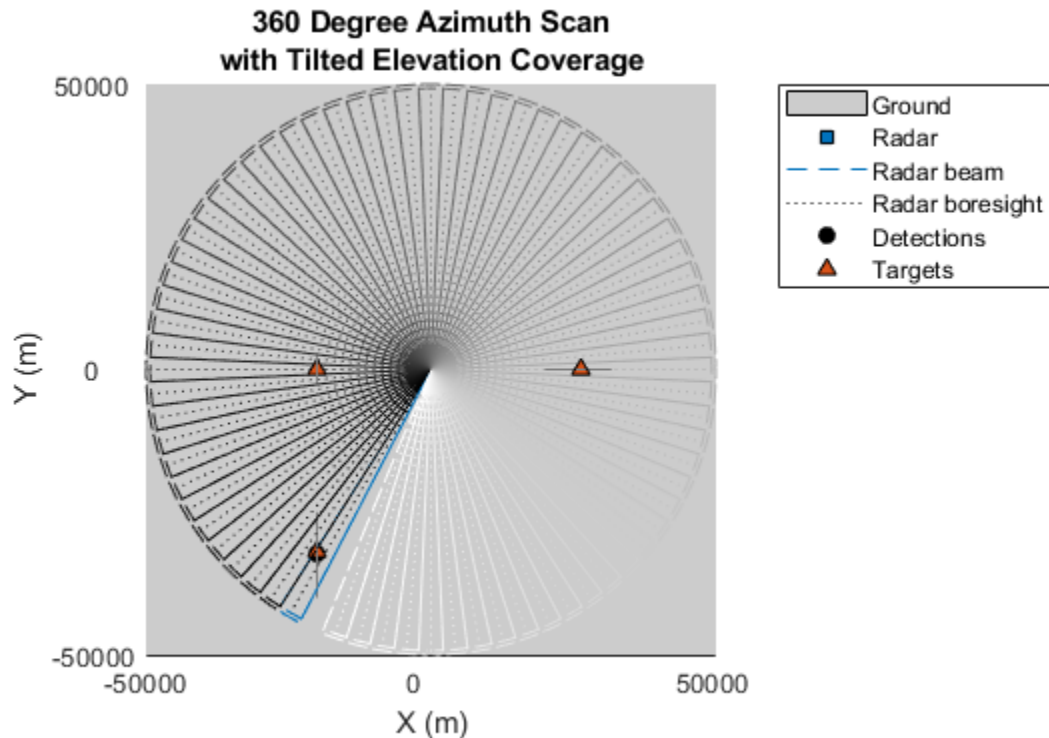
In the preceding figures, you observe that the antenna is tilted upwards so that no radar energy is directed below the ground. The entire radar beam (blue, dashed lines) lies above the ground.

360 Degree Azimuth with Elevation Raster

Sometimes a radar must perform 360 degree surveillance which covers a region greater than can be spanned by its elevation field of view. In this case, the radar mechanically rotates in azimuth and mechanically steps its antenna in elevation at the end of each 360 degree scan. This is a form of raster scanning where each raster bar is spaced by the radar's elevation field of view across the radar's elevation scan limits.

Setup the radar to scan a 10 degree region above the ground using a beam which spans 5 degrees in elevation. This produces 3 elevation raster bars at 0, -5, and -10 degrees elevation.

```
release(radar);
elFov = 5; % deg
radar.FieldOfView(2) = elFov;
radar.MechanicalScanLimits(2,:) = [-10 0]-elFov/2;

% Increase the scan rate of the radar to show 2 complete cycles of the
% raster scan pattern.
rpm = 5;
fov = radar.FieldOfView;
scanrate = rpm*360/60;          % deg/s
updaterate = scanrate/fov(1);   % Hz
radar.UpdateRate = updaterate;
```

5-127

```
% Run the simulation
helperScanRadarModesExample('Run simulation',ts,platform,radar,3);
```

360 Degree Azimuth with Elevation Raster
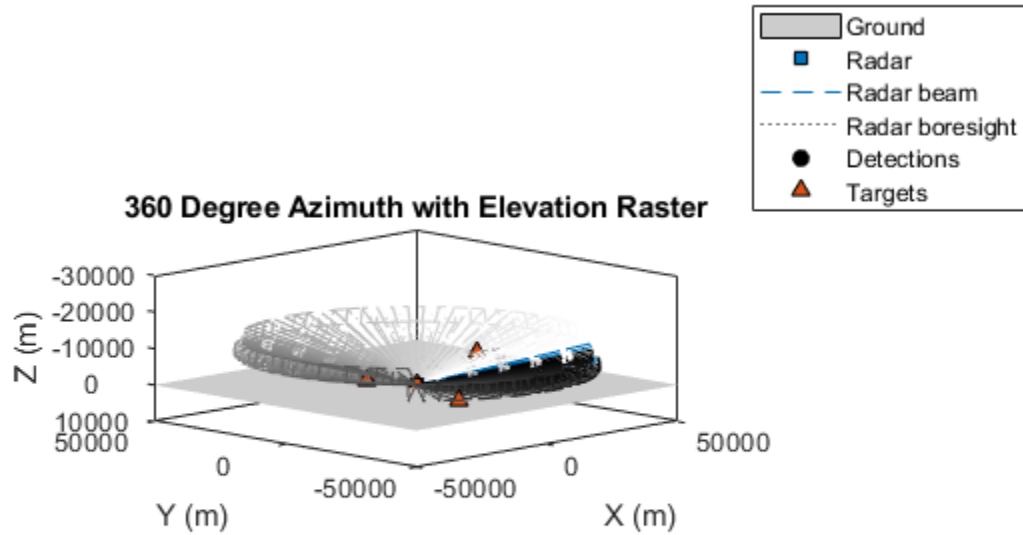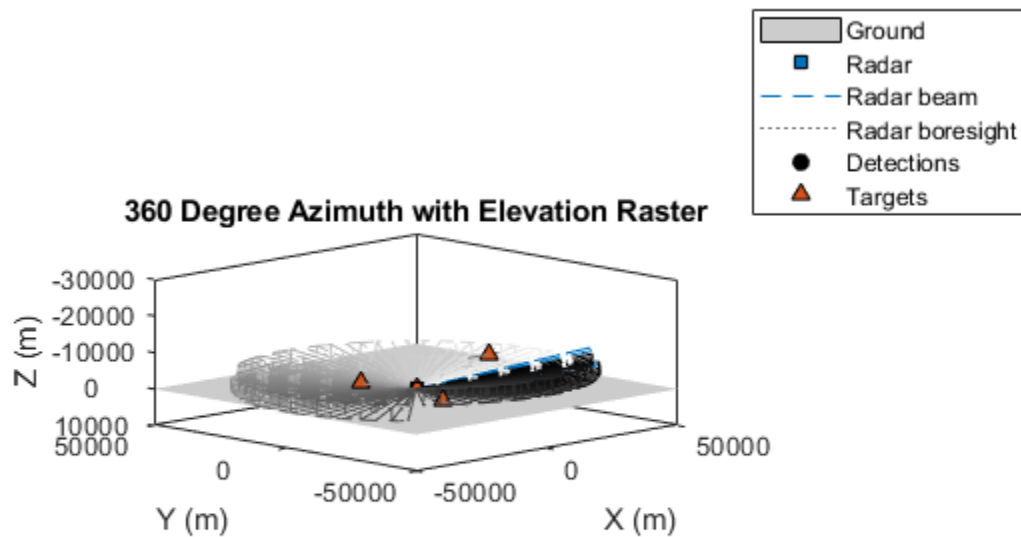
The two preceding figures show the radar's beam positions. The figure on the left shows the beam positions as the beam steps down from the first to the second raster bar (notice the step in elevation in the middle of the figure). The figure on the right shows beam positions after the previous raster scan has completed. In this case, the radar is stepping up from the third raster bar to the second raster bar. The distance between each raster bar is the radar's elevation field of view.

Sector Scan

Mechanical Azimuth Sector Scan

Scanning a full 360 degree sector is time consuming. If the targets of interest are known to occupy a smaller region, a sector scan is typically used. By scanning over a smaller azimuth sector, a higher update rate is achieved for each target within the sector without increasing the mechanical scan rate of the radar.

Setup the radar to scan a 90 degree azimuth sector by setting the mechanical scan limits to span 45 degrees on either side of the radar's mounted orientation. Disable elevation to constrain the radar's beam to the horizontal plane.

Setup a `monostaticRadarSensor` with this configuration by specifying the 'Sector' configuration for the radar.

```
radar = monostaticRadarSensor(1, 'Sector','MountingAngles',[-135 0 0],'MountingLocation',[0 0 -1

% Set the radar's azimuth field of view to 5 degrees to display larger beams
radar.FieldOfView(1) = 5;
```

5-129

```
% Set the update rate to show multiple cycles of the sector scan
rpm = 2;
fov = radar.FieldOfView;
scanrate = rpm*360/60;          % deg/s
updaterate = scanrate/fov(1);   % Hz
radar.UpdateRate = updaterate;

% Run the simulation
helperScanRadarModesExample('Run simulation',ts,platform,radar,4);
```

Mechanical Azimuth Sector Scan

Mechanical Azimuth Sector Scan
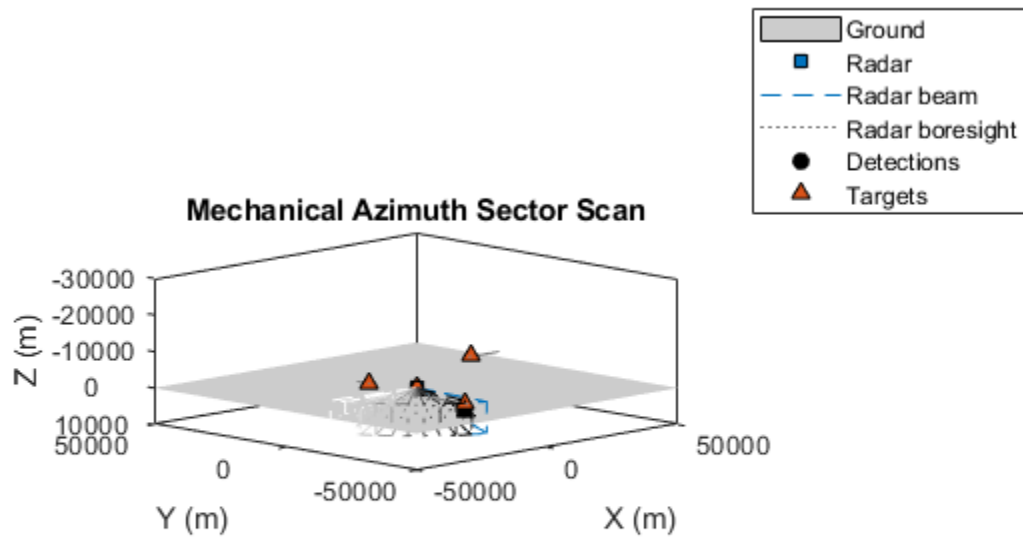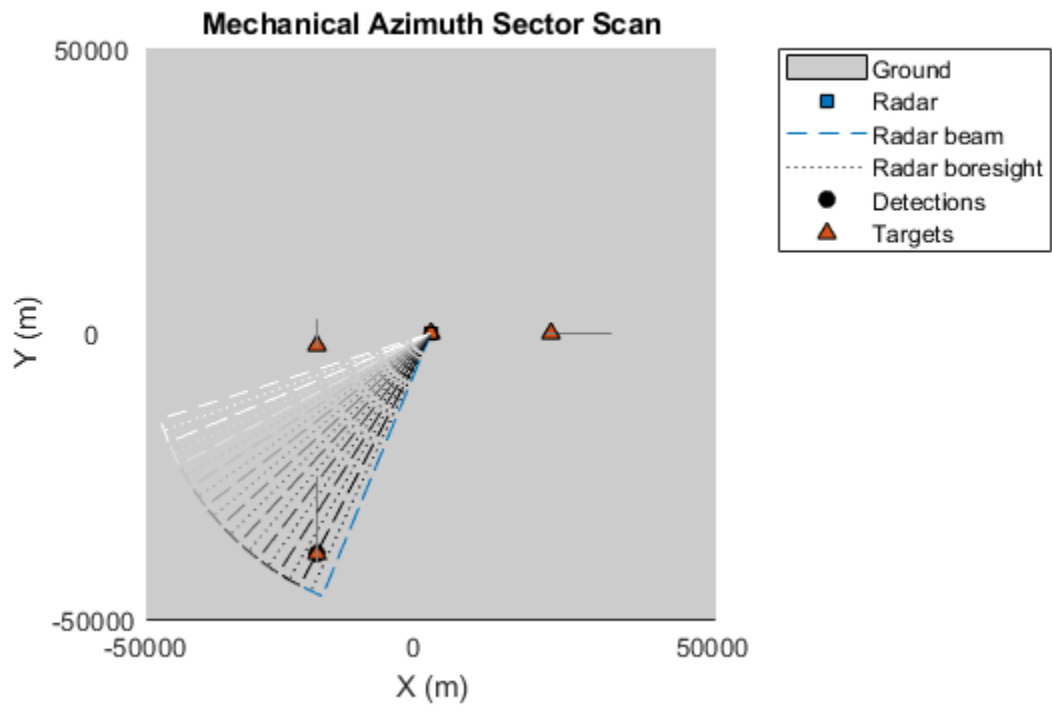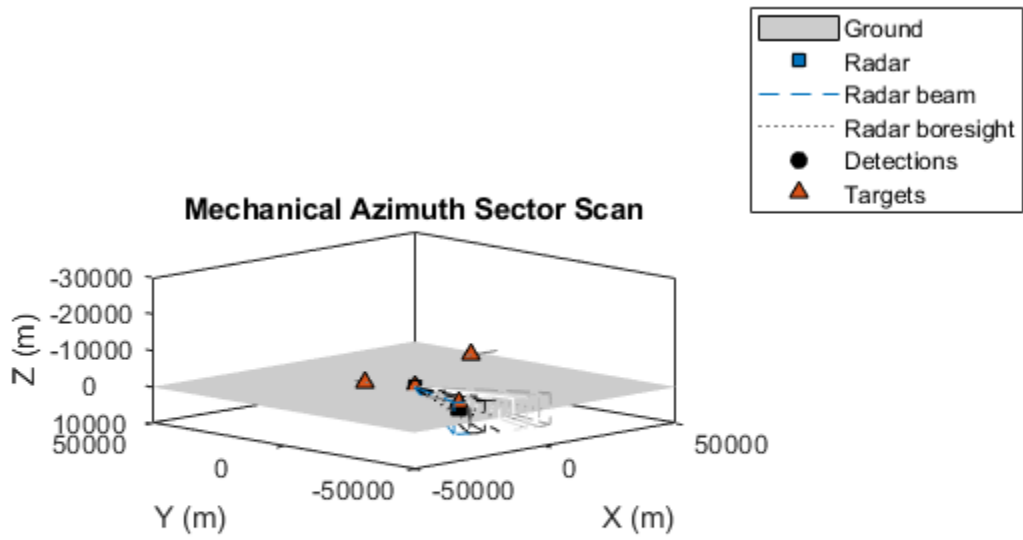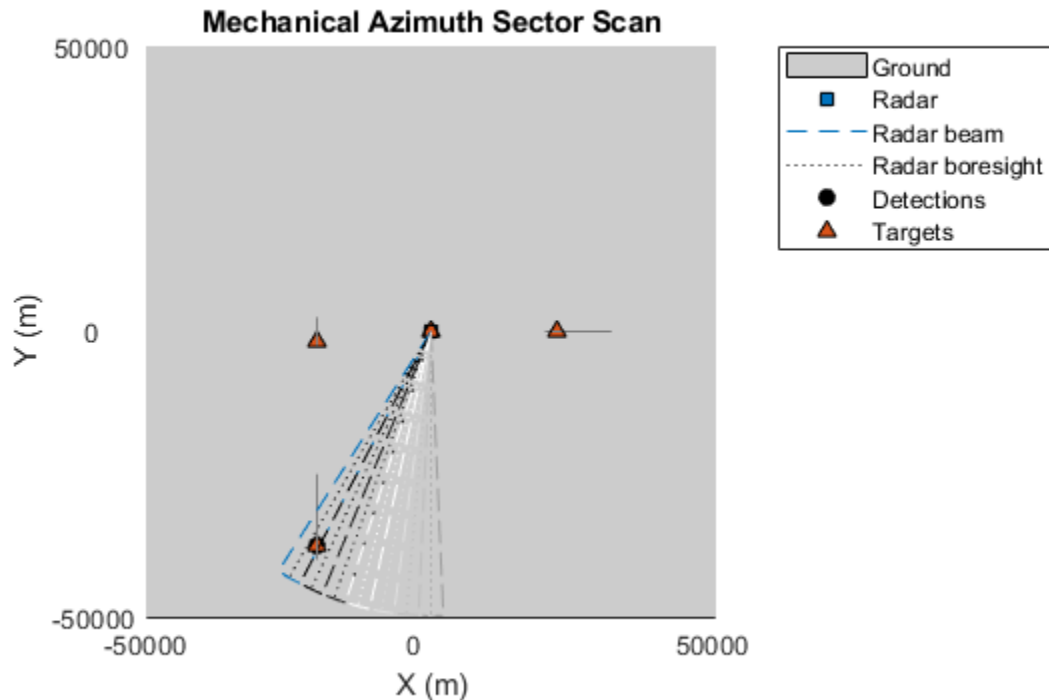
The top row of the figures shows the first scan of the azimuth sector, with the beam traversing the sector from the left to the right side of the figure. The bottom row shows the following scan of the azimuth sector, where the direction of the mechanical scan has reversed, traversing the figure from the right to the left.

Once again, half of the radar's beam lies below the horizontal ground plane. You can mechanically tilt the beam upward or downward using the same technique as you used in the preceding section *360 Degree Azimuth Scan with Tilted Elevation Coverage*.

Mechanical Azimuth Sector Scan with Electronic Elevation Scan

Some radars scan mechanically in azimuth and electronically stack multiple beams in elevation at the antenna's boresight. This avoids the need to perform a raster scan to slowly search out the region of interest. Electronically steering and processing multiple beams in a single dwell position requires more complicated antenna hardware and signal processing algorithms, but provides higher update rates on each target in the sector.

Configure the radar to mechanically scan its beam in azimuth while processing a 10 degree elevation field of view by electronically stacking multiple elevation beams at each dwell position.

```
release(radar);

% Enable mechanical and electronic scanning
radar.ScanMode = 'Mechanical and electronic';

% Enable elevation scanning and measurements
```

```
radar.HasElevation = true;

% Elevation scanning is performed electronically. Set the electronic scan
% limits in elevation to the mechanical scan limits. Set the mechanical
% scan limits to zero to disable mechanical scan in elevation
radar.ElectronicScanLimits(:) = 0;
radar.ElectronicScanLimits(2,:) = radar.MechanicalScanLimits(2,:);
radar.MechanicalScanLimits(2,:) = [0 0];

% Confirm that the elevation field of view is greater than the elevation
% spanned by the scan limits
elSpan = diff(radar.ElectronicScanLimits(2,:))
isLarger = radar.FieldOfView(2)>elSpan


elSpan =

    10


isLarger =

  logical

   1


% Run the simulation
helperScanRadarModesExample('Run simulation',ts,platform,radar,5);
```

Mechanical Azimuth Sector Scan
with Electronic Elevation Scan

Mechanical Azimuth Sector Scan with Electronic Elevation Scan

Mechanical Azimuth Sector Scan with Electronic Elevation Scan
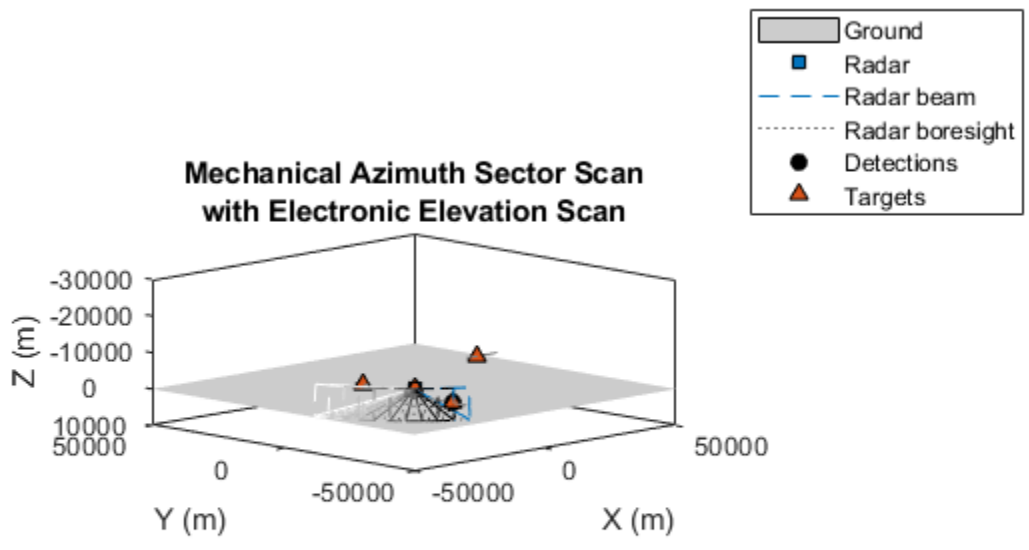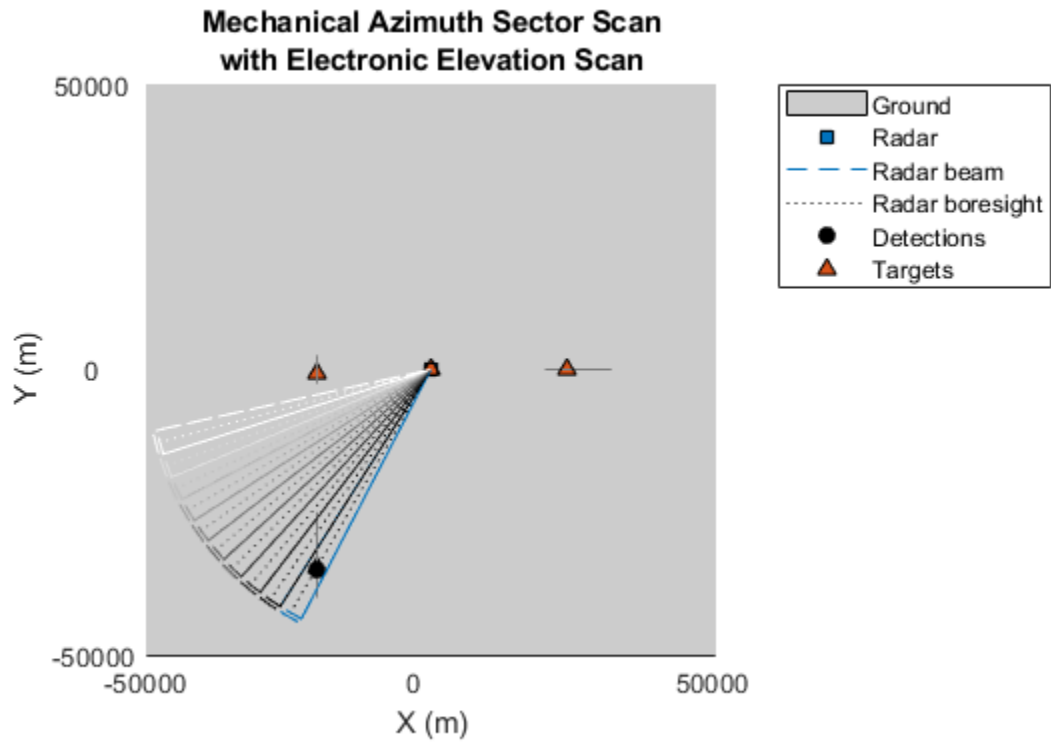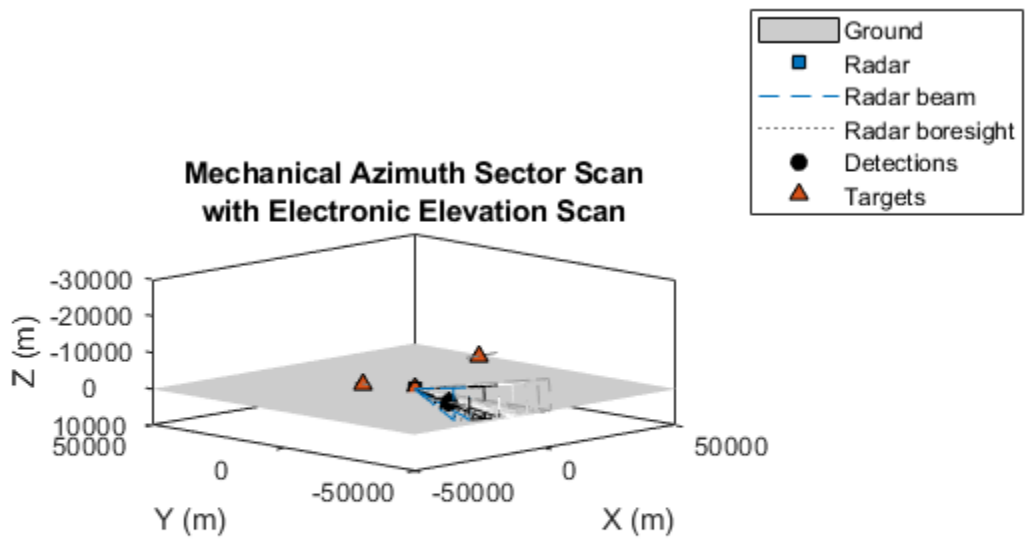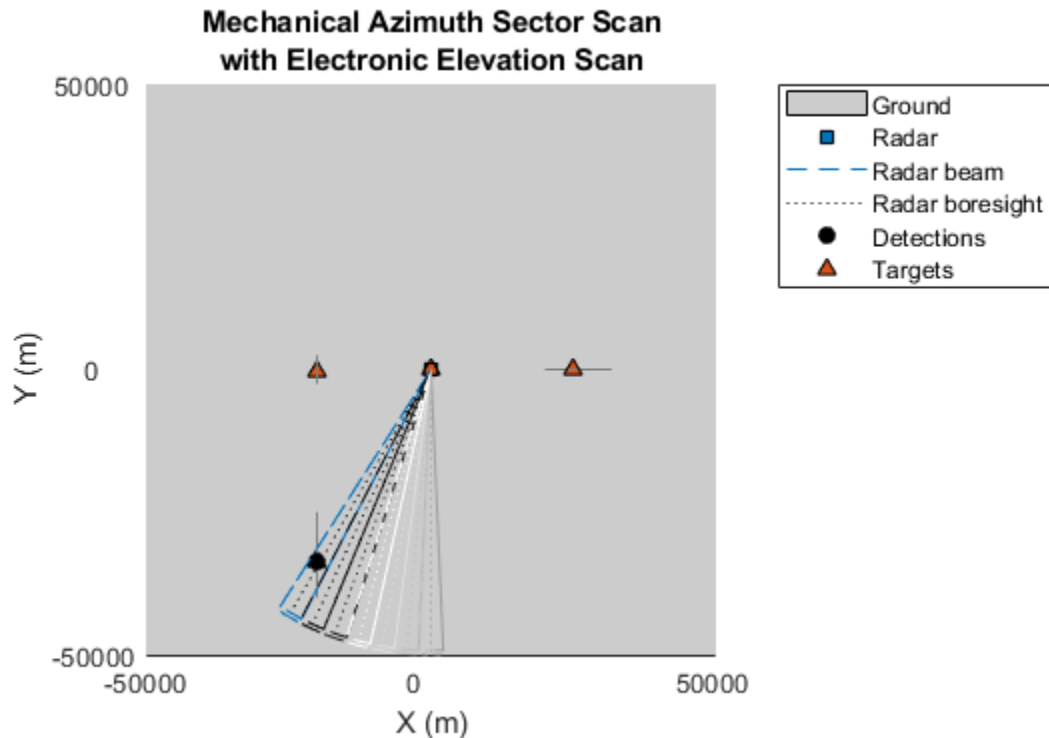
The preceding figures show the radar detecting the inbound target. Notice that the radar's boresight (black, dotted line) lies in the horizontal ground plane, but that the radar's beam is offset in elevation from its boresight. This offset of the beam position from the radar's boresight is accomplished by electronically steering beams in elevation.

Electronic Azimuth Sector Scan

The azimuth sector can be surveyed using electronic scanning as well. Electronically scan the same azimuth sector by creating a `monostaticRadarSensor` using the 'Sector' scan configuration with its scan mode set to 'Electronic' instead of 'Mechanical'. Enable elevation so that the region above the ground is scanned by stacking beams in elevation to span the entire elevation scan limits.

```
radar = monostaticRadarSensor(1, 'Sector','ScanMode','Electronic','HasElevation',true, ...
    'MountingAngles',[-135 0 0],'MountingLocation',[0 0 -15]);

% Set update rate to show multiple cycles of the raster scan pattern
radar.UpdateRate = updaterate;

% Set the radar's azimuth field of view to 5 degrees to display larger beams
radar.FieldOfView(1) = 5;

% Run the simulation
helperScanRadarModesExample('Run simulation',ts,platform,radar,6);
```

Electronic Azimuth Sector Scan

Electronic Azimuth Sector Scan

Electronic Azimuth Sector Scan
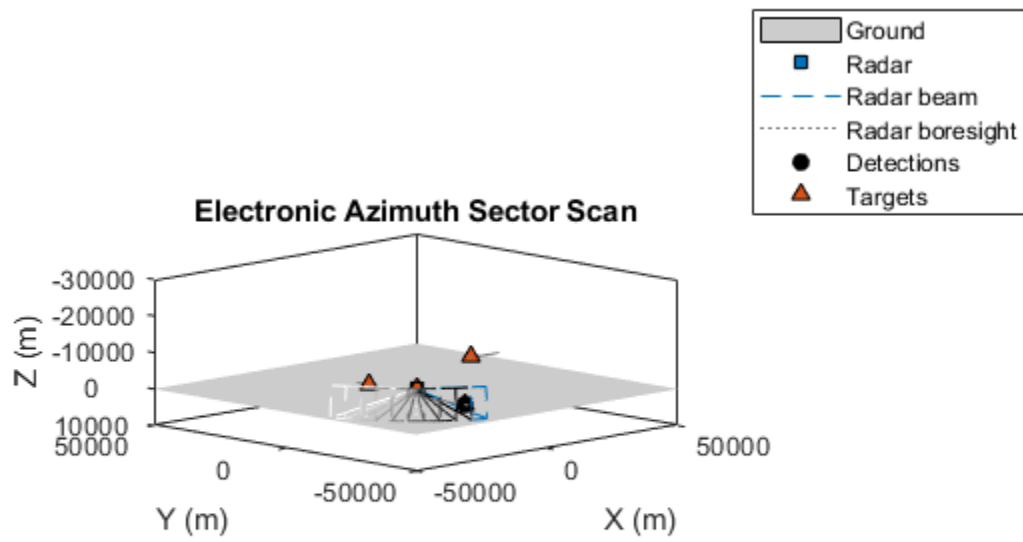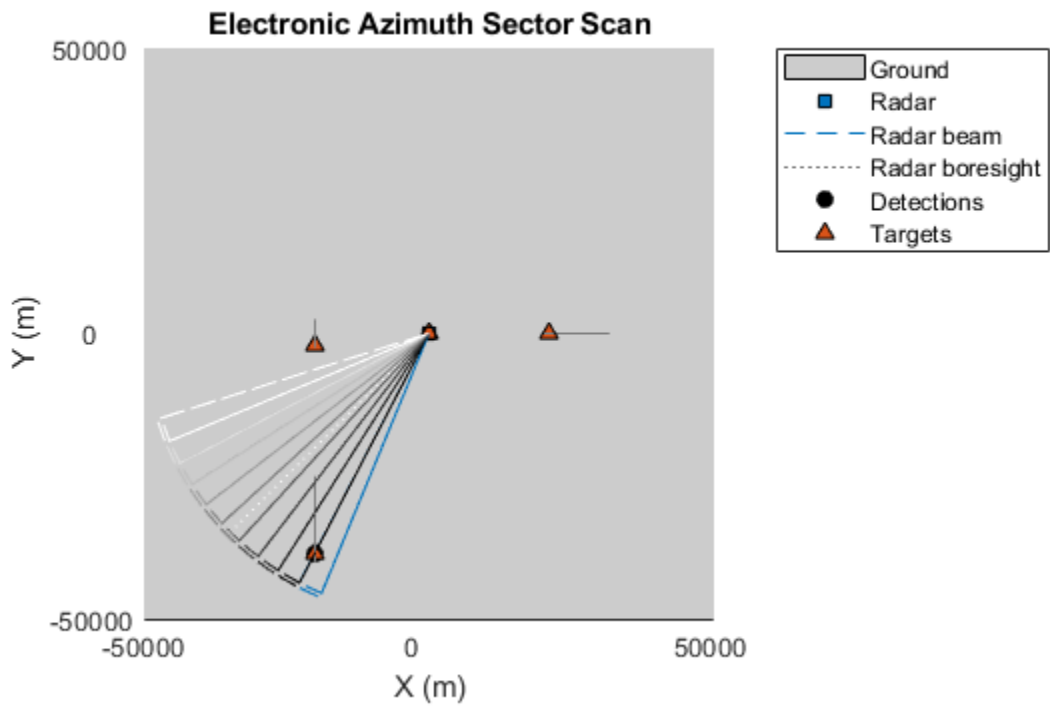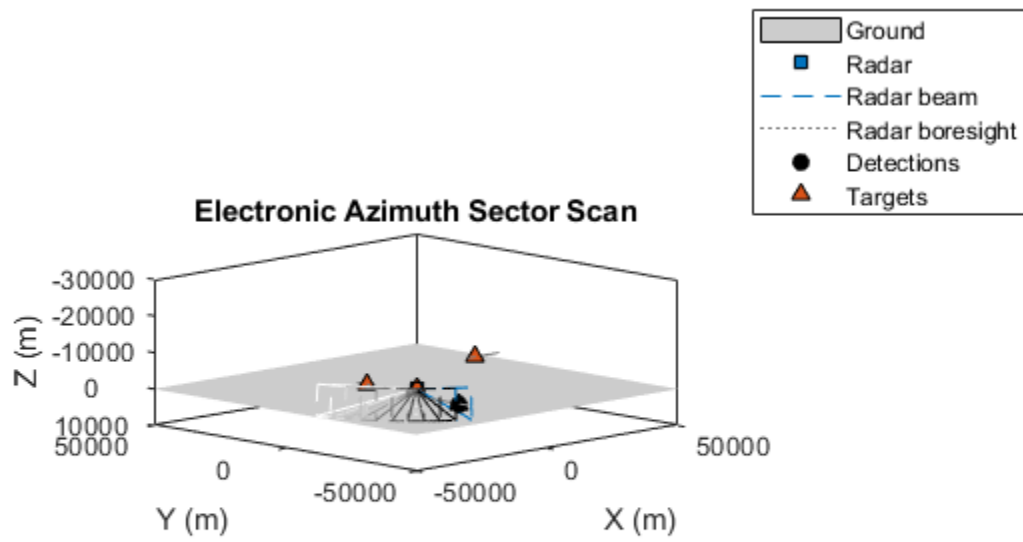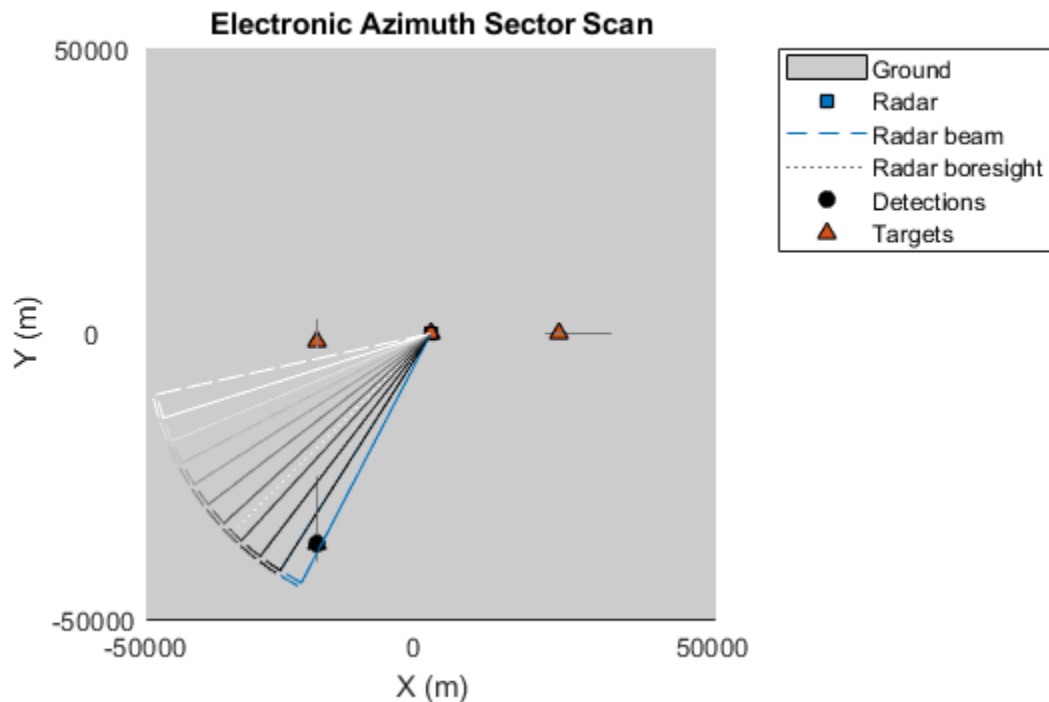
The preceding figures show that electronic sector scans always scan in the same direction (in this case, from the left side of the figure to the right side of the figure). Unlike mechanical scanning, where the next beam position is constrained by the antenna's current mechanical position, electronic scanning can instantaneously move the beams within the scanned sector.

Raster Scan

Mechanical Raster Scan

The radar can be easily configured to execute a mechanical raster scan pattern as follows.

```
radar = monostaticRadarSensor(1, 'Raster','MountingAngles', [-135 0 0], 'MountingLocation', [0 0

% Set update rate to show multiple cycles of the raster scan pattern
radar.UpdateRate = updaterate;

% Set the radar's azimuth field of view to 5 degrees to display larger beams
radar.FieldOfView(1) = 5;

% Run the simulation
helperScanRadarModesExample('Run simulation',ts,platform,radar,7);
```
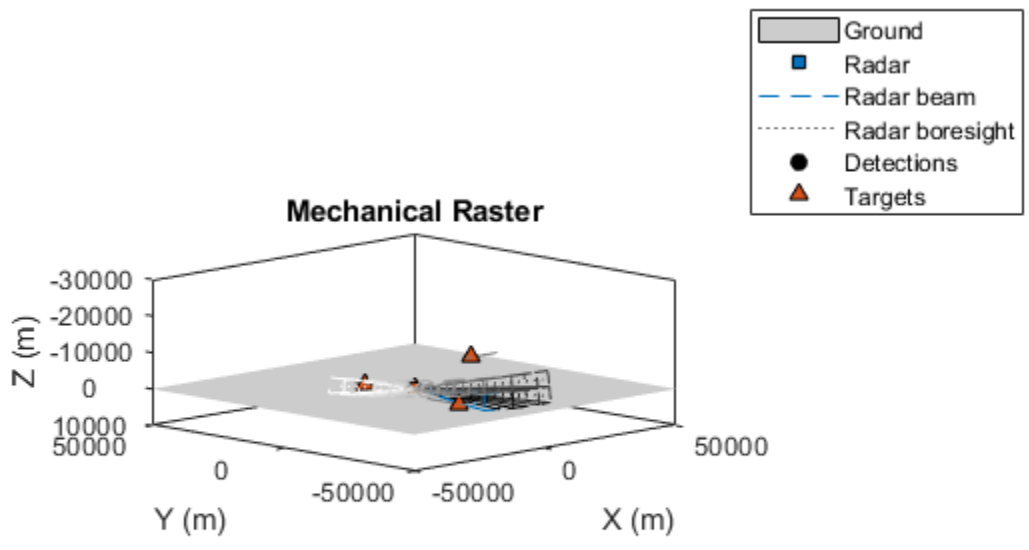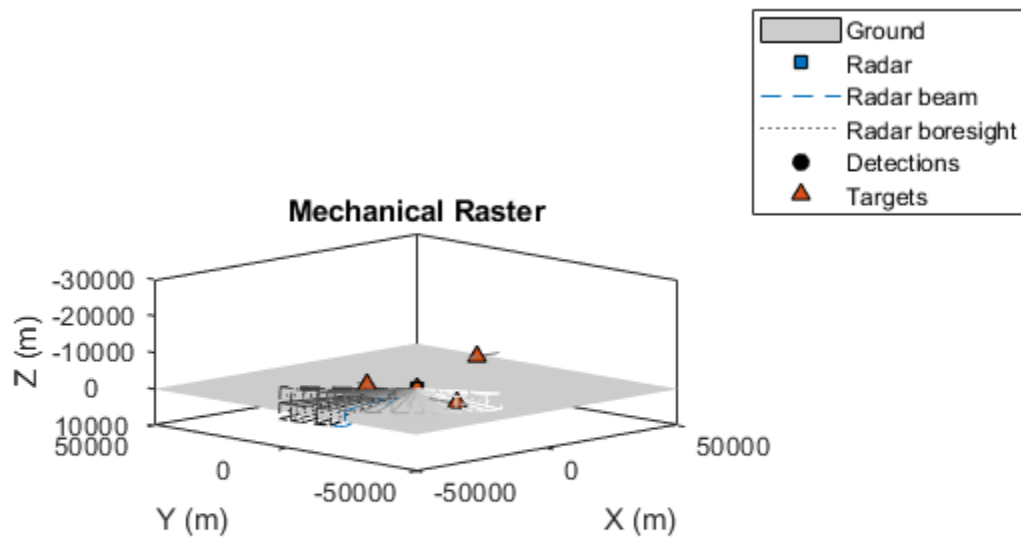
Mechanical Raster

The preceding figures show the radar's beam position along each of the radar's 3 elevation raster scan bars. At the end of each azimuth scan, the radar steps in elevation by its field of view and reverses the direction of its azimuth scan. When an elevation scan limit is reached, the radar begins a new raster scan by reversing the direction in which it steps the beams in elevation.

You can use this configuration as a starting point and adjust the mechanical scan limits and field of view to match the scan pattern for the radar you wish to model.

Electronic Raster Scan

You can also configure the radar to perform an electronic raster scan pattern. Electronic raster scan patterns immediately repeat the same scan sequence after each scan, but a mechanical raster reverses its scan sequence to return the antenna's mechanical position back to its origin.

Configure the radar to perform an electronic raster scan by setting its scan mode to 'Electronic'.

```
radar = monostaticRadarSensor(1,'Raster','ScanMode','Electronic','MountingAngles',[-135 0 0],'Mou

% Set update rate to show multiple cycles of the raster scan pattern
radar.UpdateRate = updaterate;

% Set the radar's azimuth field of view to 5 degrees to display larger beams
radar.FieldOfView(1) = 5;

% Run the simulation
helperScanRadarModesExample('Run simulation',ts,platform,radar,8);
```
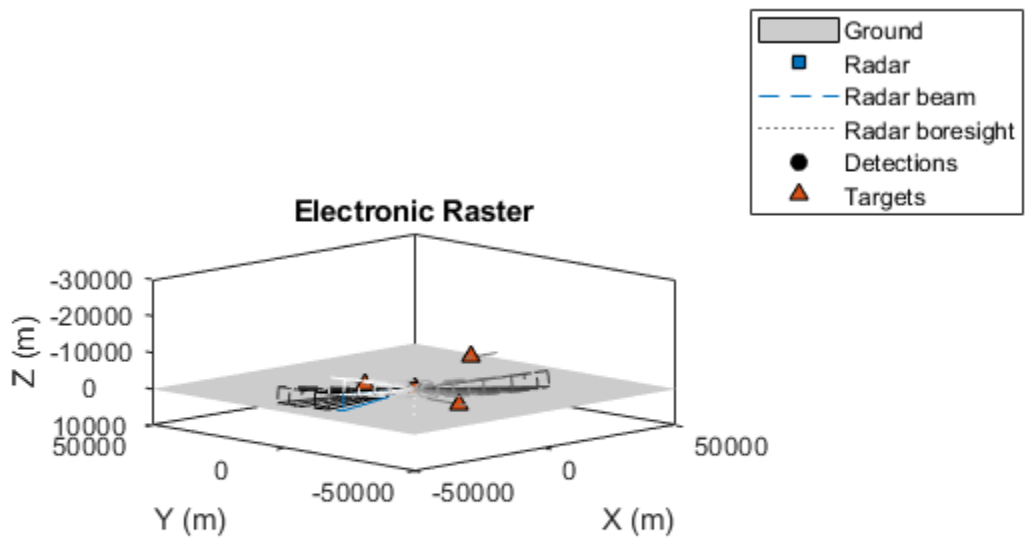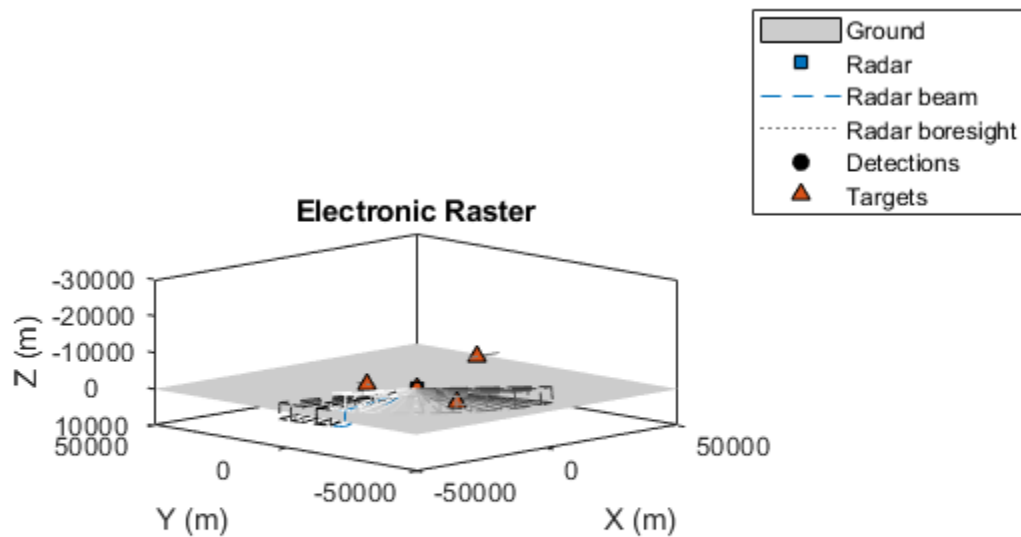
Electronic Raster

The preceding figures show the radar's beam position along each of the radar's 3 elevation raster scan bars. At the end of each azimuth scan, the radar steps in elevation by its field of view and continues scanning in azimuth in the same direction as the previous scan. When an elevation scan limit is reached, the radar repeats the same sequence of raster scan positions. With electronic scanning, the radar can instantaneously return to the beginning of the scan pattern and is not constrained by the current mechanical position of the radar.

You can use this configuration as a starting point and adjust the electronic scan limits and field of view to match the scan pattern for the radar you wish to model.

Summary

This example shows you how to model many common scan patterns using the `monostaticRadarSensor`. This model provides presets that enable you to quickly configure the radar and make additional minor adjustments to the scan parameters to match the radar you are modeling.

Extended Object Tracking of Highway Vehicles with Radar and Camera

This example shows you how to track highway vehicles around an ego vehicle. Vehicles are extended objects, whose dimensions span multiple sensor resolution cells. As a result, the sensors report multiple detections of these objects in a single scan. In this example, you will use different extended object tracking techniques to track highway vehicles and evaluate the results of their tracking performance.

Introduction

In conventional tracking approaches such as global nearest neighbor (`multiObjectTracker`, `trackerGNN`), joint probabilistic data association (`trackerJPDA`) and multi-hypothesis tracking (`trackerTOMHT`), tracked objects are assumed to return one detection per sensor scan. With the development of sensors that have better resolution, such as a high-resolution radar, the sensors typically return more than one detection of an object. For example, the image below depicts multiple detections for a single vehicle that spans multiple radar resolution cells. In such cases, the technique used to track the objects is known as extended object tracking [1].
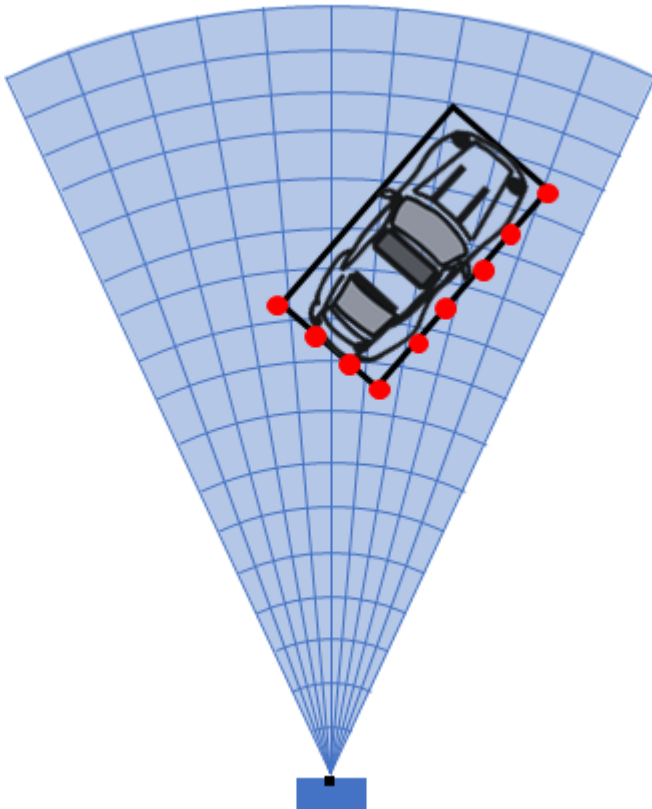
The key benefit of using a high-resolution sensor is getting more information about the object, such as its dimensions and orientation. This additional information can improve the probability of detection and reduce the false alarm rate.

Extended objects present new challenges to conventional trackers, because these trackers assume a single detection per object per sensor. In some cases, you can cluster the sensor data to provide the

conventional trackers with a single detection per object. However, by doing so, the benefit of using a high-resolution sensor may be lost.

In contrast, extended object trackers can handle multiple detections per object. In addition, these trackers can estimate not only the kinematic states, such as position and velocity of the object, but also the dimensions and orientation of the object. In this example, you track vehicles around the ego vehicle using the following trackers:

- A conventional multi-object tracker using a point-target model, `multiObjectTracker`
- A GGIW-PHD (Gamma Gaussian Inverse Wishart PHD) tracker, `trackerPHD` with `ggiwphd` filter
- A GM-PHD (Gaussian mixture PHD) tracker, `trackerPHD` with `gmphd` filter using rectangular target model

You will evaluate the tracking results of all trackers using `trackErrorMetrics` and `trackAssignmentMetrics`, which provide multiple measures of effectiveness of a tracker. You will also evaluate the results using the Optimal SubPattern Assignment Metric (OSPA), `trackOSPAMetric`, which aims to evaluate the performance of a tracker using a combined score.

Setup

Scenario

In this example, there is an ego vehicle and four other vehicles: a vehicle ahead of the ego vehicle in the center lane, a vehicle behind the ego vehicle in the center lane, a truck ahead of the ego vehicle in the right lane and an overtaking vehicle in the left lane.

In this example, you simulate an ego vehicle that has 6 radar sensors and 2 vision sensors covering the 360-degree field of view. The sensors have some overlap and some coverage gap. The ego vehicle is equipped with a long-range radar sensor and a vision sensor on both the front and back of the vehicle. Each side of the vehicle has two short-range radar sensors, each covering 90 degrees. One sensor on each side covers from the middle of the vehicle to the back. The other sensor on each side covers from the middle of the vehicle forward.

```
% Create the scenario
exPath = fullfile(matlabroot,'examples','driving_fusion','main');
addpath(exPath)
[scenario, egoVehicle, sensors] = helperCreateScenario;

% Create the display object
display = helperExtendedTargetTrackingDisplay;

% Create the Animation writer to record each frame of the figure for
% animation writing. Set 'RecordGIF' to true to enable GIF writing.
gifWriter = helperGIFWriter('Figure',display.Figure,...
    'RecordGIF',false);
```
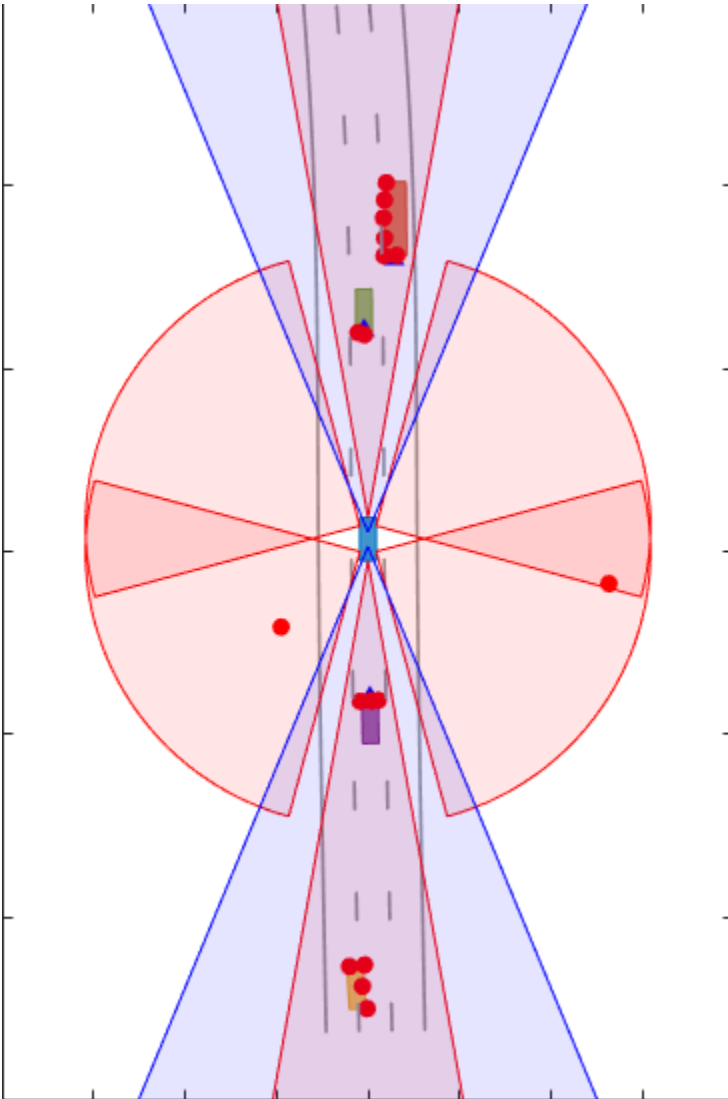
Metrics

In this example, you use some key metrics to assess the tracking performance of each tracker. In particular, you assess the trackers based on their accuracy in estimating the positions, velocities, dimensions (length and width) and orientations of the objects. These metrics can be evaluated using the `trackErrorMetrics` class. To define the error of a tracked target from its ground truth, this example uses a 'custom' error function, `helperExtendedTargetError`, listed at the end of this example.

You will also assess the performance based on metrics such as number of false tracks or redundant tracks. These metrics can be calculated using the `trackAssignmentMetrics` class. To define the distance between a tracked target and a truth object, this example uses a 'custom' error function, `helperExtendedTargetDistance`, listed at the end of this example. The function defines the distance metric as the sum of distances in position, velocity, dimension and yaw.

`trackErrorMetrics` and `trackAssignmentMetrics` provide multiple measures of effectiveness of a tracking algorithm. You will also assess the performance based on the Optimal SubPattern

Assignment Metric (OSPA), which provides a single score value for the tracking algorithm at each time step. This metric can be calculated using the `trackOSPAMetric` class. The 'custom' distance function defined for OSPA is same as the assignment metrics.

```
% actorProfiles to provide the error function size information about each
% truth.
aProfiles = actorProfiles(scenario);

% Function to return the errors given track and truth.
errorFcn = @(track,truth)helperExtendedTargetError(track,truth,aProfiles);

% Function to return the distance between track and truth
distFcn = @(track,truth)helperExtendedTargetDistance(track,truth,aProfiles);

% Function to return the IDs from the ground truth. The default
% identifier assumes that the truth is identified with PlatformID. In
% drivingScenario, truth is identified with an ActorID.
truthIdFcn = @(x)[x.ActorID];

% Create metrics object.
tem = trackErrorMetrics(...
    'ErrorFunctionFormat','custom',...
    'EstimationErrorLabels',{'PositionError','VelocityError','DimensionsError','YawError'},...
    'EstimationErrorFcn',errorFcn,...
    'TruthIdentifierFcn',truthIdFcn);

tam = trackAssignmentMetrics(...
    'DistanceFunctionFormat','custom',...
    'AssignmentDistanceFcn',distFcn,...
    'DivergenceDistanceFcn',distFcn,...
    'TruthIdentifierFcn',truthIdFcn,...
    'AssignmentThreshold',30,...
    'DivergenceThreshold',35);

% Create ospa metric object
tom = trackOSPAMetric(...
    'Distance','custom',...
    'DistanceFcn',distFcn,...
    'TruthIdentifierFcn',truthIdFcn);
```

Point Object Tracker

The `multiObjectTracker` System object™ assumes one detection per object per sensor and uses a global nearest neighbor approach to associate detections to tracks. It assumes that every object can be detected at most once by a sensor in a scan. In this case, the simulated radar sensors have a high enough resolution to generate multiple detections per object. If these detections are not clustered, the tracker generates multiple tracks per object. Clustering returns one detection per cluster, at the cost of having a larger uncertainty covariance and losing information about the true object dimensions. Clustering also makes it hard to distinguish between two objects when they are close to each other, for example, when one vehicle passes another vehicle.

```
trackerRunTimes = zeros(0,3);
ospaMetric = zeros(0,3);

% Create a multiObjectTracker
tracker = multiObjectTracker(...
    'FilterInitializationFcn', @helperInitPointFilter, ...
```

```
        'AssignmentThreshold', 30, ...
        'ConfirmationThreshold', [4 5], ...
        'DeletionThreshold', 3);

% Reset the random number generator for repeatable results
seed = 2018;
S = rng(seed);
timeStep = 1;

% For multiObjectTracker, the radar reports in Ego Cartesian frame and does
% not report velocity. This allows us to cluster detections from multiple
% sensors.
for i = 1:6
    sensors{i}.HasRangeRate = false;
    sensors{i}.DetectionCoordinates = 'Ego Cartesian';
end
```

Run the scenario

```
while advance(scenario) && ishghandle(display.Figure)
    % Get the scenario time
    time = scenario.SimulationTime;

    % Collect detections from the ego vehicle sensors
    [detections,isValidTime] = helperDetect(sensors, egoVehicle, time);

    % Update the tracker if there are new detections
    if any(isValidTime)
        % Detections must be clustered first for the point tracker
        detectionClusters = helperClusterRadarDetections(detections);

        % Update the tracker
        tic
        % confirmedTracks are in scenario coordinates
        confirmedTracks = updateTracks(tracker, detectionClusters, time);
        t = toc;

        % Update the metrics
        % a. Obtain ground truth
        groundTruth = scenario.Actors(2:end); % All except Ego

        % b. Update assignment metrics
        tam(confirmedTracks,groundTruth);
        [trackIDs,truthIDs] = currentAssignment(tam);

        % c. Update error metrics
        tem(confirmedTracks,trackIDs,groundTruth,truthIDs);

        % d. Update ospa metric
        ospaMetric(timeStep,1) = tom(confirmedTracks, groundTruth);

        % Update bird's-eye-plot
        % Convert tracks to ego coordinates for display
        confirmedTracksEgo = helperConvertToEgoCoordinates(egoVehicle, confirmedTracks);
        display(egoVehicle, sensors, detections, confirmedTracksEgo, detectionClusters);
        drawnow;

        % Record tracker run times
```

```
                    trackerRunTimes(timeStep,1) = t;
                    timeStep = timeStep + 1;

                    % Capture frames for animation
                    gifWriter();
                end
            end

    % Capture the cumulative track metrics. The error metrics show the averaged
    % value of the error over the simulation.
    assignmentMetricsMOT = tam.trackMetricsTable;
    errorMetricsMOT = tem.cumulativeTruthMetrics;

    % Write GIF if requested
    writeAnimation(gifWriter,'multiObjectTracking');
```

These results show that, with clustering, the tracker can keep track of the objects in the scene. However, it also shows that the track associated with the overtaking vehicle (yellow) moves from the front of the vehicle at the beginning of the scenario to the back of the vehicle at the end. At the beginning of the scenario, the overtaking vehicle is behind the ego vehicle (blue), so radar and vision detections are made from its front. As the overtaking vehicle passes the ego vehicle, radar detections are made from the side of the overtaking vehicle and then from its back, and the track moves to the back of the vehicle.

You can also see that the clustering is not perfect. When the passing vehicle passes the vehicle that is behind the ego vehicle (purple), both tracks are slightly shifted to the left due to the imperfect clustering. A redundant track is created on the track initially due to multiple clusters created when part of the side edge is missed. Also, a redundant track appears on the passing vehicle during the end because the distances between its detections increase.
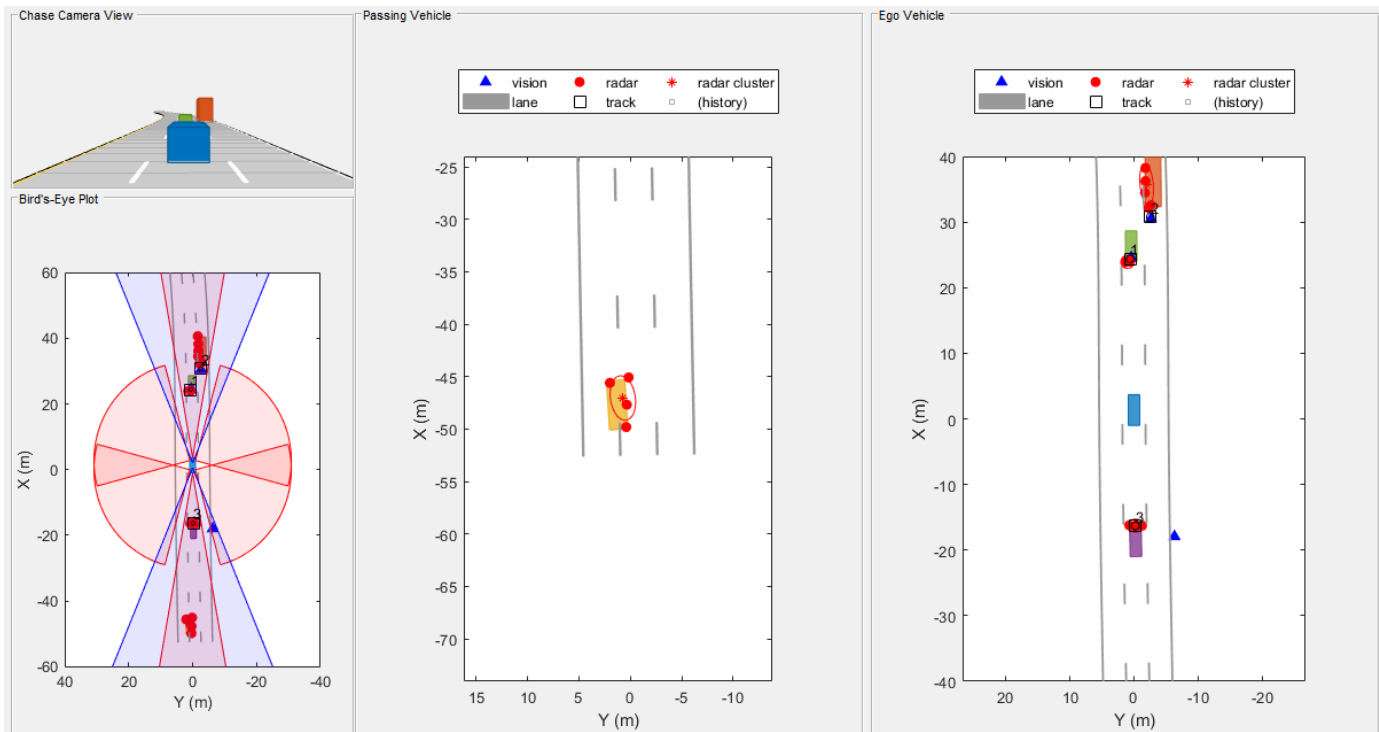
GGIW-PHD Extended Object Tracker

In this section, you use a GGIW-PHD tracker (`trackerPHD` with `ggiwphd`) to track objects. Unlike `multiObjectTracker`, which uses one filter per track, the GGIW-PHD is a multi-target filter which describes the probability hypothesis density (PHD) of the scenario. To model the extended target, GGIW-PHD uses the following distributions:

Gamma: Positive value to describe expected number of detections.

Gaussian: State vector to describe target's kinematic state

Inverse-Wishart: Positive-definite matrix to describe the elliptical extent.

The model assumes that each distribution is independent of each other. Thus, the probability hypothesis density (PHD) in GGIW-PHD filter is described by a weighted sum of the probability density functions of several GGIW components.

A PHD tracker requires calculating the detectability of each component in the density. The calculation of detectability requires configurations of each sensor used with the tracker. You define these configurations for `trackerPHD` using the `trackingSensorConfiguration` class. Review the `helperCreateSensorConfigurations` function to see how sensor properties can be utilized to define the sensor configurations for the tracker.

```
% Set up sensor configurations
%
sensorConfigurations = helperCreateSensorConfigurations(sensors,egoVehicle);

% The transform function and filter initialization functions are state and
% filter dependent. Therefore, they are not set in the helper function.
for i = 1:numel(sensorConfigurations)
    % You can use a different technique to initialize a filter for each
    % sensor by using a different function for each configuration.
    sensorConfigurations{i}.FilterInitializationFcn = @helperInitGGIWFilter;

    % Tracks are defined in constant turn-rate state-space in the scenario
    % coordinates. The MeasurementFcn for constant turn-rate model can be
    % used as the transform function.
    sensorConfigurations{i}.SensorTransformFcn = @ctmeas;
end
```

Define the tracker.

In contrast to a point object tracker, which usually takes into account one partition (cluster) of detections, the trackerPHD creates multiple possible partitions of a set of detections and evaluate it against the current components in the PHD filter. The 3 and 5 in the function below defines the lower and upper Mahalanobis distance between detections. This is equivalent to defining that each cluster of detection must be a minimum of 3 resolutions apart and maximum of 5 resolutions apart from each other.

```
partFcn = @(x)partitionDetections(x,3,5);

tracker = trackerPHD('SensorConfigurations', sensorConfigurations,...
    'PartitioningFcn',partFcn,...
    'AssignmentThreshold',220,...% Minimum negative log-likelihood of a detection cell (multiple
    'ExtractionThreshold',0.8,...% Weight threshold of a filter component to be declared a track
    'ConfirmationThreshold',0.95,...% Weight threshold of a filter component to be declared a con
```

```
                'MergingThreshold',50,...% Threshold to merge components
                'HasSensorConfigurationsInput',true... % Tracking is performed in scenario frame and hence se
                );
```

Run the simulation.

```
% Release and restart all objects.
restart(scenario);
release(tem);
release(tam);
% No penality for trackerPHD
tam.AssignmentThreshold = tam.AssignmentThreshold - 2;
release(display);
display.PlotClusteredDetection = false;
gifWriter.pFrames = {};
for i = 1:numel(sensors)
    release(sensors{i});
    if i <= 6
        sensors{i}.HasRangeRate = true;
        sensors{i}.DetectionCoordinates = 'Sensor spherical';
    end
end

% Restore random seed.
rng(seed)

% First time step
timeStep = 1;

% Run the scenario
while advance(scenario) && ishghandle(display.Figure)
    % Get the scenario time
    time = scenario.SimulationTime;

    % Get the poses of the other vehicles in ego vehicle coordinates
    ta = targetPoses(egoVehicle);

    % Collect detections from the ego vehicle sensors
    [detections, isValidTime, configurations] = helperDetect(sensors, egoVehicle, time, sensorCon

    % Update the tracker with all the detections. Note that there is no
    % need to cluster the detections before passing them to the tracker.
    % Also, the sensor configurations are passed as an input to the
    % tracker.
    tic
    % confirmedTracks are in scenario coordinates
    confirmedTracks = tracker(detections,configurations,time);
    t = toc;

    % Update the metrics
    % a. Obtain ground truth
    groundTruth = scenario.Actors(2:end); % All except Ego

    % b. Update assignment metrics
    tam(confirmedTracks,groundTruth);
    [trackIDs,truthIDs] = currentAssignment(tam);

    % c. Update error metrics
```

```
        tem(confirmedTracks,trackIDs,groundTruth,truthIDs);

        % d. Update ospa metric
        ospaMetric(timeStep,2) = tom(confirmedTracks, groundTruth);

        % Update the bird's-eye plot
        % Convert tracks to ego coordinates for display
        confirmedTracksEgo = helperConvertToEgoCoordinates(egoVehicle, confirmedTracks);
        display(egoVehicle, sensors, detections, confirmedTracksEgo);
        drawnow;

        % Record tracker run times
        trackerRunTimes(timeStep,2) = t;
        timeStep = timeStep + 1;

        % Capture frames for GIF
        gifWriter();
end

% Capture the truth and track metrics tables
assignmentMetricsGGIWPHD = tam.trackMetricsTable;
errorMetricsGGIWPHD = tem.cumulativeTruthMetrics;

% Write GIF if requested
writeAnimation(gifWriter,'ggiwphdTracking');
```

These results show that the GGIW-PHD can handle multiple detections per object per sensor, without the need to cluster these detections first. Moreover, by using the multiple detections, the tracker estimates the position, velocity, dimensions and orientation of each object. The dashed elliptical shape in the figure demonstrates the expected extent of the target. The filter initialization function specifies multiple possible sizes and their relative weights using multiple components. The list can be expanded to add more sizes with added computational complexity. In contrast, you can also initialize one component per detection with a higher uncertainty in dimensions. This will enable the tracker to estimate the dimensions of the objects automatically. That said, the accuracy of the estimate will depend on the observability of the target dimensions and is susceptible to shrinkage and enlargement of track dimensions as the targets move around the ego vehicle.

The GGIW-PHD filter assumes that detections are distributed around the target's elliptical center. Therefore, the tracks tend to follow observable portions of the vehicle. Such observable portions include rear face of the vehicle that is directly ahead of the ego vehicle or the front face of the vehicle directly behind the ego vehicle for example, the rear and front face of the vehicle directly ahead and behind of the ego vehicle respectively. In contrast, the length and width of the passing vehicle was fully observed during the simulation. Therefore, its estimated ellipse has a better overlap with the actual shape.
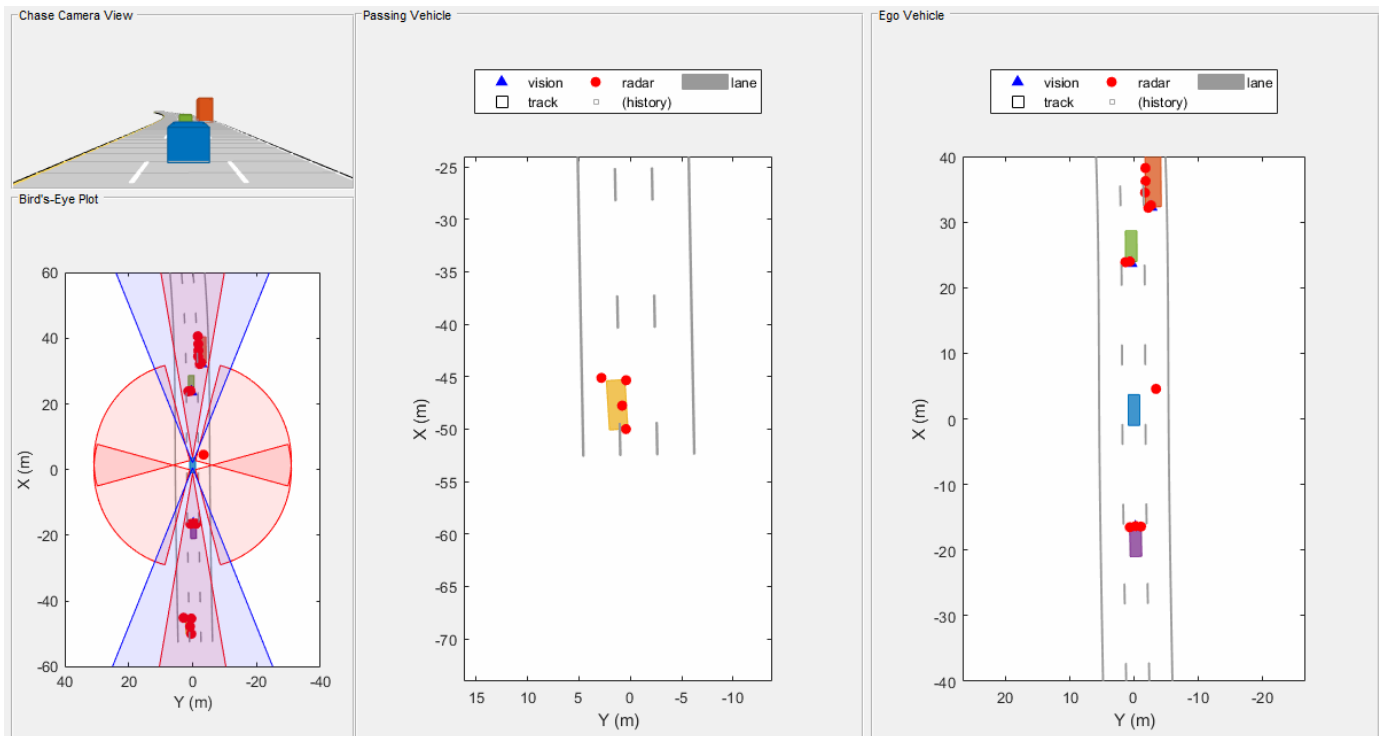
GM-PHD Rectangular Object Tracker

In this section, you use a GM-PHD tracker (`trackerPHD` with `gmphd`) and a rectangular target model (`initctrectgmphd`) to track objects. Unlike `ggiwphd`, which uses an elliptical shape to track extent, `gmphd` allows you to use a Gaussian distribution to define the shape of your choice. The rectangular target model is defined by motion models, `ctrect` and `ctrectjac` and measurement models, `ctrectmeas` and `ctrectmeasjac`.

The sensor configurations defined for trackerPHD earlier remain the same, except for definition of `SensorTransformFcn` and `FilterInitializationFcn`.

```
for i = 1:numel(sensorConfigurations)
    sensorConfigurations{i}.FilterInitializationFcn = @helperInitRectangularFilter; % Initialize
    sensorConfigurations{i}.SensorTransformFcn = @ctrectcorners; % Use corners to calculate dete
end

% Define tracker using new sensor configurations
tracker = trackerPHD('SensorConfigurations', sensorConfigurations,...
    'PartitioningFcn',partFcn,...
    'AssignmentThreshold',220,...% Minimum negative log-likelihood of a detection cell to add bir
    'ExtractionThreshold',0.8,...% Weight threshold of a filter component to be declared a track
    'ConfirmationThreshold',0.95,...% Weight threshold of a filter component to be declared a con
    'MergingThreshold',50,...% Threshold to merge components
    'HasSensorConfigurationsInput',true... % Tracking is performed in scenario frame and hence se
    );

% Release and restart all objects.
restart(scenario);
for i = 1:numel(sensors)
    release(sensors{i});
end
```

```
release(tem);
release(tam);
release(display);
display.PlotClusteredDetection = false;
gifWriter.pFrames = {};

% Restore random seed.
rng(seed)

% First time step
timeStep = 1;

% Run the scenario
while advance(scenario) && ishghandle(display.Figure)
    % Get the scenario time
    time = scenario.SimulationTime;

    % Get the poses of the other vehicles in ego vehicle coordinates
    ta = targetPoses(egoVehicle);

    % Collect detections from the ego vehicle sensors
    [detections, isValidTime, configurations] = helperDetect(sensors, egoVehicle, time, sensorCon

    % Update the tracker with all the detections. Note that there is no
    % need to cluster the detections before passing them to the tracker.
    % Also, the sensor configurations are passed as an input to the
    % tracker.
    tic
    % confirmedTracks are in scenario coordinates
    confirmedTracks = tracker(detections,configurations,time);
    t = toc;

    % Update the metrics
    % a. Obtain ground truth
    groundTruth = scenario.Actors(2:end); % All except Ego

    % b. Update assignment metrics
    tam(confirmedTracks,groundTruth);
    [trackIDs,truthIDs] = currentAssignment(tam);

    % c. Update error metrics
    tem(confirmedTracks,trackIDs,groundTruth,truthIDs);

    % d. Update ospa metric
    ospaMetric(timeStep,3) = tom(confirmedTracks, groundTruth);

    % Update the bird's-eye plot
    % Convert tracks to ego coordinates for display
    confirmedTracksEgo = helperConvertToEgoCoordinates(egoVehicle, confirmedTracks);
    display(egoVehicle, sensors, detections, confirmedTracksEgo);
    drawnow;

    % Record tracker run times
    trackerRunTimes(timeStep,3) = t;
    timeStep = timeStep + 1;

    % Capture frames for GIF
    gifWriter();
```

```
end

% Capture the truth and track metrics tables
assignmentMetricsGMPHD = tam.trackMetricsTable;
errorMetricsGMPHD = tem.cumulativeTruthMetrics;

% Write GIF if requested
writeAnimation(gifWriter,'gmphdTracking');

% Return the random number generator to its previous state
rng(S)
rmpath(exPath)
```

These results show that the GM-PHD can also handle multiple detections per object per sensor. Similar to GGIW-PHD, it also estimates the size and orientation of the object. The filter initialization function uses similar approach as GGIW-PHD tracker and initializes multiple components for different sizes.

You can notice that the estimated tracks, which are modeled as rectangles, have a good fit with the simulated ground truth object, depicted by the solid color patches. In particular, the tracks are able to correctly track the shape of the vehicle along with the kinematic center.
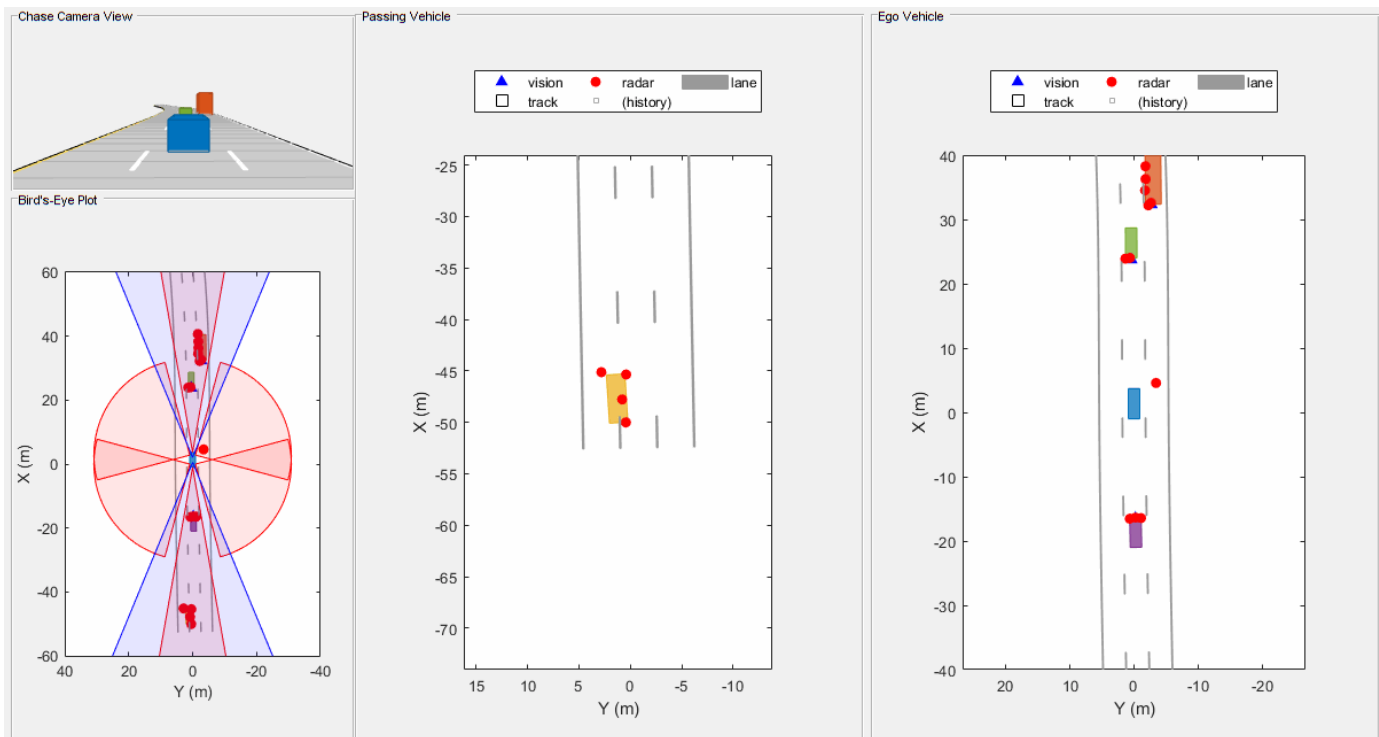
Evaluate Tracking Performance

Evaluate the tracking performance of each tracker using quantitative metrics such as the estimation error in position, velocity, dimensions and orientation. Also evaluate the track assignments using metrics such as redundant and false tracks.

Assignment metrics

```
helperPlotAssignmentMetrics(assignmentMetricsMOT, assignmentMetricsGGIWPHD, assignmentMetricsGMPH
```
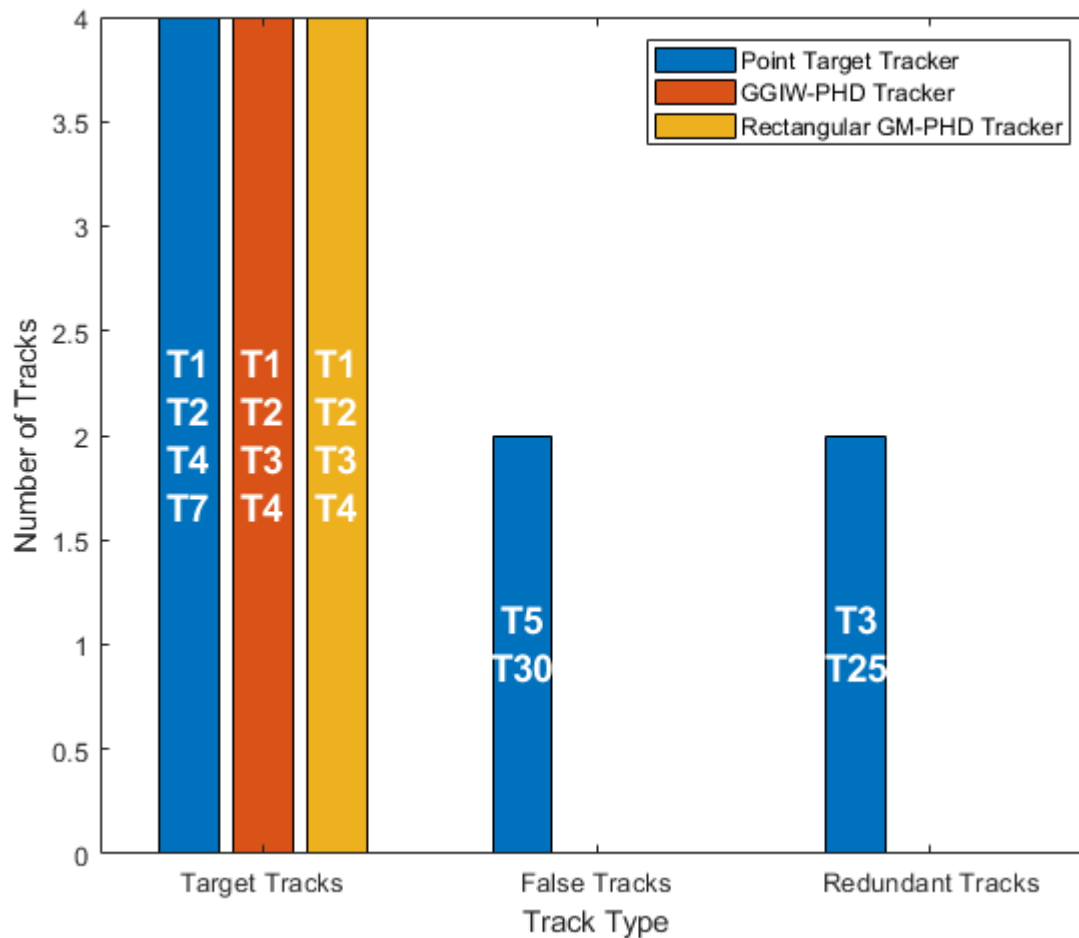
The assignment metrics illustrate that redundant and false tracks were initialized and confirmed by the point object tracker. These tracks result due to imperfect clustering, where detections belonging to the same target were clustered into more than one clustered detection. In contrast, the GGIW-PHD tracker and the GM-PHD tracker maintain tracks on all four targets and do not create any false or redundant tracks. These metrics show that both extended object trackers correctly partition the detections and associate them with the correct tracks.

Error metrics

```
helperPlotErrorMetrics(errorMetricsMOT, errorMetricsGGIWPHD, errorMetricsGMPHD);
```
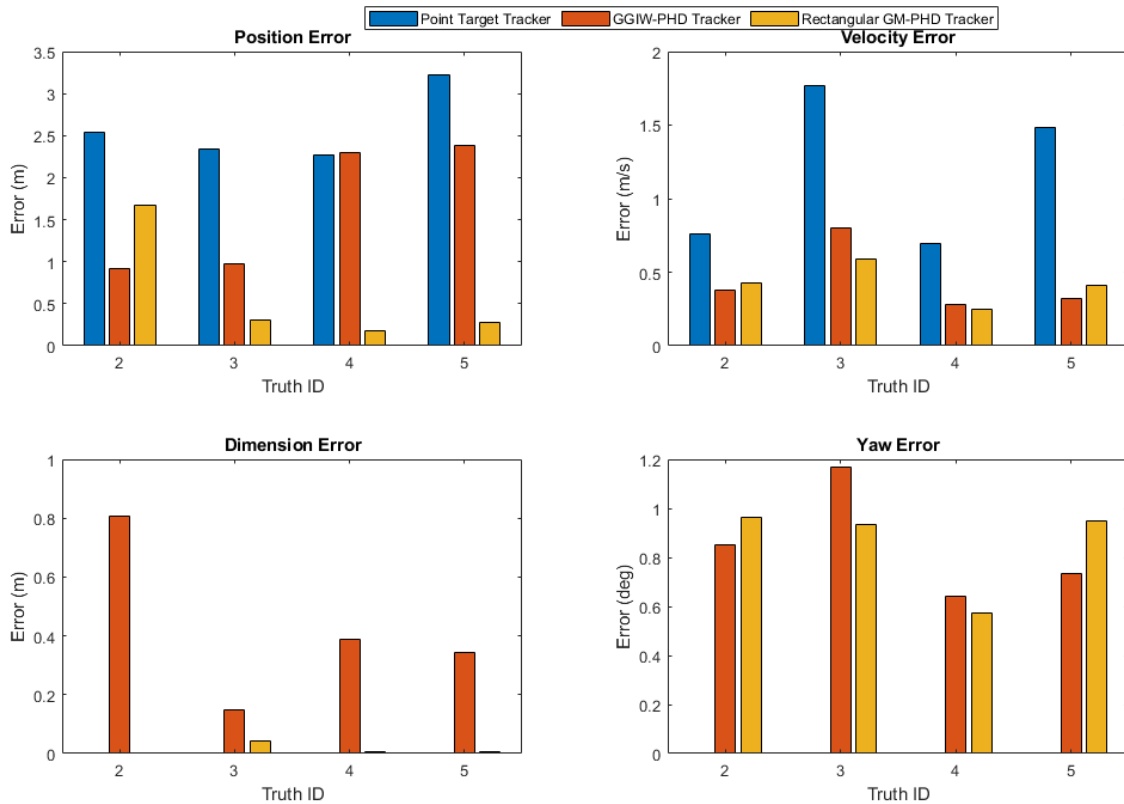
The plot shows the average estimation errors for the three types of trackers used in this example. Because the point object tracker does not estimate the yaw and dimensions of the objects, they are now shown in the plots. The point object tracker is able to estimate the kinematics of the objects with a reasonable accuracy. The position error of the vehicle behind the ego vehicle is higher because it was dragged to the left when the passing vehicle overtakes this vehicle. This is also an artifact of imperfect clustering when the objects are close to each other.

As described earlier, the GGIW-PHD tracker assumes that measurements are distributed around the object's extent, which results in center of the tracks on observable parts of the vehicle. This can also be seen in the position error metrics for TruthID 2 and 4. The tracker is able to estimate the dimensions of the object with about 0.3 meters accuracy for the vehicles ahead and behind the ego vehicle. Because of higher certainty defined for the vehicles' dimensions in the `helperInitGGIWFilter` function, the tracker does not collapse the length of these vehicles, even when the best-fit ellipse has a very low length. As passing vehicle (TruthID 3) was observed on all dimensions, its dimensions are measured more accurately than the other vehicles. However, as the passing vehicle maneuvers with respect to the ego vehicle, the error in yaw estimate is higher.

The GM-PHD in this example uses a rectangular shaped target model and uses received measurements to evaluate expected measurements on the boundary of the target. This model helps the tracker estimate the shape and orientation more accurately. However, the process of evaluating expected measurements on the edges of a rectangular target, is computationally more expensive.

OSPA Metric

As described earlier, the OSPA metric aims to describe the performance of a tracking algorithm using a single score. Notice that the OSPA sufficiently captures the performance of the tracking algorithm which decreases from GM-PHD to GGIW-PHD to the point-target tracker, as described using the error and assignment metrics.

```
ospaFig = figure;
plot(ospaMetric,'LineWidth',2);
legend('Point Target Tracker','GGIW-PHD Tracker','Rectangular GM-PHD Tracker');
xlabel('Time step (k)');
ylabel('OSPA');
```
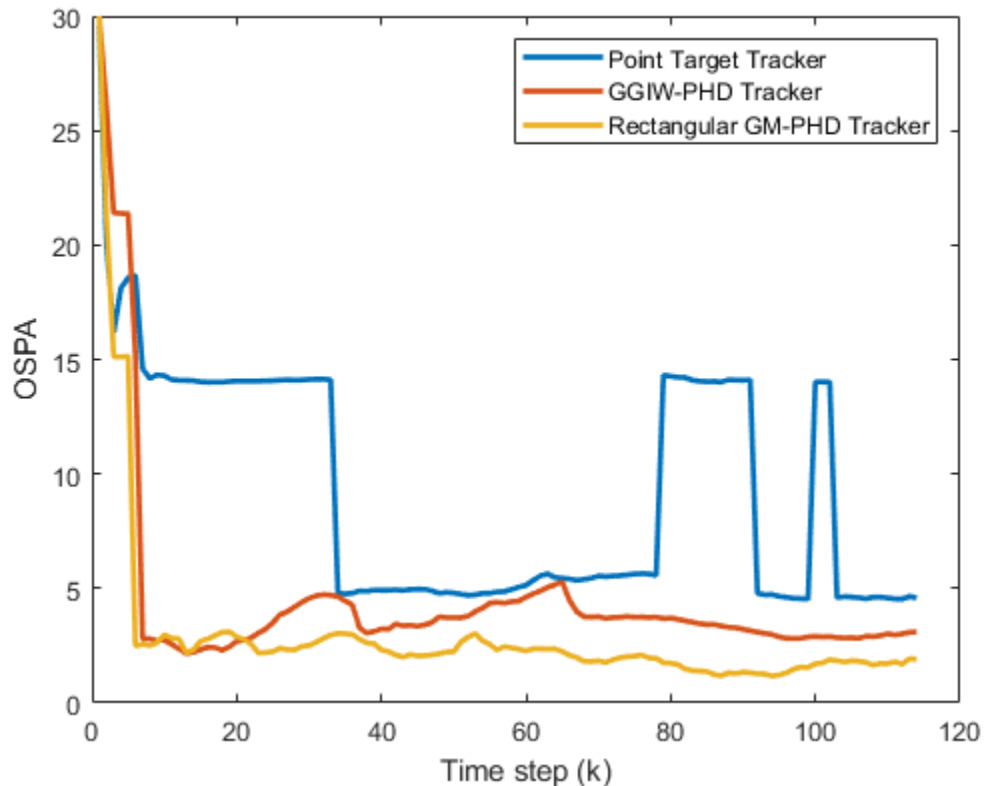

Compare Time Performance

Previously, you learned about different techniques, the assumptions they make about target models, and the resulting tracking performance. Now compare the run-times of the trackers. Notice that GGIW-PHD filter offers significant computational advantages over the GM-PHD, at the cost of decreased tracking performance.

```
runTimeFig = figure;
h = plot(trackerRunTimes(3:end,:)./trackerRunTimes(3:end,1),'LineWidth',2);
legend('Point Target Tracker','GGIW-PHD Tracker','Rectangular GM-PHD Tracker');
xlabel('Time step (k)');
ylabel('$$\frac{t_{tracker}}{t_{multiObjectTracker}}$$','interpreter','latex','fontsize',14);
ylim([0 max([h.YData]) + 1]);
```
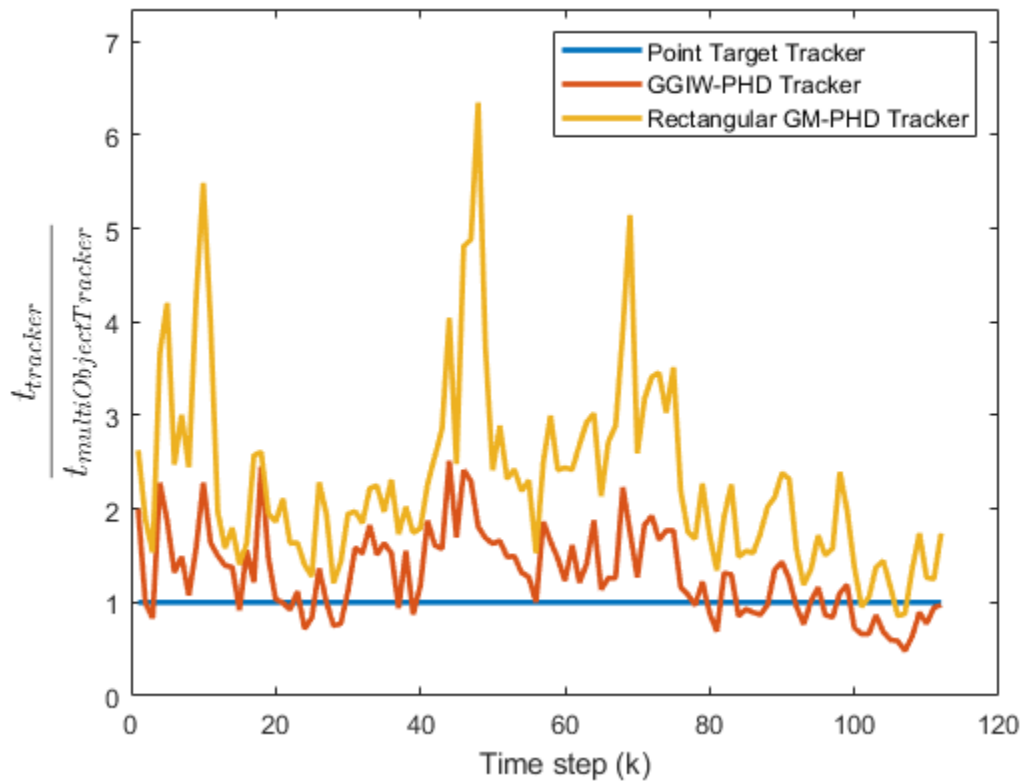
Summary

This example showed how to track objects that return multiple detections in a single sensor scan using different approaches. These approaches can be used to track objects with high-resolution sensors, such as a radar or laser sensor.

References

[1] Granström, Karl, Marcus Baum, and Stephan Reuter. "Extended Object Tracking: Introduction, Overview and Applications." *Journal of Advances in Information Fusion*. Vol. 12, No. 2, December 2017.

[2] Granström, Karl, Christian Lundquist, and Umut Orguner. "Tracking rectangular and elliptical extended targets using laser measurements." 14th International Conference on Information Fusion. IEEE, 2011.

[3] Granström, Karl. "Extended target tracking using PHD filters." 2012

Supporting Functions

helperExtendedTargetError

Function to define the error between tracked target and the associated ground truth.

```
function [posError,velError,dimError,yawError] = helperExtendedTargetError(track,truth,aProfiles
% Errors as a function of target track and associated truth.
```

```matlab
% Get true information from the ground truth.
truePos = truth.Position(1:2)';
% Position is at the rear axle for all vehicles. We would like to compute
% the error from the center of the vehicle
rot = [cosd(truth.Yaw) -sind(truth.Yaw);sind(truth.Yaw) cosd(truth.Yaw)];
truePos = truePos + rot*[truth.Wheelbase/2;0];

trueVel = truth.Velocity(1:2);
trueYaw = truth.Yaw(:);
thisActor = truth.ActorID;
thisProfile = aProfiles([aProfiles.ActorID] == thisActor);
trueDims = [thisProfile.Length;thisProfile.Width];

% Get estimated value from track.
% GGIW-PHD tracker outputs a field 'Extent' and 'SourceIndex'
% GM-PHD tracker outputs 'SourceIndex', but not 'Extent'
% multiObjectTracker does not output 'SourceIndex'

if ~isfield(track,'SourceIndex')
    estPos = track.State([1 3]);
    estVel = track.State([2 4]);
    % No yaw or dimension information in multiObjectTracker.
    estYaw = nan;
    estDims = [nan;nan];
elseif isfield(track,'Extent') % trackerPHD with GGIWPHD
    estPos = track.State([1 3]);
    estVel = track.State([2 4]);
    estYaw = atan2d(estVel(2),estVel(1));
    d = eig(track.Extent);
    dims = 2*sqrt(d);
    estDims = [max(dims);min(dims)];
else % trackerPHD with GMPHD
    estPos = track.State(1:2);
    estYaw = track.State(4);
    estVel = [track.State(3)*cosd(estYaw);track.State(3)*sind(estYaw)];
    estDims = track.State(6:7);
end

% Compute 2-norm of error for each attribute.
posError = norm(truePos(:) - estPos(:));
velError = norm(trueVel(:) - estVel(:));
dimError = norm(trueDims(:) - estDims(:));
yawError = norm(trueYaw(:) - estYaw(:));
end
```

helperExtendedTargetDistance

Function to define the distance between a track and a ground truth.

```matlab
function dist = helperExtendedTargetDistance(track,truth,aProfiles)
% This function computes the distance between track and a truth.

% Copyright 2019 The MathWorks, Inc.

% Errors in each aspect
```

```matlab
[posError,velError,dimError,yawError] = helperExtendedTargetError(track,truth,aProfiles);

% For multiObjectTracker, add a constant penalty for not estimating yaw
% and dimensions
if isnan(dimError)
    dimError = 1;
end
if isnan(yawError)
    yawError = 1;
end

% Distance is the sum of errors
dist = posError + velError + dimError + yawError;


end
```

helperInitGGIWFilter

Function to create a ggiwphd filter from a detection cell.

```matlab
function phd = helperInitGGIWFilter(varargin)
% helperInitGGIWFilter A function to initialize the GGIW-PHD filter for the
% Extended Object Tracking example

% Create a ggiwphd filter using 5 states and the constant turn-rate models.
phd = ggiwphd(zeros(5,1),eye(5),...
    'StateTransitionFcn',@constturn,...
    'StateTransitionJacobianFcn',@constturnjac,...
    'MeasurementFcn',@ctmeas,...
    'MeasurementJacobianFcn',@ctmeasjac,...
    'HasAdditiveMeasurementNoise',true,...
    'HasAdditiveProcessNoise',false,...
    'ProcessNoise',diag([1 1 3]),...
    'MaxNumComponents',1000,...
    'ExtentRotationFcn',@extentRotFcn,...
    'PositionIndex',[1 3]);

% If the function is called with no inputs i.e. the predictive portion of
% the birth density, no components are added to the mixture.
if nargin == 0
    % Nullify to return 0 components.
    nullify(phd);
else
    % When called with detections input, add two components to the filter,
    % one for car and one for truck, More components can be added based on
    % prior knowledge of the scenario, example, pedestrian or motorcycle.
    % This is a "multi-model" type approach. Another approach can be to add
    % only 1 component with a higher covariance in the dimensions. The
    % later is computationally less demanding, but has a tendency to track
    % observable dimensions of the object. For example, if only the back is
    % visible, the measurement noise may cause the length of the object to
    % shrink.

    % Detections
    detections = varargin{1};
```

```matlab
        % Enable elevation measurements to create a 3-D filter using
        % initctggiwphd
        if detections{1}.SensorIndex < 7
            for i = 1:numel(detections)
                detections{i}.Measurement = [detections{i}.Measurement(1);0;detections{i}.Measurement
                detections{i}.MeasurementNoise = blkdiag(detections{i}.MeasurementNoise(1,1),0.4,dete
                detections{i}.MeasurementParameters{1}(1).HasElevation = true;
            end
        end
        phd3d = initctggiwphd(detections);

        % Set states of the 2-D filter using 3-D filter
        phd.States = phd3d.States(1:5);
        phd.StateCovariances = phd3d.StateCovariances(1:5,1:5);

        phd.DegreesOfFreedom = 1000;
        phd.ScaleMatrices = (1000-4)*diag([4.7/2 1.8/2].^2);

        % Add truck dimensions as second component
        append(phd,phd);
        phd.ScaleMatrices(:,:,2) = (1000-4)*diag([8.1/2 2.45/2].^2);
        phd.GammaForgettingFactors = [1.03 1.03];

        % Relative weights of the components. Can be treated as probability of
        % existence of a car vs a truck on road.
        phd.Weights = [0.7 0.3];
    end
end

function R = extentRotFcn(x,dT)
    % Rotation of the extent during prediction.
    w = x(5);
    theta = w*dT;
    R = [cosd(theta) -sind(theta);sind(theta) cosd(theta)];
end
```

helperInitRectangularFilter

Function to create a gmphd rectangular target filter from a detection cell

```matlab
function filter = helperInitRectangularFilter(varargin)
% helperInitRectangularFilter A function to initialize the rectangular
% target PHD filter for the Extended Object Tracking example

% Copyright 2019 The MathWorks, Inc.

if nargin == 0
    % If called with no inputs, simply use the initctrectgmphd function to
    % create a PHD filter with no components.
    filter = initctrectgmphd;
    % Set process noise
    filter.ProcessNoise = diag([1 3]);
else
    % When called with detections input, add two components to the filter,
    % one for car and one for truck, More components can be added based on
```

```
% prior knowledge of the scenario, example, pedestrian or motorcycle.
% This is a "multi-model" type approach. Another approach can be to add
% only 1 component with a higher covariance in the dimensions. The
% later is computationally less demanding, but has a tendency to track
% observable dimensions of the object. For example, if only the back is
% visible, the measurement noise may cause the length of the object to
% shrink.

% Detections
detections = varargin{1};

% Create a GM-PHD filter with rectangular model
filter = initctrectgmphd(detections);

% Length width of a passenger car
filter.States(6:7,1) = [4.7;1.8];

% High certainty in dimensions
lCov = 1e-4;
wCov = 1e-4;
lwCorr = 0.5;
lwCov = sqrt(lCov*wCov)*lwCorr;
filter.StateCovariances(6:7,6:7,1) = [lCov lwCov;lwCov wCov];

% Add one more component by appending the filter with itself.
append(filter,filter);

% Set length and width to a truck dimensions
filter.States(6:7,2) = [8.1;2.45];

% Relative weights of each component
filter.Weights = [0.7 0.3];
end

end
```

Tracking Closely Spaced Targets Under Ambiguity

This example shows how to track objects when the association of sensor detections to tracks is ambiguous. In this example, you use a single-hypothesis tracker, a multiple-hypothesis tracker, and a probabilistic data association tracker to compare how the trackers handle this ambiguity. To track, the maneuvering targets better, you estimate the motion of the targets by using various models.

Introduction

Tracking is the process of estimating the situation based on data gathered by one or more sources of information. Tracking attempts to answer the following questions:

1 How many objects are there?

2 Where are the objects located in space?

3 What is their speed and direction of motion?

4 How do the objects maneuver relative to each other?

Trackers rely on sensor data such as radar detections. Sensors can generate two detections for two targets if the sensors can resolve the targets spatially. If two targets are closely spaced, they can fall within a single sensor resolution cell and the sensor reports only one detection of them. To illustrate the problem, the following figure shows a radar with a 1.5 degrees azimuth resolution. At a range of 20 km, the radar sensor bin has a width of above 540 m, which means that any detection reported from that range is located at the center of the bin, with a ±270 m uncertainty around it. Any two targets within this bin are reported as a single detection.
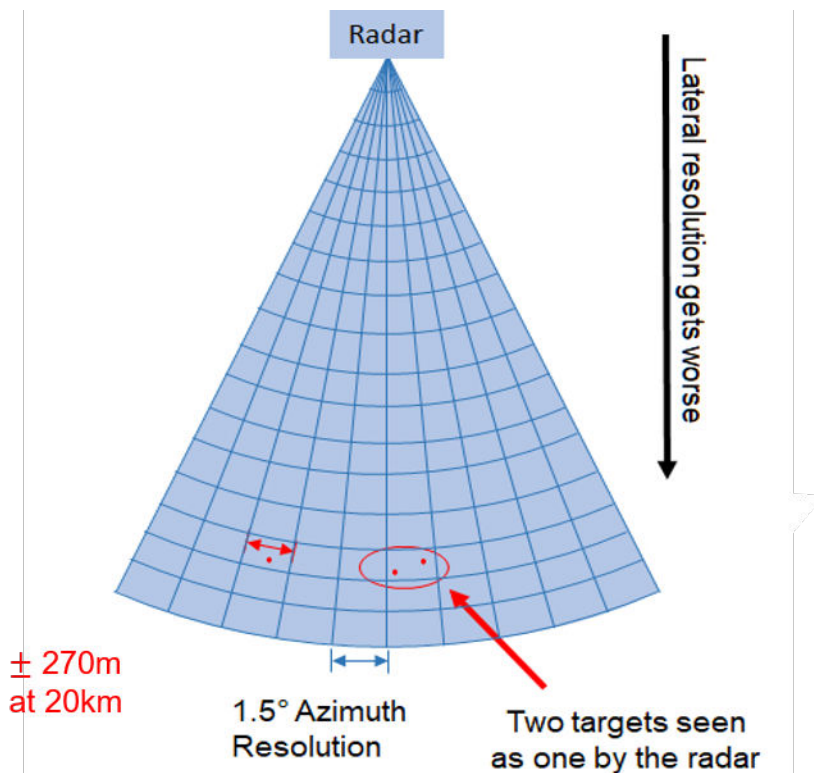
Scenario

To demonstrate a case where sensor reports are ambiguously assigned to tracks, you create a simple scenario. In this scenario, a single `monostaticRadarSensor` object, located at the origin (not shown), scans a small region about 20 km from the radar. Initially, the radar reports about two detections per scan. When the detections are coming from a region around the X = 0, Y = -20 km position, the radar reports a single detection per scan for a while, followed by two radar detections reported from around Y = -19.5km and toward the sensor (up).

The scenario and detections log is already saved in a matfile. You can uncomment the lines below to regenerate the scenario and the synthetic detections.

```
load ATCdata.mat
% scenario = helperCreateScenario;
% dataLog = helperRunDetections(scenario);
plotScenarioAndDetections(dataLog);
```
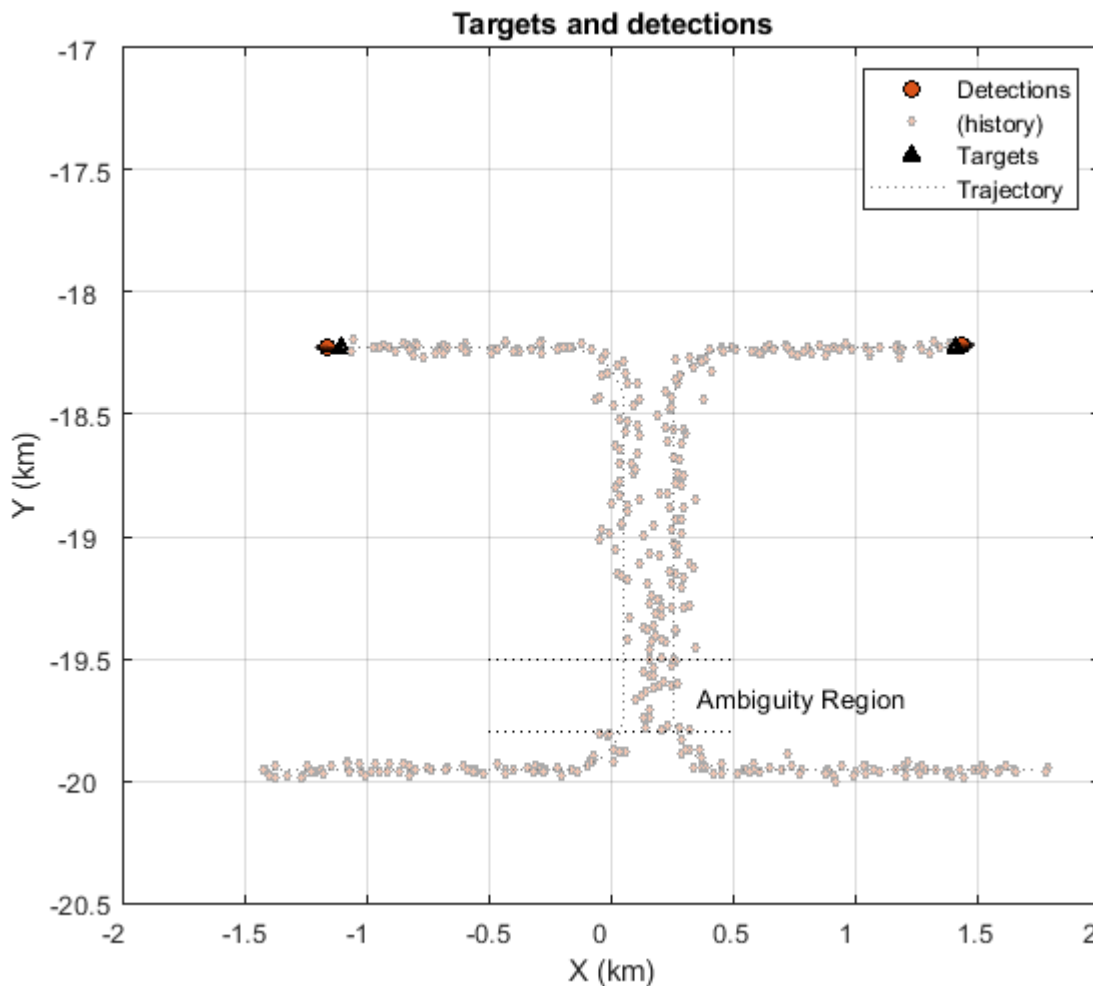

Looking only at the detections in this plot, it is reasonable to assume that there are two targets, but it is unclear whether their trajectories crossed. The targets did not cross as seen by their trajectory.

Single-Hypothesis Tracker with Constant Velocity Model

The simplest choice of a multi-target tracker is a single-hypothesis tracker like the `trackerGNN`. You set it to use a score logic, to allow easier comparison with `trackerTOMHT` later in this example.

This example is small and does not require more than 20 tracks. The gate should be chosen to allow tracks to be established without spurious tracks and was increased from the default.

The sensor bin volume can be roughly estimated using the determinant of a detection measurement noise. In this case, the value is about 1e9, so you set the volume to 1e9. The value of beta should specify how many new objects are expected in a unit volume. As the number of objects in this scenario is constant, set beta to be very small. The values for probability of detection and false alarm rate are taken from the corresponding values of the radar.

```matlab
numTracks = 20; % Maximum number of tracks
gate = 45;      % Association gate
vol = 1e9;      % Sensor bin volume
beta = 1e-14;   % Rate of new targets in a unit volume
pd = 0.8;       % Probability of detection
far = 1e-6;     % False alarm rate
```

You use a constant velocity extended Kalman filter. The `initCVFilter` function modifies the filter that `initcvekf` returns to allow for a higher uncertainty in the velocity terms and a higher horizontal acceleration in the process noise. These modifications are needed to track the targets in this scenario, which move at a high speed and maneuver by turning in the horizontal plane.

```matlab
tracker = trackerGNN( ...
    'FilterInitializationFcn',@initCVFilter,...
    'MaxNumTracks', numTracks, ...
    'MaxNumSensors', 1, ...
    'AssignmentThreshold',gate, ...
    'TrackLogic', 'Score', ...
    'DetectionProbability', pd, 'FalseAlarmRate', far, ...
    'Volume', vol, 'Beta', beta);
```

The following line runs the scenario and produces the visualization.

```matlab
[trackSummary, truthSummary, trackMetrics, truthMetrics,timeGNNCV] = helperRunTracker(dataLog,tra
```

Tracker: GNN. Model: CV
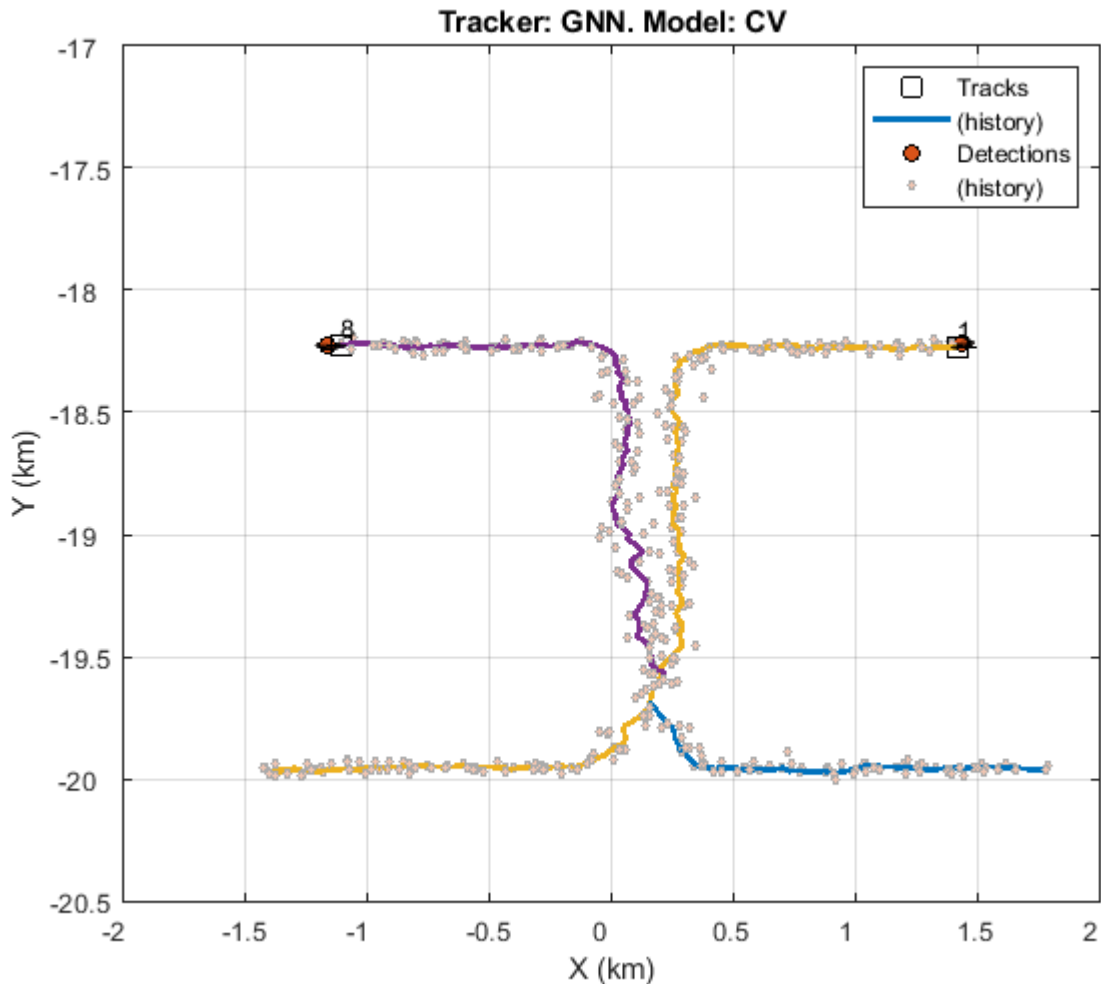
With the benefit of using a simulated ground truth, we can compare the results of the tracker with the truth, using `trackAssignmentMetrics`. The results show that there are two truth objects, but three tracks were generated by the tracker. One of the tracks did not survive until the end of the scenario, that is, it was dropped, and two other tracks did.

At the end of the scenario, truth object 2 was associated with track 8, which was created midway through the scenario after track 2 was dropped. Truth object 3 was assigned track 1 at the end of the scenario, but has two breaks.

Note that the establishment length of both truth objects was small while there were few false tracks, which means that the confirmation threshold is good.

```
disp(trackSummary)
disp(truthSummary)
```

TrackID	AssignedTruthID	Surviving	TotalLength	DivergenceStatus
1	3	true	190	false

2	NaN	false	77	true
8	2	true	111	false

TruthID	AssociatedTrackID	TotalLength	BreakCount	EstablishmentLength
2	8	192	1	4
3	1	192	2	2

Use the `trackErrorMetrics` to analyze the quality of tracking in terms of position and velocity errors between the truth objects and the tracks associated with them. The results show that the position errors are about 50-60 m and the velocity errors are about 30 m/s RMS.

```
disp(trackMetrics)
disp(truthMetrics)
```

TrackID	posRMS	velRMS	posANEES	velANEES
1	59.461	26.119	7.9331	3.495
2	51.803	39.433	5.7494	3.0362
8	62.914	28.877	7.2004	3.0465

TruthID	posRMS	velRMS	posANEES	velANEES
2	62.147	27.546	8.3597	3.9608
3	56.218	32.109	6.3234	2.6642

Single-Hypothesis Tracker with Interacting Multiple Models

The combination of a single-hypothesis tracker and a constant velocity filter could not track the maneuvering targets. This result is evident by the fact that one of the tracks stops and then a new track begins, while there are only two targets in the scenario. It is also evident from the inability to follow the target turns.

One option to improve the tracker is to modify the filter to be an interacting multiple-model (IMM) filter, which allows you to consider two or more motion models for the targets. The filter switches to the correct model based on the likelihood of one model over the other given the measurements. In this example, the targets move in a constant velocity until they turn at a constant rate, so you define an IMM filter with these two models using the `initIMMFilter` function.

```
tracker = trackerGNN( ...
    'FilterInitializationFcn',@initIMMFilter,...
    'MaxNumTracks', numTracks, ...
    'MaxNumSensors', 1, ...
    'AssignmentThreshold',gate, ...
    'TrackLogic', 'Score', ...
    'DetectionProbability', pd, 'FalseAlarmRate', far, ...
    'Volume', vol, 'Beta', beta);
[trackSummary, truthSummary, trackMetrics, truthMetrics, timeGNNIMM] = helperRunTracker(dataLog,
```
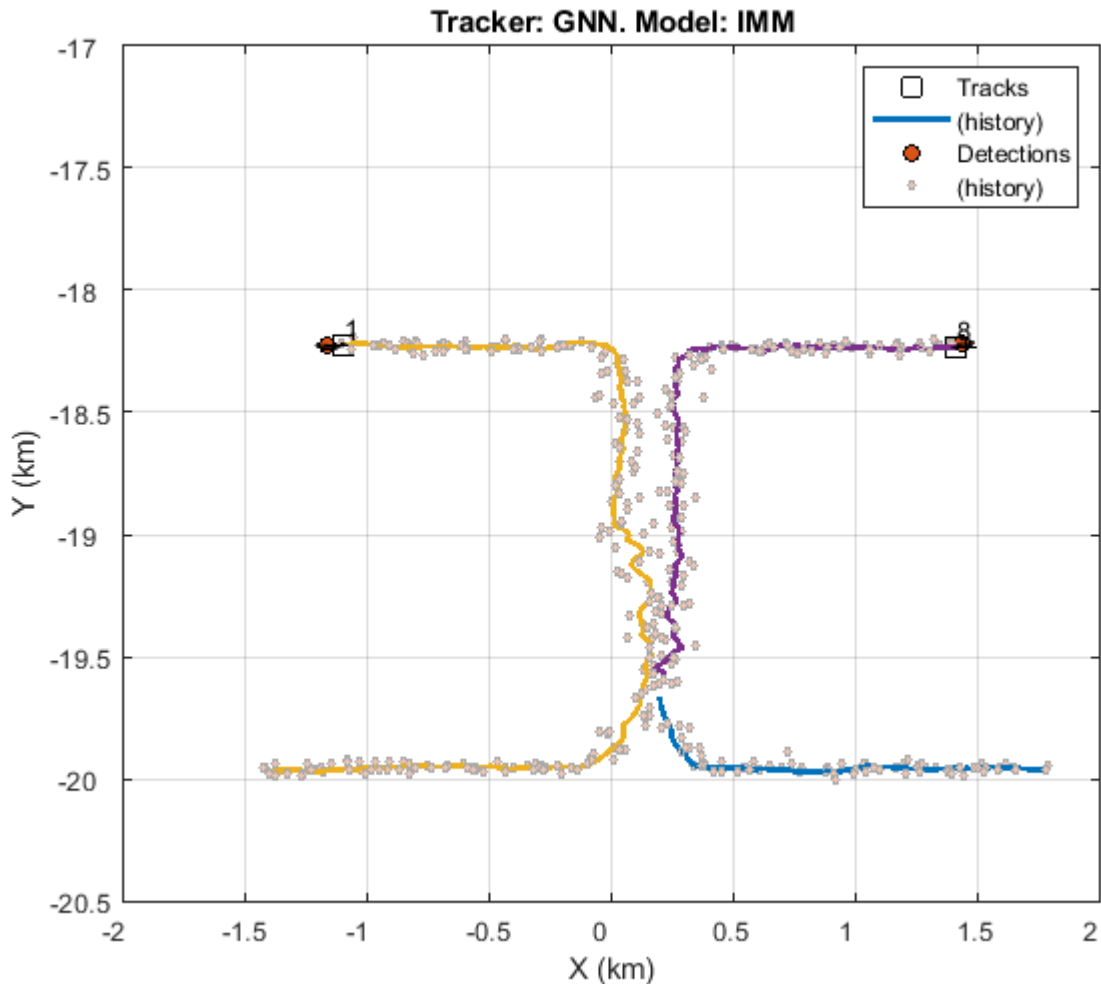
The addition of an IMM filter enables the tracker to identify the target maneuver correctly. You can observe this conclusion by looking at the tracks generated during the target turns and notice how well both turns are tracked. As a result, truth object 2 has zero breaks, which you can see in the plot by the continuous history of its associated track.

However, even with the interacting models, one of the tracks breaks in the ambiguity region. The single-hypothesis tracker gets only one detection with that region, and can update only one of the tracks with it, coasting the other track. After a few updates, the score of the coasted track falls below the deletion threshold and the tracker drops the track.

```
disp(trackSummary)
disp(truthSummary)
```

TrackID	AssignedTruthID	Surviving	TotalLength	DivergenceStatus
1	2	true	190	false
2	NaN	false	77	true
8	3	true	111	false

TruthID	AssociatedTrackID	TotalLength	BreakCount	EstablishmentLength
2	1	192	0	4
3	8	192	1	2

The use of an IMM also improved the tracking accuracy, as the position errors reduced to 40-60 meters, while the velocity errors reduced to 25-30 m/s. This improvement can be attributed to the lower process noise values used in the IMM for each model, which allow better smoothing of the noisy measurements.

```
disp(trackMetrics)
disp(truthMetrics)
```

TrackID	posRMS	velRMS	posANEES	velANEES
1	52.445	22.4	7.6006	6.0031
2	48.149	35.239	6.5401	3.4887
8	34.787	20.279	5.2549	4.2032

TruthID	posRMS	velRMS	posANEES	velANEES
2	52.445	22.4	7.6006	6.0031
3	40.75	27.364	5.7772	3.9128

Multiple-Hypothesis Tracker with Constant Velocity Model

To resolve the problem of the broken track, use a multiple-hypothesis tracker (MHT), `trackerTOMHT`. In the ambiguity region, the tracker generates multiple hypotheses about the association of the detection with the tracks. In particular, it maintains one hypothesis that the first track is assigned this detection and another hypothesis that the second track is assigned the detection. By doing so, both tracks are kept alive, so that the tracker gets enough detections to resolve the ambiguity in the next updates.

`trackerTOMHT` uses the same parameters as `trackerGNN`, except:

- `AssignmentThreshold` allows a track to be considered unassigned even when there is an assigned detection. This setting allows multiple branches per track. However, the second element of the gate disallows new tracks from assigned detections, to reduce the total number of tracks and improve performance.
- `MaxNumHistoryScans` is increased to 10, to delay the `NScanPruning`.
- `MaxNumTrackBranches` is increased to 5, to allow more branches (hypotheses) per track.

```
tracker = trackerTOMHT( ...
    'FilterInitializationFcn',@initCVFilter, ...
    'MaxNumTracks', numTracks, ...
    'MaxNumSensors', 1, ...
    'AssignmentThreshold', [0.2, 1, 1]*gate,...
    'DetectionProbability', pd, 'FalseAlarmRate', far, ...
    'Volume', vol, 'Beta', beta, ...
    'MaxNumHistoryScans', 10,...
```

```
            'MaxNumTrackBranches', 5,...
            'NScanPruning', 'Hypothesis', ...
            'OutputRepresentation', 'Tracks');

    [trackSummary, truthSummary, trackMetrics, truthMetrics, timeTOMHTCV] = helperRunTracker(dataLog
```

Tracker: TOMHT. Model: CV
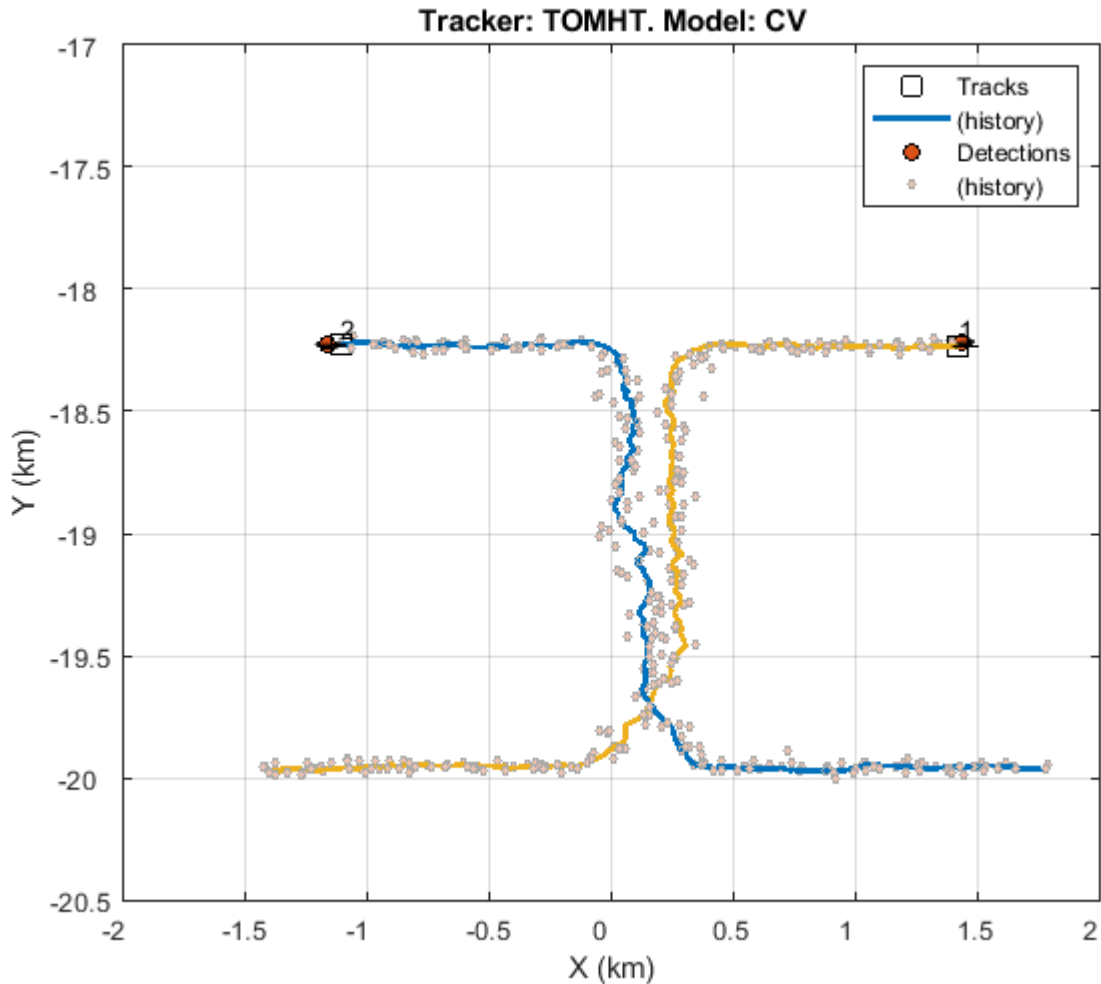
The results show the multiple-hypothesis tracker is capable of tracking the two truth objects throughout the scenario. For the ambiguity region, the MHT tracker formulates two hypotheses about the assignment:

1   The detection is assigned to track 1.

2   The detection is assigned to track 2.

With these hypotheses, both tracks generate branches (track hypotheses) that update them using the same detection. Obviously, using the same detection to update both tracks causes the tracks to become closer in their estimate, and eventually the two tracks may coalesce. However, if the duration of the ambiguous assignment is short, the tracker may be able to resolve the two tracks when there are two detections. In this case, you see that the two tracks cross each other, but the metrics show

that the break count for each truth is 1, meaning that the true targets probably did not cross each other.

```
disp(trackSummary)
disp(truthSummary)
```

TrackID	AssignedTruthID	Surviving	TotalLength	DivergenceStatus
1	3	true	190	false
2	2	true	191	false

TruthID	AssociatedTrackID	TotalLength	BreakCount	EstablishmentLength
2	2	192	1	2
3	1	192	1	2

In terms of tracking accuracy, the position and velocity errors of this tracker are similar to the ones from the combination of a single-hypothesis tracker with a constant velocity filter.

```
disp(trackMetrics)
disp(truthMetrics)
```

TrackID	posRMS	velRMS	posANEES	velANEES
1	45.532	28.802	4.5909	3.2827
2	61.968	29.851	6.7191	2.7338

TruthID	posRMS	velRMS	posANEES	velANEES
2	48.877	28.57	5.0144	3.2761
3	58.203	29.895	6.1414	2.8049

Multiple-Hypothesis Tracker with Interacting Multiple Models

Building on the success of using an multiple-hypothesis tracker with constant velocity to maintain continuous tracking, combine the tracker with the benefits of an IMM filter. The IMM filter may be more successful in preventing the track crossing as it improves the tracking when the targets turn. The following code configures `trackerTOMHT` with a `trackingIMM` filter.

```
tracker = trackerTOMHT( ...
    'FilterInitializationFcn',@initIMMFilter, ...
    'MaxNumTracks', numTracks, ...
    'MaxNumSensors', 1, ...
    'AssignmentThreshold', [0.2, 1, 1]*gate,...
    'DetectionProbability', pd, 'FalseAlarmRate', far, ...
    'Volume', vol, 'Beta', beta, ...
    'MaxNumHistoryScans', 10,...
    'MaxNumTrackBranches', 5,...
    'NScanPruning', 'Hypothesis', ...
    'OutputRepresentation', 'Tracks');
[trackSummary, truthSummary, trackMetrics, truthMetrics, timeTOMHTIMM] = helperRunTracker(dataLog
```

Tracker: TOMHT. Model: IMM
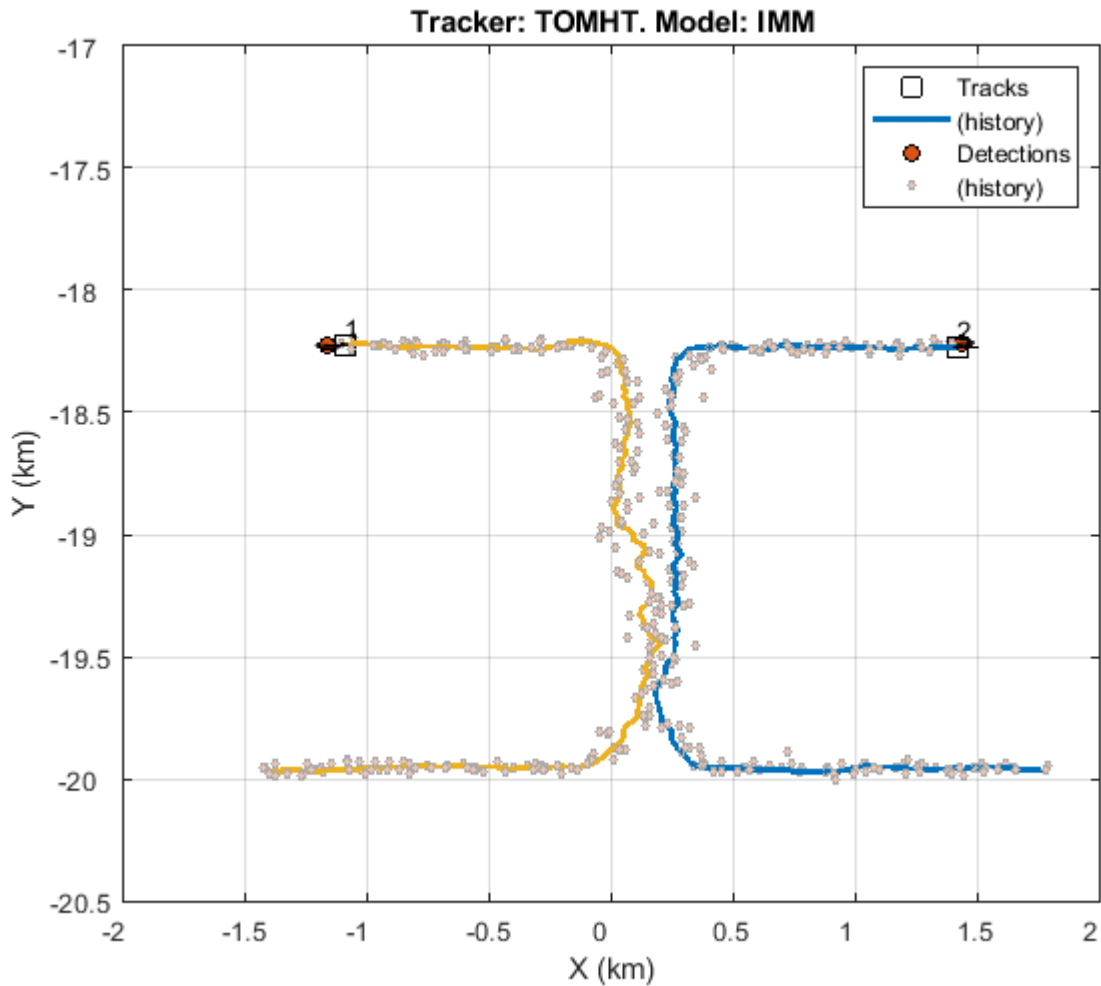
The plot shows that the two tracks did not cross. This result is also evident in the break count of the true targets below, which shows zero breaks.

You can also see the true path of the targets, shown in solid line.

```
disp(trackSummary)
disp(truthSummary)
```

TrackID	AssignedTruthID	Surviving	TotalLength	DivergenceStatus
1	2	true	190	false
2	3	true	191	false

TruthID	AssociatedTrackID	TotalLength	BreakCount	EstablishmentLength
2	1	192	0	2
3	2	192	0	2

The tracking accuracy is similar to the combination of single-hypothesis tracker with IMM filter. Note that the truth accuracy and the associated track accuracy are the same because there was no break in the tracking throughout the scenario.

```
disp(trackMetrics)
disp(truthMetrics)
```

TrackID	posRMS	velRMS	posANEES	velANEES
1	55.947	24.442	6.9595	5.5471
2	37.797	25.949	5.2605	4.0935

TruthID	posRMS	velRMS	posANEES	velANEES
2	55.947	24.442	6.9595	5.5471
3	37.797	25.949	5.2605	4.0935

Joint Probabilistic Data Association Tracker with Constant Velocity Model

Although the tracking metrics are greatly improved by using `trackerTOMHT` over `trackerGNN`, the computer processing time is also significantly increased. A Joint Probabilistic Data Association (JPDA) tracker, `trackerJPDA`, will allow you to explore further trade-off considerations. Unlike GNN, JPDA allows a single detection to be used for updating multiple tracks in its vicinity. Moreover, multiple detections can be clustered together with several tracks and update each of those tracks. This resolves the issue of the broken track. However, JPDA does not maintain multiple hypothesis over multiple scans, which makes it a sub-optimal approach as opposed to MHT.

`trackerJPDA` uses the same parameters as `trackerGNN` except:

- `ClutterDensity` is the ratio of `FalseAlarmRate` and `Volume`.
- `NewTargetDensity` replaces `Beta`.
- `TimeTolerance` allows for processing multiple detections from the scanning radar in a single cluster.

Additionally, set the `TrackLogic` to `Integrated` which is conceptually closer to the score logic used with the previous two trackers.

```
tracker = trackerJPDA(...
    'FilterInitializationFcn',@initCVFilter,...
    'MaxNumTracks', numTracks, ...
    'MaxNumSensors', 1, ...
    'AssignmentThreshold',gate, ...
    'TrackLogic','Integrated',...
    'DetectionProbability', pd, ...
    'ClutterDensity', far/vol, ...
    'NewTargetDensity', beta,...
    'TimeTolerance',0.05);

[trackSummary, truthSummary, trackMetrics, truthMetrics, timeJPDACV] = helperRunTracker(dataLog,
```

Tracker: JPDA. Model: CV
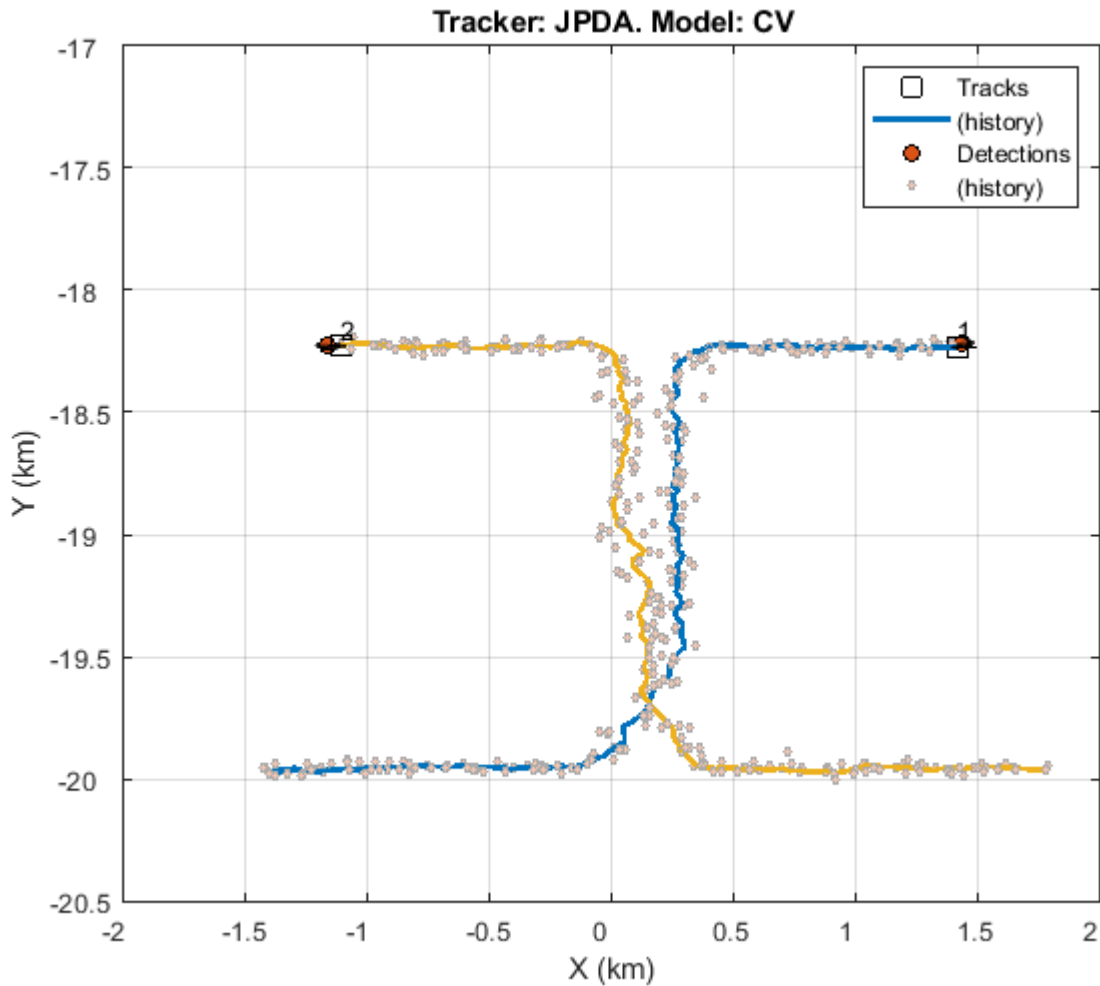
Although, trackerJPDA does not maintain multiple hypothesis, it allows both tracks to remain confirmed in the ambiguity region where only one detection is reported per update. Both tracks can be assigned to the detection with different probabilities. However, the tracks cross each other as observed before with the other trackers. The metrics break count of each truth is also 1.

```
disp(trackSummary)
disp(truthSummary)
```

TrackID	AssignedTruthID	Surviving	TotalLength	DivergenceStatus
1	3	true	191	false
2	2	true	191	false

TruthID	AssociatedTrackID	TotalLength	BreakCount	EstablishmentLength
2	2	192	1	1
3	1	192	1	2

The tracking accuracy is on par with what is obtained with `trackerGNN` and `trackerTOMHT`.

```
disp(trackMetrics)
disp(truthMetrics)
```

TrackID	posRMS	velRMS	posANEES	velANEES
1	44.918	29.911	5.1894	3.5599
2	58.904	31.484	7.0023	3.0944

TruthID	posRMS	velRMS	posANEES	velANEES
2	47.304	29.606	5.4677	3.5093
3	56.106	31.564	6.5976	3.1815

Joint Probabilistic Data Association Tracker with Interacting Multiple Models

As seen with the other two trackers, the turn before the ambiguity region is better addressed with interacting multiple models. `trackerJPDA` can also be used with `trackingIMM`, using the same `FilterInitializationFcn` as before.

```
tracker = trackerJPDA( ...
    'FilterInitializationFcn',@initIMMFilter,...
    'MaxNumTracks', numTracks, ...
    'MaxNumSensors', 1, ...
    'AssignmentThreshold',gate, ...
    'TrackLogic','Integrated',...
    'DetectionProbability', pd, ...
    'ClutterDensity', far/vol, ...
    'NewTargetDensity', beta,...
    'TimeTolerance',0.05);

[trackSummary, truthSummary, trackMetrics, truthMetrics,timeJPDAIMM] = helperRunTracker(dataLog,t
```
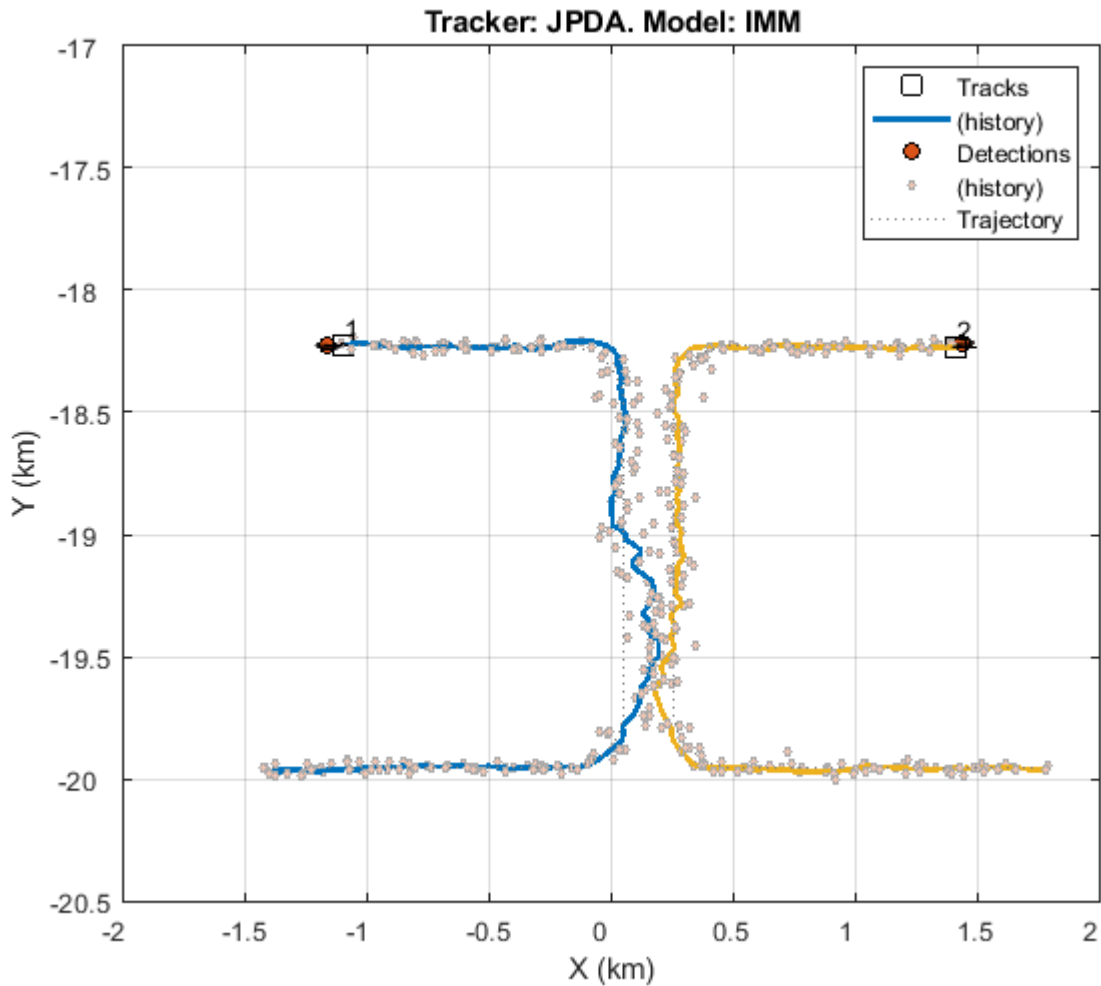
The results are comparable to `trackerTOMHT`, thanks to the Interacting Multiple Models, the targets are more precisely tracked during the turn and are sufficiently separated before entering the ambiguity region. Both tracks 1 and 2 are assigned to a target with zero break counts.

```
disp(trackSummary)
disp(truthSummary)
```

TrackID	AssignedTruthID	Surviving	TotalLength	DivergenceStatus
1	2	true	191	false
2	3	true	191	false

TruthID	AssociatedTrackID	TotalLength	BreakCount	EstablishmentLength
2	1	192	0	1
3	2	192	0	2

Again, tracking accuracy results are similar to what is obtained previously using `trackerGNN` and `trackerTOMHT`.

```
disp(trackMetrics)
disp(truthMetrics)
```

TrackID	posRMS	velRMS	posANEES	velANEES
1	57.172	27.523	7.6831	6.2685
2	39.21	29.184	6.1317	4.769

TruthID	posRMS	velRMS	posANEES	velANEES
2	57.172	27.523	7.6831	6.2685
3	39.21	29.184	6.1317	4.769

Time Performance Comparison

Another point of comparison for each tracker and filter combination is the runtime. The plot below shows records of the tracking loop duration. The results are normalized based on the runtime value of the GNN tracker with CV model.

```
figure
c1 = categorical({'CV','IMM'});
timeData = [timeGNNCV timeJPDACV timeTOMHTCV ; timeGNNIMM timeJPDAIMM timeTOMHTIMM]/timeGNNCV;
bar(c1,timeData)
legend('GNN','JPDA','TOMHT','Location','northwest')
ylabel('Normalized Time (x timeGNN-CV)')
title('Time Performance Comparison')
ax = gca;
ax.YGrid = 'on';
```
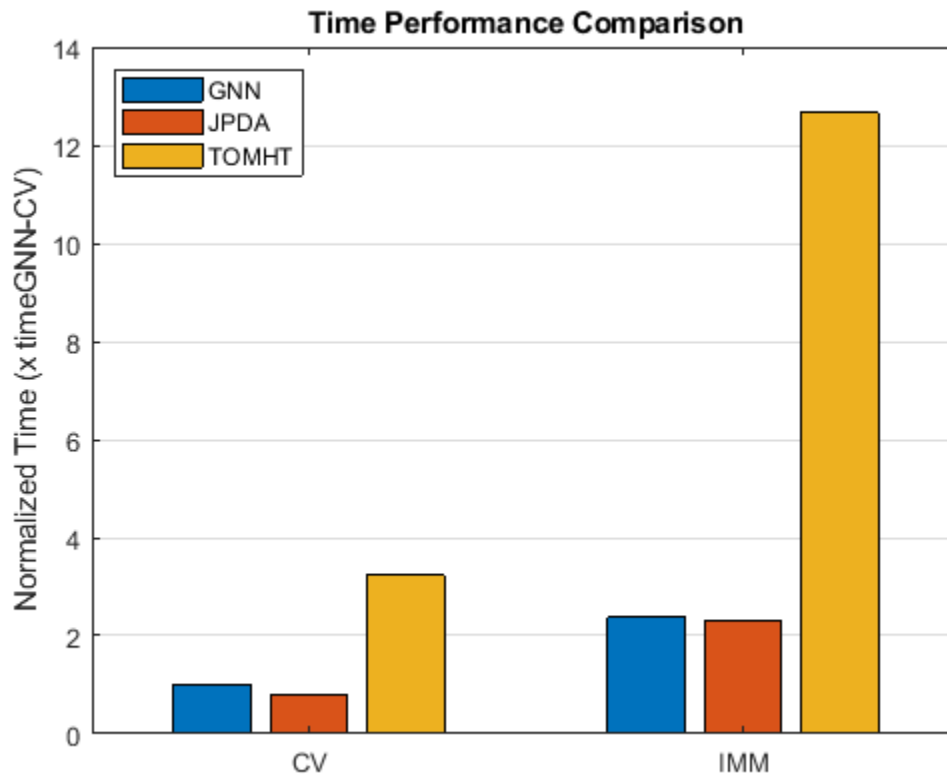
The results show that GNN and JPDA can track the targets 5 to 6 times faster than MHT depending on the motion model. The IMM motion model makes all three trackers run 3 to 4 times slower. Note that each tracker processing time varies differently depending on the scenario's number of target, density of false alarms, density of targets, etc. This example does not guarantee similar performance comparison in different use cases.

Summary

In this example, you created a scenario in which two maneuvering targets are detected by a single sensor, where some of their motion is within an area of ambiguity. You used six combinations of trackers and filters to show the contribution of each to the overall tracking. You observed that the constant velocity filter was less accurate in tracking the targets during their maneuver, which required an interacting multiple-model filter. You also observed the ability of MHT and JPDA to handle the case of ambiguous association of detections to tracks, and how it can be used to maintain continuous tracking while a single-hypothesis tracker cannot do that. Finally, you noticed the trade-offs between tracking results and processing time when choosing a tracker. In this case JPDA proves to be the best option. In different scenarios, you may require the more complex MHT when neither GNN nor JPDA gives acceptable tracking results. You may as well prefer GNN if there are less ambiguity regions or low clutter density.

Visual-Inertial Odometry Using Synthetic Data

This example shows how to estimate the pose (position and orientation) of a ground vehicle using an inertial measurement unit (IMU) and a monocular camera. In this example, you:

1   Create a driving scenario containing the ground truth trajectory of the vehicle.

2   Use an IMU and visual odometry model to generate measurements.

3   Fuse these measurements to estimate the pose of the vehicle and then display the results.

Visual-inertial odometry estimates pose by fusing the visual odometry pose estimate from the monocular camera and the pose estimate from the IMU. The IMU returns an accurate pose estimate for small time intervals, but suffers from large drift due to integrating the inertial sensor measurements. The monocular camera returns an accurate pose estimate over a larger time interval, but suffers from a scale ambiguity. Given these complementary strengths and weaknesses, the fusion of these sensors using visual-inertial odometry is a suitable choice. This method can be used in scenarios where GPS readings are unavailable, such as in an urban canyon.

Create a Driving Scenario with Trajectory

Create a `drivingScenario` (Automated Driving Toolbox) object that contains:

*   The road the vehicle travels on
*   The buildings surrounding either side of the road
*   The ground truth pose of the vehicle
*   The estimated pose of the vehicle

The ground truth pose of the vehicle is shown as a solid blue cuboid. The estimated pose is shown as a transparent blue cuboid. Note that the estimated pose does not appear in the initial visualization because the ground truth and estimated poses overlap.

Generate the baseline trajectory for the ground vehicle using the `waypointTrajectory` System object™. Note that the `waypointTrajectory` is used in place of `drivingScenario/trajectory` since the acceleration of the vehicle is needed. The trajectory is generated at a specified sampling rate using a set of waypoints, times of arrival, and velocities.

```
% Create the driving scenario with both the ground truth and estimated
% vehicle poses.
scene = drivingScenario;
groundTruthVehicle = vehicle(scene, 'PlotColor', [0 0.4470 0.7410]);
estVehicle = vehicle(scene, 'PlotColor', [0 0.4470 0.7410]);

% Generate the baseline trajectory.
sampleRate = 100;
wayPoints = [  0    0 0;
             200    0 0;
             200   50 0;
             200  230 0;
             215  245 0;
             260  245 0;
             290  240 0;
             310  258 0;
             290  275 0;
             260  260 0;
             -20  260 0];
```

```
t = [0 20 25 44 46 50 54 56 59 63 90].';
speed = 10;
velocities = [ speed     0 0;
               speed     0 0;
                   0 speed 0;
                   0 speed 0;
               speed     0 0;
               speed     0 0;
               speed     0 0;
                   0 speed 0;
              -speed     0 0;
              -speed     0 0;
              -speed     0 0];

traj = waypointTrajectory(wayPoints, 'TimeOfArrival', t, ...
    'Velocities', velocities, 'SampleRate', sampleRate);

% Add a road and buildings to scene and visualize.
helperPopulateScene(scene, groundTruthVehicle);
```
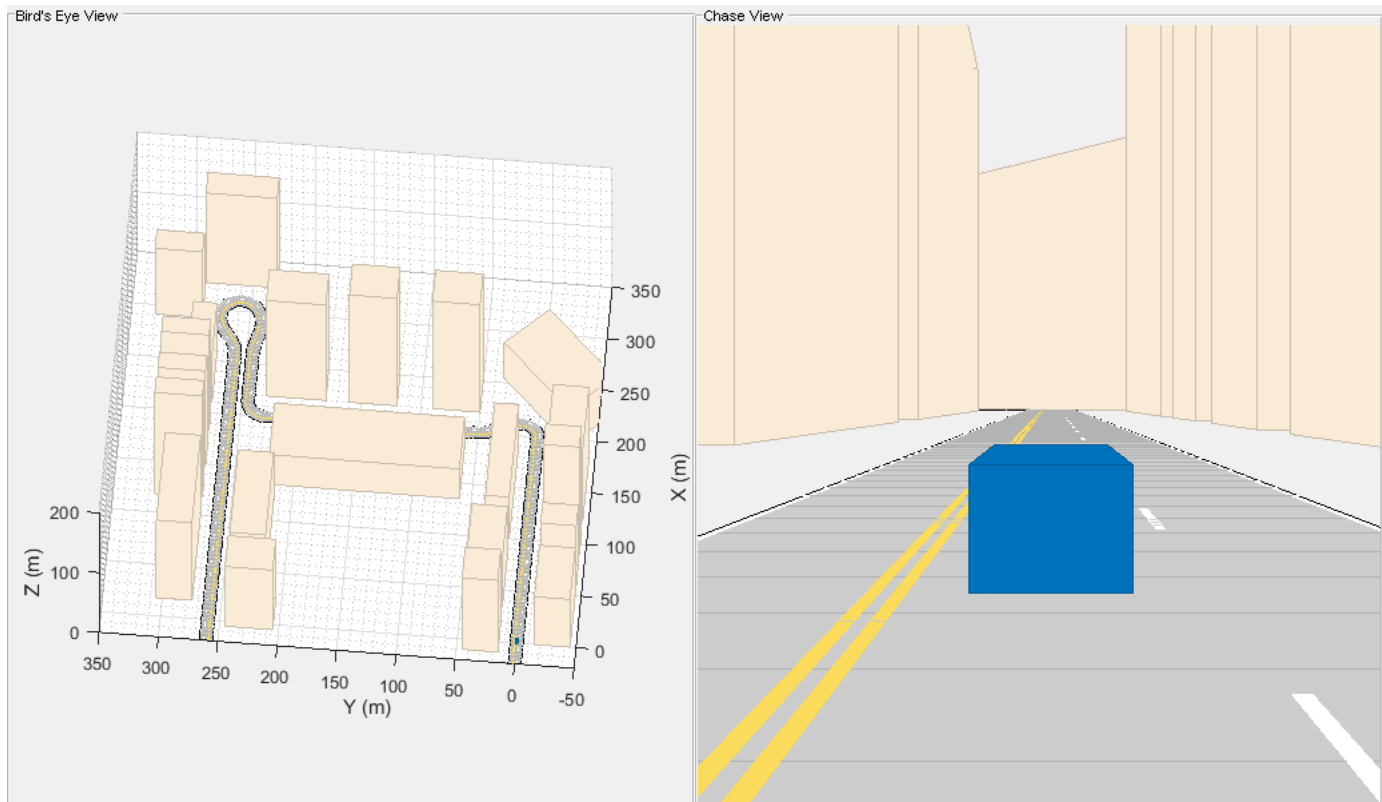

Create a Fusion Filter

Create the filter to fuse IMU and visual odometry measurements. This example uses a loosely coupled method to fuse the measurements. While the results are not as accurate as a tightly coupled method, the amount of processing required is significantly less and the results are adequate. The fusion filter uses an error-state Kalman filter to track orientation (as a quaternion), position, velocity, and sensor biases.

The `insfilterErrorState` object has the following functions to process sensor data: `predict` and `fusemvo`.

The `predict` function takes the accelerometer and gyroscope measurements from the IMU as inputs. Call the `predict` function each time the accelerometer and gyroscope are sampled. This function predicts the state forward by one time step based on the accelerometer and gyroscope measurements, and updates the error state covariance of the filter.

The `fusemvo` function takes the visual odometry pose estimates as input. This function updates the error states based on the visual odometry pose estimates by computing a Kalman gain that weighs the various inputs according to their uncertainty. As with the `predict` function, this function also updates the error state covariance, this time taking the Kalman gain into account. The state is then updated using the new error state and the error state is reset.

```
filt = insfilterErrorState('IMUSampleRate', sampleRate, ...
    'ReferenceFrame', 'ENU')
% Set the initial state and error state covariance.
helperInitialize(filt, traj);


filt =

  insfilterErrorState with properties:

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

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

Specify the Visual Odometry Model

Define the visual odometry model parameters. These parameters model a feature matching and tracking-based visual odometry system using a monocular camera. The `scale` parameter accounts for the unknown scale of subsequent vision frames of the monocular camera. The other parameters model the drift in the visual odometry reading as a combination of white noise and a first-order Gauss-Markov process.

```
% The flag useVO determines if visual odometry is used:
% useVO = false; % Only IMU is used.
useVO = true; % Both IMU and visual odometry are used.

paramsVO.scale = 2;
paramsVO.sigmaN = 0.139;
paramsVO.tau = 232;
paramsVO.sigmaB = sqrt(1.34);
paramsVO.driftBias = [0 0 0];
```

Specify the IMU Sensor

Define an IMU sensor model containing an accelerometer and gyroscope using the `imuSensor` System object. The sensor model contains properties to model both deterministic and stochastic noise sources. The property values set here are typical for low-cost MEMS sensors.

```
% Set the RNG seed to default to obtain the same results for subsequent
% runs.
rng('default')

imu = imuSensor('SampleRate', sampleRate, 'ReferenceFrame', 'ENU');

% Accelerometer
imu.Accelerometer.MeasurementRange =  19.6; % m/s^2
imu.Accelerometer.Resolution = 0.0024; % m/s^2/LSB
imu.Accelerometer.NoiseDensity = 0.01; % (m/s^2)/sqrt(Hz)

% Gyroscope
imu.Gyroscope.MeasurementRange = deg2rad(250); % rad/s
imu.Gyroscope.Resolution = deg2rad(0.0625); % rad/s/LSB
imu.Gyroscope.NoiseDensity = deg2rad(0.0573); % (rad/s)/sqrt(Hz)
imu.Gyroscope.ConstantBias = deg2rad(2); % rad/s
```

Set Up the Simulation

Specify the amount of time to run the simulation and initialize variables that are logged during the simulation loop.

```
% Run the simulation for 60 seconds.
numSecondsToSimulate = 60;
numIMUSamples = numSecondsToSimulate * sampleRate;

% Define the visual odometry sampling rate.
imuSamplesPerCamera = 4;
numCameraSamples = ceil(numIMUSamples / imuSamplesPerCamera);

% Preallocate data arrays for plotting results.
[pos, orient, vel, acc, angvel, ...
    posVO, orientVO, ...
    posEst, orientEst, velEst] ...
    = helperPreallocateData(numIMUSamples, numCameraSamples);

% Set measurement noise parameters for the visual odometry fusion.
RposVO = 0.1;
RorientVO = 0.1;
```

Run the Simulation Loop

Run the simulation at the IMU sampling rate. Each IMU sample is used to predict the filter's state forward by one time step. Once a new visual odometry reading is available, it is used to correct the current filter state.

There is some drift in the filter estimates that can be further corrected with an additional sensor such as a GPS or an additional constraint such as a road boundary map.

```
cameraIdx = 1;
for i = 1:numIMUSamples
    % Generate ground truth trajectory values.
```

```matlab
    [pos(i,:), orient(i,:), vel(i,:), acc(i,:), angvel(i,:)] = traj();

    % Generate accelerometer and gyroscope measurements from the ground truth
    % trajectory values.
    [accelMeas, gyroMeas] = imu(acc(i,:), angvel(i,:), orient(i));

    % Predict the filter state forward one time step based on the
    % accelerometer and gyroscope measurements.
    predict(filt, accelMeas, gyroMeas);

    if (1 == mod(i, imuSamplesPerCamera)) && useVO
        % Generate a visual odometry pose estimate from the ground truth
        % values and the visual odometry model.
        [posVO(cameraIdx,:), orientVO(cameraIdx,:), paramsVO] = ...
            helperVisualOdometryModel(pos(i,:), orient(i,:), paramsVO);

        % Correct filter state based on visual odometry data.
        fusemvo(filt, posVO(cameraIdx,:), RposVO, ...
            orientVO(cameraIdx), RorientVO);

        cameraIdx = cameraIdx + 1;
    end

    [posEst(i,:), orientEst(i,:), velEst(i,:)] = pose(filt);

    % Update estimated vehicle pose.
    helperUpdatePose(estVehicle, posEst(i,:), velEst(i,:), orientEst(i));

    % Update ground truth vehicle pose.
    helperUpdatePose(groundTruthVehicle, pos(i,:), vel(i,:), orient(i));

    % Update driving scenario visualization.
    updatePlots(scene);
    drawnow limitrate;
end
```

5-187
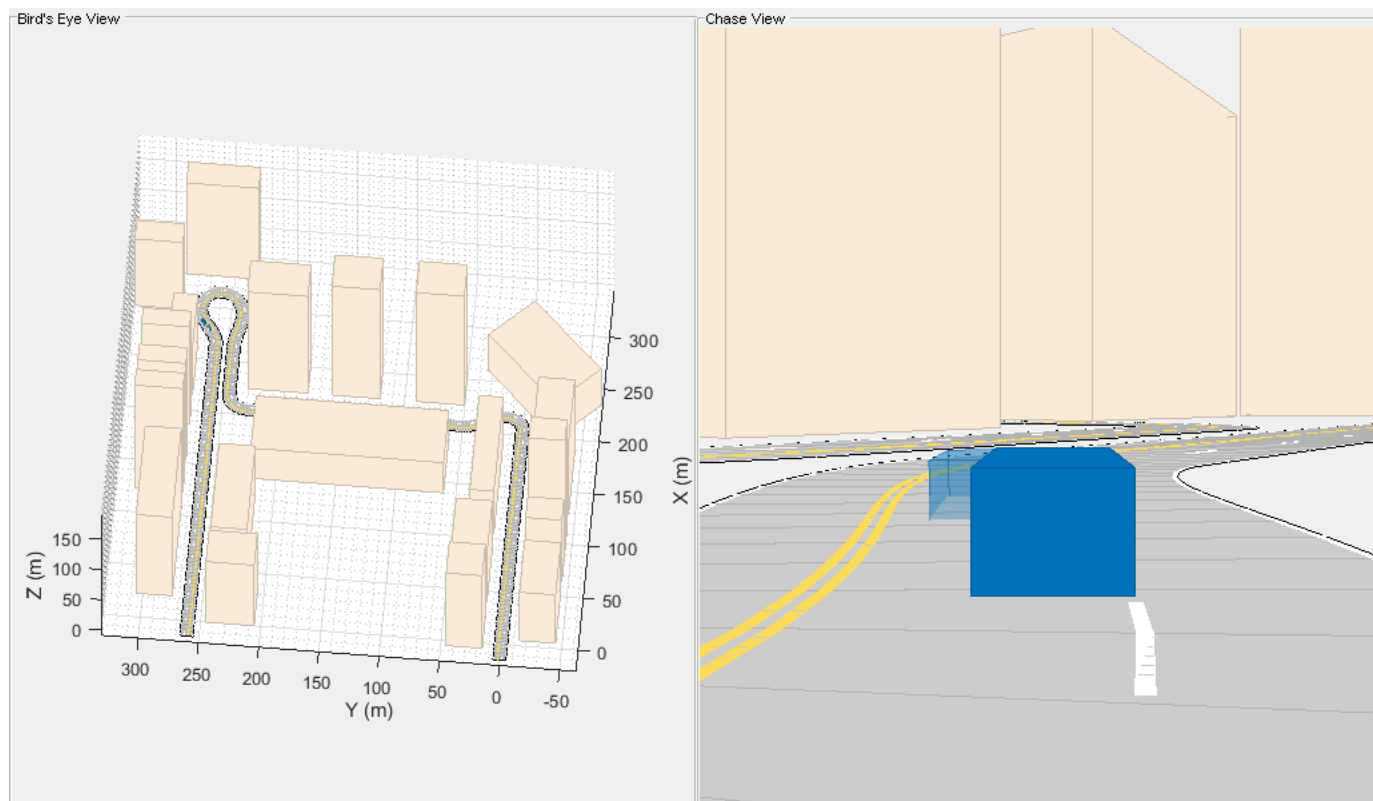
Plot the Results

Plot the ground truth vehicle trajectory, the visual odometry estimate, and the fusion filter estimate.

```
figure
if useVO
    plot3(pos(:,1), pos(:,2), pos(:,3), '-.', ...
        posVO(:,1), posVO(:,2), posVO(:,3), ...
        posEst(:,1), posEst(:,2), posEst(:,3), ...
        'LineWidth', 3)
    legend('Ground Truth', 'Visual Odometry (VO)', ...
        'Visual-Inertial Odometry (VIO)', 'Location', 'northeast')
else
    plot3(pos(:,1), pos(:,2), pos(:,3), '-.', ...
        posEst(:,1), posEst(:,2), posEst(:,3), ...
        'LineWidth', 3)
    legend('Ground Truth', 'IMU Pose Estimate')
end
view(-90, 90)
title('Vehicle Position')
xlabel('X (m)')
ylabel('Y (m)')
grid on
```
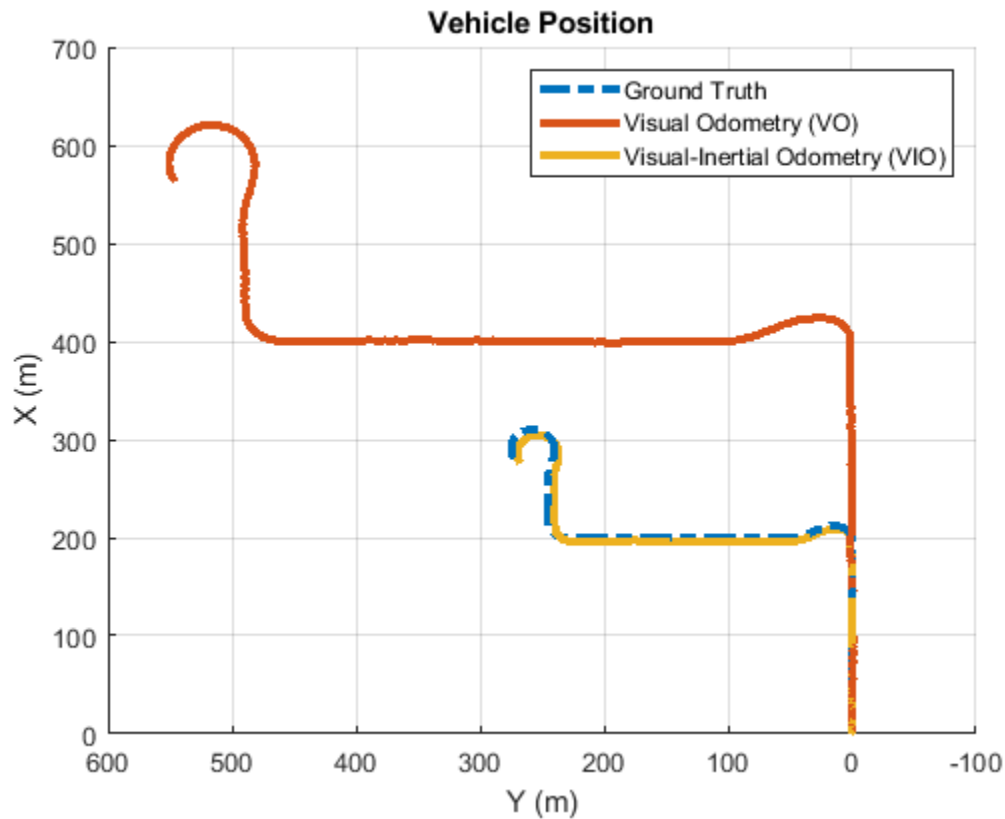
The plot shows that the visual odometry estimate is relatively accurate in estimating the shape of the trajectory. The fusion of the IMU and visual odometry measurements removes the scale factor uncertainty from the visual odometry measurements and the drift from the IMU measurements.

Supporting Functions

helperVisualOdometryModel

Compute visual odometry measurement from ground truth input and parameters struct. To model the uncertainty in the scaling between subsequent frames of the monocular camera, a constant scaling factor combined with a random drift is applied to the ground truth position.

```
function [posVO, orientVO, paramsVO] ...
    = helperVisualOdometryModel(pos, orient, paramsVO)

% Extract model parameters.
scaleVO = paramsVO.scale;
sigmaN = paramsVO.sigmaN;
tau = paramsVO.tau;
sigmaB = paramsVO.sigmaB;
sigmaA = sqrt((2/tau) + 1/(tau*tau))*sigmaB;
b = paramsVO.driftBias;

% Calculate drift.
b = (1 - 1/tau).*b + randn(1,3)*sigmaA;
drift = randn(1,3)*sigmaN + b;
paramsVO.driftBias = b;
```

```
% Calculate visual odometry measurements.
posVO = scaleVO*pos + drift;
orientVO = orient;
end
```

helperInitialize

Set the initial state and covariance values for the fusion filter.

```
function helperInitialize(filt, traj)

% Retrieve the initial position, orientation, and velocity from the
% trajectory object and reset the internal states.
[pos, orient, vel] = traj();
reset(traj);

% Set the initial state values.
filt.State(1:4) = compact(orient(1)).';
filt.State(5:7) = pos(1,:).';
filt.State(8:10) = vel(1,:).';

% Set the gyroscope bias and visual odometry scale factor covariance to
% large values corresponding to low confidence.
filt.StateCovariance(10:12,10:12) = 1e6;
filt.StateCovariance(end) = 2e2;
end
```

helperPreallocateData

Preallocate data to log simulation results.

```
function [pos, orient, vel, acc, angvel, ...
    posVO, orientVO, ...
    posEst, orientEst, velEst] ...
    = helperPreallocateData(numIMUSamples, numCameraSamples)

% Specify ground truth.
pos = zeros(numIMUSamples, 3);
orient = quaternion.zeros(numIMUSamples, 1);
vel = zeros(numIMUSamples, 3);
acc = zeros(numIMUSamples, 3);
angvel = zeros(numIMUSamples, 3);

% Visual odometry output.
posVO = zeros(numCameraSamples, 3);
orientVO = quaternion.zeros(numCameraSamples, 1);

% Filter output.
posEst = zeros(numIMUSamples, 3);
orientEst = quaternion.zeros(numIMUSamples, 1);
velEst = zeros(numIMUSamples, 3);
end
```

helperUpdatePose

Update the pose of the vehicle.

```
function helperUpdatePose(veh, pos, vel, orient)

veh.Position = pos;
veh.Velocity = vel;
rpy = eulerd(orient, 'ZYX', 'frame');
veh.Yaw = rpy(1);
veh.Pitch = rpy(2);
veh.Roll = rpy(3);
end
```

References

- Sola, J. "Quaternion Kinematics for the Error-State Kalman Filter." ArXiv e-prints, arXiv:1711.02508v1 [cs.RO] 3 Nov 2017.
- R. Jiang, R., R. Klette, and S. Wang. "Modeling of Unbounded Long-Range Drift in Visual Odometry." 2010 Fourth Pacific-Rim Symposium on Image and Video Technology. Nov. 2010, pp. 121-126.

Tracking Maneuvering Targets

This example shows how to track maneuvering targets using various tracking filters. The example shows the difference between filters that use a single motion model and multiple motion models.

Define a Scenario

In this example, you define a single target that initially travels at a constant velocity of 200 m/s for 33 seconds, then enters a constant turn of 10 deg/s. The turn lasts for 33 seconds, then the target accelerates in a straight line at 3 m/s^2.

```
[trueState, time, fig1] = helperGenerateTruthData;
dt = diff(time(1:2));
numSteps = numel(time);
figure(fig1)
```
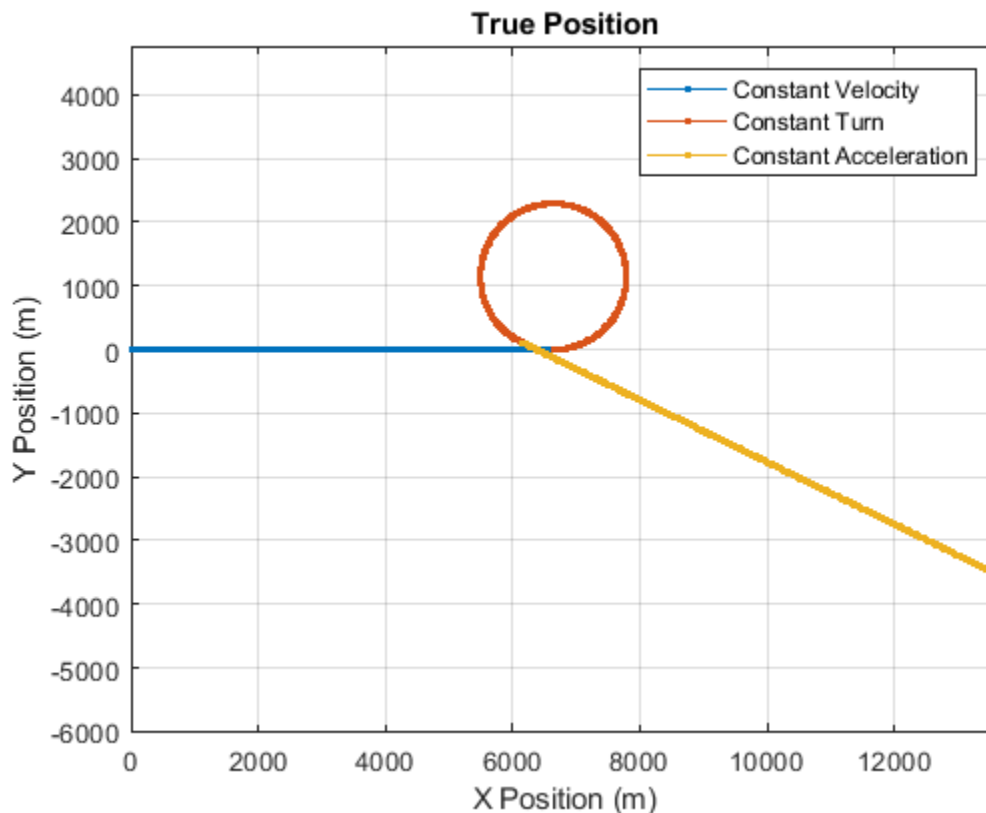

Define the measurements to be the position and add normal random noise with a standard deviation of 1 to the measurements.

```
% Set the RNG seed for repeatable results
s = rng;
rng(2018);
positionSelector = [1 0 0 0 0 0;0 0 1 0 0 0;0 0 0 0 1 0]; % Position from state
truePos = positionSelector * trueState;
measNoise = randn(size(truePos));
measPos = truePos + measNoise;
```

Track Using a Constant-Velocity Filter

You define a `trackingEKF` with a constant-velocity motion model. You use the first measurement to define the initial state and state covariance, and set the process noise to be non-additive, to define the process noise in terms of the unknown acceleration in the x, y, and z components. This definition is similar to how the function `initcvekf` works.

```
initialState = positionSelector' * measPos(:,1);
initialCovariance = diag([1,1e4,1,1e4,1,1e4]); % Velocity is not measured
cvekf = trackingEKF(@constvel, @cvmeas, initialState, ...
    'StateTransitionJacobianFcn', @constveljac, ...
    'MeasurementJacobianFcn', @cvmeasjac, ...
    'StateCovariance', initialCovariance, ...
    'HasAdditiveProcessNoise', false, ...
    'ProcessNoise', eye(3));
```

For each measurement, you predict the filter, calculate the distance of the predicted state from the true position, and correct the filter using the measurement to obtain the filtered estimate of the position.

```
dist = zeros(1,numSteps);
estPos = zeros(3,numSteps);
for i = 2:size(measPos,2)
    predict(cvekf, dt);
    dist(i) = distance(cvekf,truePos(:,i)); % Distance from true position
    estPos(:,i) = positionSelector * correct(cvekf, measPos(:,i));
end
figure(fig1);
plot(estPos(1,:),estPos(2,:),'.g','DisplayName','CV Low PN')
title('True and Estimated Positions')
axis([5000 8000 -500 2500])
```

True and Estimated Positions
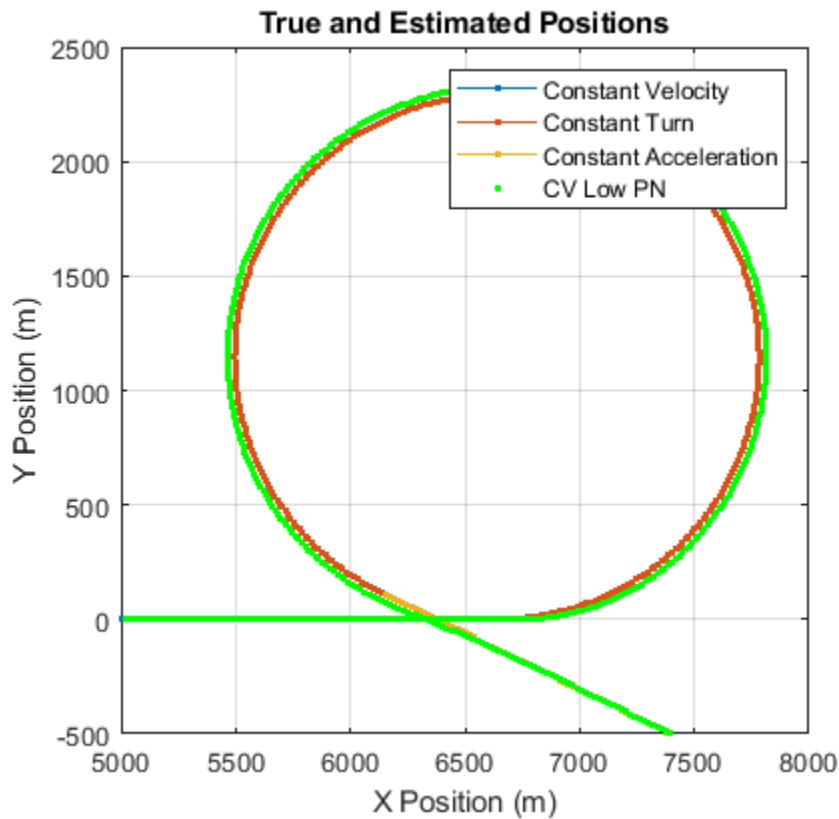
As depicted in the figure, the filter is able to track the constant velocity part of the motion very well, but when the target executes the turn, the filter estimated position diverges from the true position. You can see the distance of the estimate from the truth in the following plot. During the turn, at 33-66 seconds, the normalized distance jumps to very high values, which means that the filter is unable to track the maneuvering target.

```
fig2 = figure;
hold on
plot((1:numSteps)*dt,dist,'g','DisplayName', 'CV Low PN')
title('Normalized Distance From Estimated Position to True Position')
xlabel('Time (s)')
ylabel('Normalized Distance')
legend
```
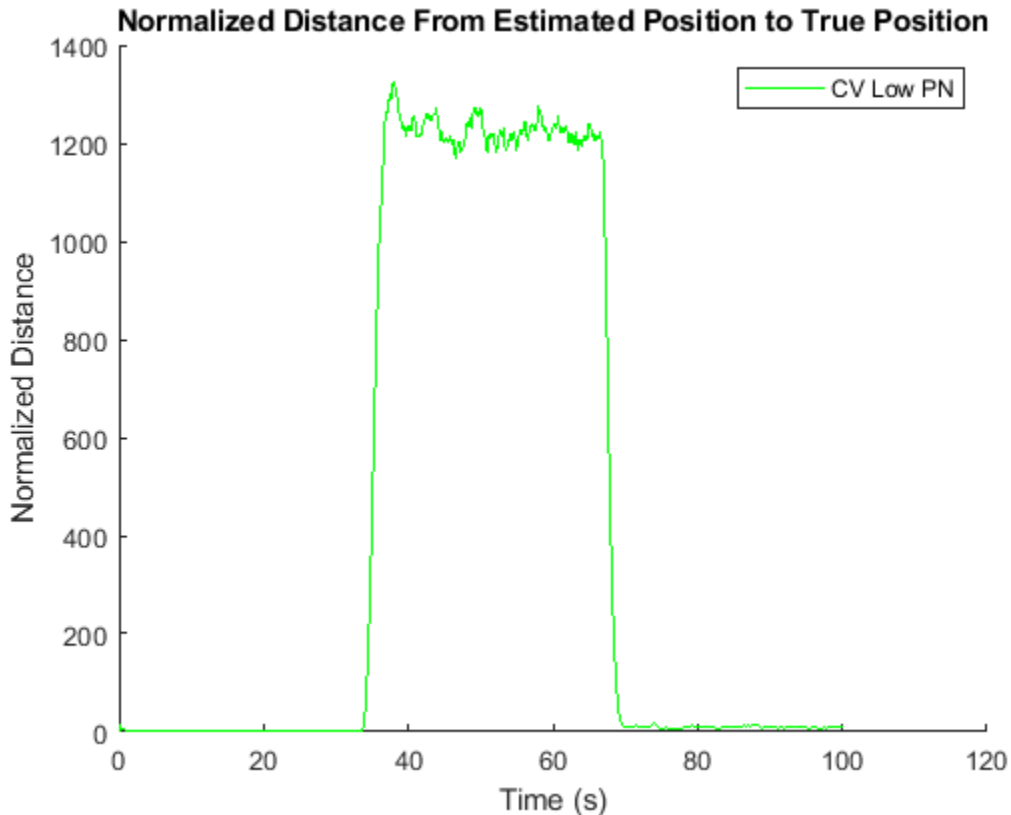
Increase the Process Noise

One possible solution is to increase the process noise. The process noise represents the unmodeled terms in the motion model. For a constant velocity model, these are the unknown acceleration terms. By increasing the process noise, you allow for larger uncertainty in the motion model, which causes the filter to rely on the measurements more than on the model. The following lines create a constant velocity filter, with high process noise values that correspond to about 5-G turn.

```
cvekf2 = trackingEKF(@constvel, @cvmeas, initialState, ...
    'StateTransitionJacobianFcn', @constveljac, ...
    'MeasurementJacobianFcn', @cvmeasjac, ...
    'StateCovariance', initialCovariance, ...
    'HasAdditiveProcessNoise', false, ...
    'ProcessNoise', diag([50,50,1])); % Large uncertainty in the horizontal acceleration
dist = zeros(1,numSteps);
estPos = zeros(3,numSteps);
for i = 2:size(measPos,2)
    predict(cvekf2, dt);
    dist(i) = distance(cvekf2,truePos(:,i)); % Distance from true position
    estPos(:,i) = positionSelector * correct(cvekf2, measPos(:,i));
end
figure(fig1)
plot(estPos(1,:),estPos(2,:),'.c','DisplayName','CV High PN')
axis([5000 8000 -500 2500])
```

True and Estimated Positions
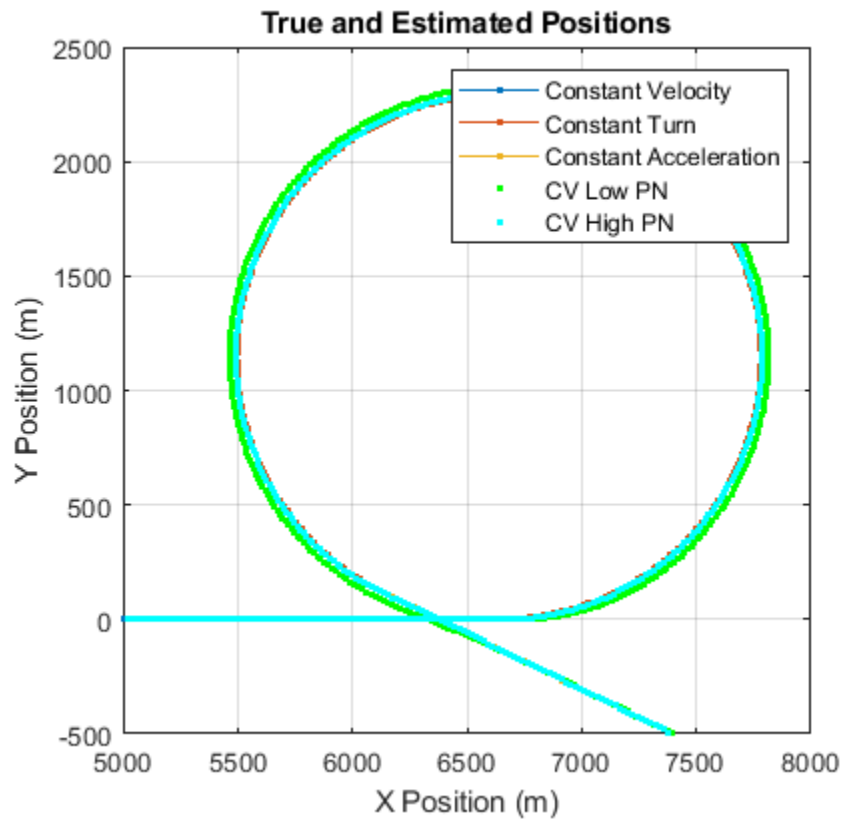
Increasing the process noise significantly improves the filter's ability to track the target during the turn. However, there is a cost: the filter is less capable of smoothing out the measurement noise during the constant velocity period of the motion. Even though the normalized distance during the turn was significantly reduced, the normalized distance increased in the first 33 seconds, during the constant velocity period of the motion.

```
figure(fig2)
plot((1:numSteps)*dt,dist,'c','DisplayName', 'CV High PN')
axis([0 100 0 50])
```

Normalized Distance From Estimated Position to True Position
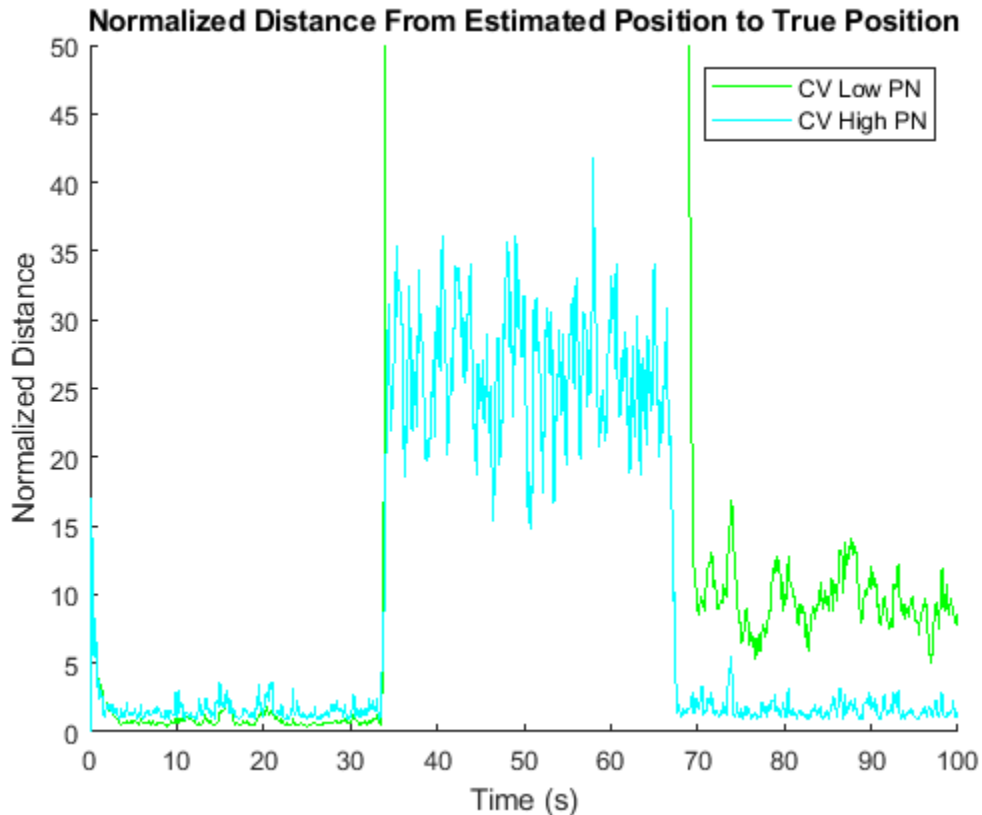
Use an Interacting Motion-Model Filter

Another solution is to use a filter that can consider all motion models at the same time, called an interacting multiple-model (IMM) filter. The IMM filter can maintain as many motion models as you want, but typically is used with 2-5 motion models. For this example, three models are sufficient: a constant velocity model, a constant turn model, and a constant acceleration model.

```
imm = trackingIMM('TransitionProbabilities', 0.99); % The default IMM has all three models
% Initialize the state and state covariance in terms of the first model
initialize(imm, initialState, initialCovariance);
```

You use the IMM filter in the same way that the EKF was used.

```
dist = zeros(1,numSteps);
estPos = zeros(3,numSteps);
modelProbs = zeros(3,numSteps);
modelProbs(:,1) = imm.ModelProbabilities;
for i = 2:size(measPos,2)
    predict(imm, dt);
    dist(i) = distance(imm,truePos(:,i)); % Distance from true position
    estPos(:,i) = positionSelector * correct(imm, measPos(:,i));
    modelProbs(:,i) = imm.ModelProbabilities;
end
figure(fig1)
plot(estPos(1,:),estPos(2,:),'.m','DisplayName','IMM')
```
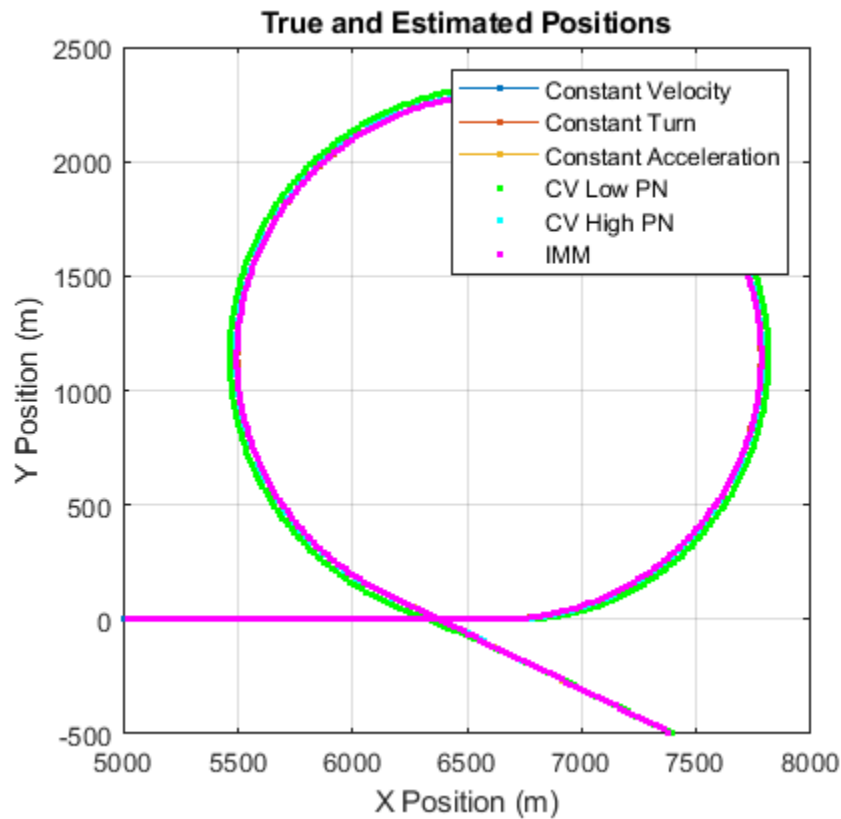
The `trackingIMM` filter is able to track the maneuvering target in all three parts of the motion.

Examining the distance between the filter's predicted state and the true position, you see that the IMM filter is able to reduce the distance for all portions of the motion. In fact, the IMM filter is better at tracking the motion than both other constant velocity models used earlier.

```
figure(fig2)
hold on
plot((1:numSteps)*dt,dist,'m','DisplayName', 'IMM')
axis([0 100 0 50])
```
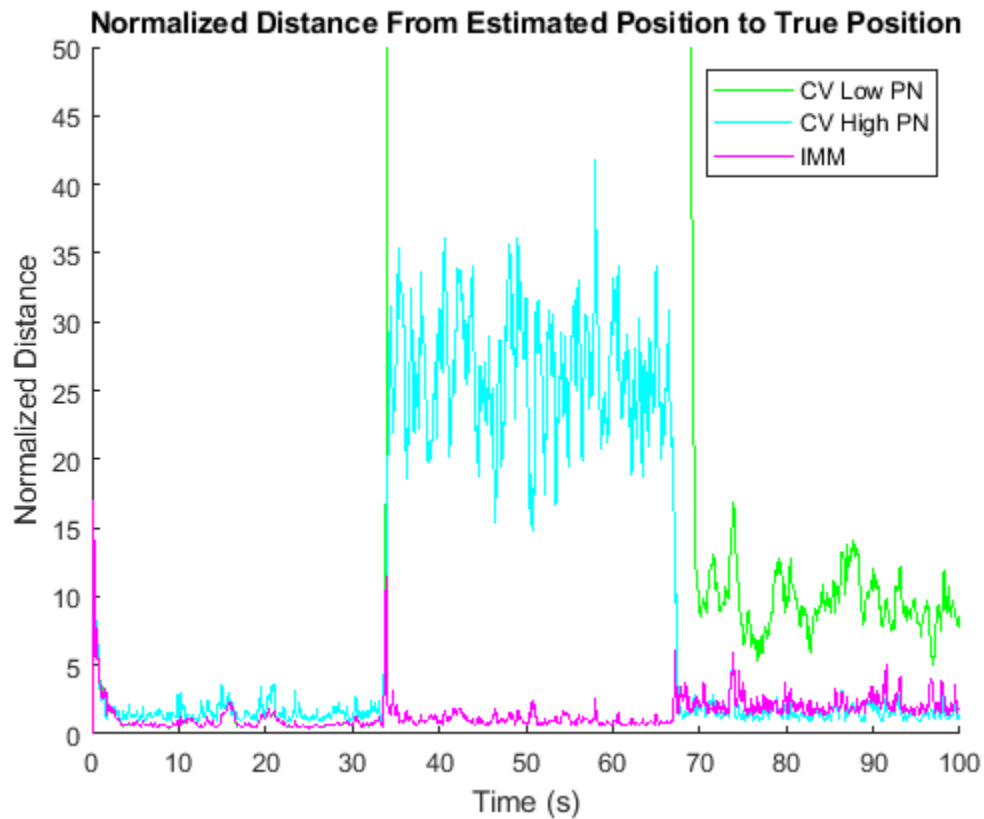
To better understand how the IMM filter works, plot the model probabilities as a function of time. The figure shows that filter is initialized with the three models having the same probability. As the filter is updated, it quickly converges to a very high probability that the model is a constant velocity model. After 33 seconds of motion, the constant velocity model is no longer true, and the probability of the constant turn model becomes very high for the duration of the turn. In the last section of the motion, during the constant acceleration maneuver, the IMM filter assigns a high probability that the motion is constant acceleration, but the filter is less certain about the correct motion model, and there is about 0.3 probability of a constant velocity motion.

```
figure
plot((1:numSteps)*dt, modelProbs)
title('Model Probabilities vs. Time')
xlabel('Time (s)')
ylabel('Model Probabilities')
legend('IMM-CV','IMM-CA','IMM-CT')
% Return the RNG to its previous state
rng(s)
```
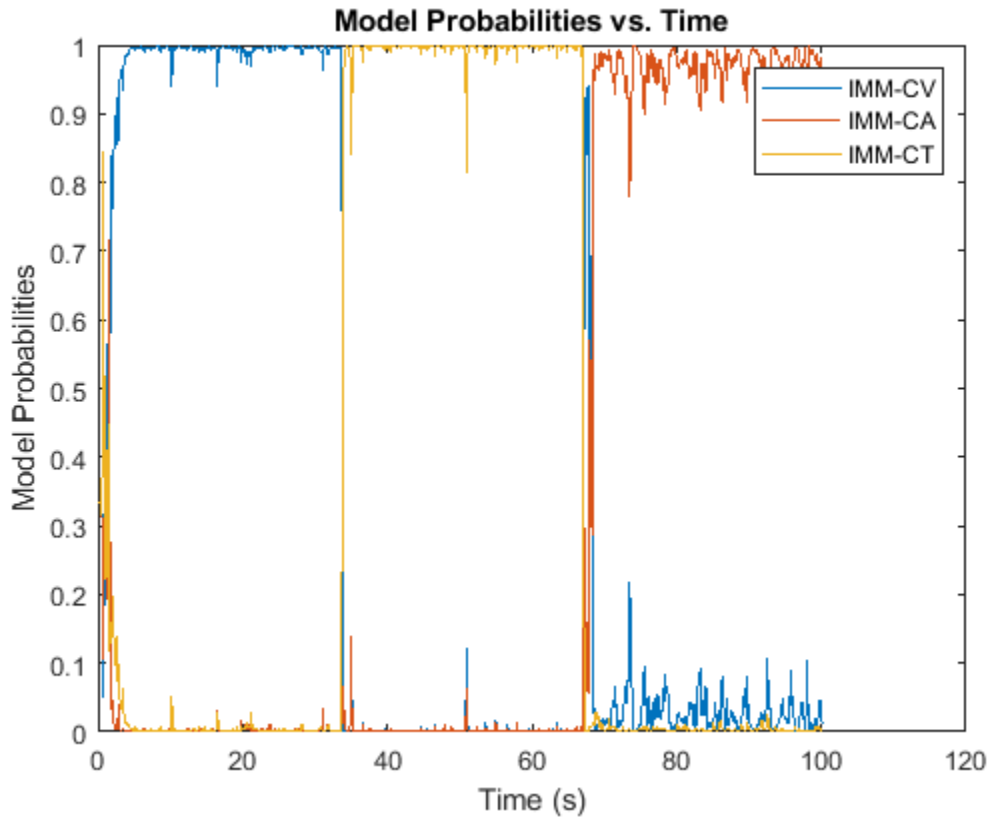
Summary

This example showed you how to track a target maneuvering with constant turn and constant acceleration motion. The example showed how you can increase the process noise to capture the unknown maneuver with a constant velocity model. You also saw how to improve the tracking of a maneuvering target by using an IMM filter.

Supporting Functions

helperGenerateTruthData

This function generates the ground truth trajectory.

```
function [Xgt, tt, figs] = helperGenerateTruthData
% Generate ground truth
vx = 200; % m/s
omega = 10; % deg/s
acc = 3; % m/s/s
dt = 0.1;
tt = (0:dt:floor(1000*dt));
figs = [];

Xgt = NaN(9,numel(tt));
Xgt(:,1) = 0;

% Constant velocity
seg1 = floor(numel(tt)/3);
```

```matlab
Xgt(2,1) = vx;
slct = eye(9);
slct(3:3:end,:) = [];
for m = 2:seg1
    X0 = slct*Xgt(:,m-1);
    X1 = constvel(X0, dt);
    X1 = slct'*X1;
    Xgt(:,m) = X1;
end

% Constant turn
seg2 = floor(2*numel(tt)/3);
slct = eye(9);
slct(3:3:end,:) = [];
for m = seg1+1:seg2
    X0 = slct*Xgt(:,m-1);
    X0 = [X0(1:4);omega];
    X1 = constturn(X0, dt);
    X1 = X1(1:4);
    X1 = [X1(1:2);0;X1(3:4);0;zeros(3,1)];
    Xgt(:,m) = X1;
end

% Constant acceleration
first = true;
for m = seg2+1:numel(tt)
    X0 = Xgt(:,m-1);
    if first
        vel = X0(2:3:end);
        ua = vel/norm(vel);
        va = acc*ua;
        X0(3:3:end) = va;
        first = false;
    end
    X1 = constacc(X0, dt);
    Xgt(:,m) = X1;
end

% Drop acceleration dimension
slct = eye(9);
slct(3:3:end,:) = [];
Xgt = slct*Xgt;

figs = [figs figure];
plot(Xgt(1,1:seg1),Xgt(3,1:seg1),'.-');
hold on;
plot(Xgt(1,seg1+1:seg2),Xgt(3,seg1+1:seg2),'.-');
plot(Xgt(1,seg2+1:end),Xgt(3,seg2+1:end),'.-');
grid on;
xlabel('X Position (m)');
ylabel('Y Position (m)');
title('True Position')
axis equal;
legend('Constant Velocity', 'Constant Turn', 'Constant Acceleration')
end
```

Multiplatform Radar Detection Fusion

This example shows how to fuse radar detections from a multiplatform radar network. The network includes two airborne and one ground-based long-range radar platforms. See the "Multiplatform Radar Detection Fusion" on page 5-202 example for details. A central tracker processes the detections from all platforms at a fixed update interval. This enables you to evaluate the network's performance against target types, platform maneuvers, as well as platform configurations and locations..

Load a recording of a tracking scenario

The `MultiplatformRadarDetectionGeneration` MAT-file contains a tracking scenario recording previously generated using the following command

```
recording = record(scene,'IncludeSensors',true,'InitialSeed',2018,'RecordingFormat','Recording')
```

where `scene` is the tracking scenario created in the Multiplatform Radar Detection Generation example.

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

Define Central Tracker

Use the `trackerGNN` as a central tracker that processes detections received from all radar platforms in the scenario.

The tracker uses the `initFilter` supporting function to initialize a constant velocity extended Kalman filter for each new track. `initFilter` modifies the filter returned by `initcvekf` to match the high target velocities. The filter's process noise is set to $1g$ ($g = 9.8 \, \text{m/s}^2$) to enable tracking of maneuvering targets in the scenario.

The tracker's `AssignmentThreshold` is set to 50 to enable detections with large range biases (due to atmospheric refraction effects at long detection ranges) to be associated with tracks in the tracker. The `DeletionThreshold` is set to 3 to delete redundant tracks quickly.

Enable the `HasDetectableTrackIDsInput` to specify the tracks that are within the field of view of at least one radar since the last update. Track logic is only evaluated on tracks which had a detection opportunity since the last tracker update.

```
tracker = trackerGNN('FilterInitializationFcn', @initFilter, ...
        'AssignmentThreshold', 50, 'DeletionThreshold', 3, ...
        'HasDetectableTrackIDsInput', true);
```

Track Targets by Fusing Detections in a Central Tracker

The following loop runs the tracking scenario recording until the end of the scenario. For each step forward in the scenario, detections are collected for processing by the central tracker. The tracker is updated with these detections every 2 seconds.

```
trackUpdateRate = 0.5;    % Update the tracker every 2 seconds

% Create a display to show the true, measured, and tracked positions of the
% detected targets and platforms.
theaterDisplay = helperMultiPlatFusionDisplay(recording,'PlotAssignmentMetrics', true);

% Construct an object to analyze assignment and error metrics
```

```matlab
tam = trackAssignmentMetrics('DistanceFunctionFormat','custom',...
    'AssignmentDistanceFcn',@truthAssignmentDistance,...
    'DivergenceDistanceFcn',@truthAssignmentDistance);

% Initialize buffers
detBuffer = {};
sensorConfigBuffer = [];
allTracks = [];
detectableTrackIDs = uint32([]);
assignmentTable = [];

% Initialize next tracker update time
nextTrackUpdateTime = 2;

while ~isDone(recording)
    % Read the next record of the recording.
    [time, truePoses, covcon, dets, senscon, sensPlatIDs] = read(recording);

    % Buffer all detections and sensor configurations until the next tracker update
    detBuffer = [detBuffer ; dets]; %#ok<AGROW>
    sensorConfigBuffer = [sensorConfigBuffer ; senscon']; %#ok<AGROW>

    % Follow the trackUpdateRate to update the tracker
    if time >= nextTrackUpdateTime || isDone(recording)

        if isempty(detBuffer)
            lastDetectionTime = time;
        else
            lastDetectionTime = detBuffer{end}.Time;
        end

        if isLocked(tracker)
            % Collect list of tracks which fell within at least one radar's field
            % of view since the last tracker update
            predictedtracks = predictTracksToTime(tracker, 'all', lastDetectionTime);
            detectableTrackIDs = detectableTracks(allTracks, predictedtracks, sensorConfigBuffer)
        end

        % Update tracker. Only run track logic on tracks that fell within at
        % least one radar's field of view since the last tracker update
        [confirmedTracks, ~, allTracks] = tracker(detBuffer, lastDetectionTime, detectableTrackID

        % Analyze and retrieve the current track-to-truth assignment metrics
        tam(confirmedTracks, truePoses);

        % Store assignment metrics in a table
        currentAssignmentTable = trackMetricsTable(tam);
        rowTimes = seconds(time*ones(size(currentAssignmentTable,1),1));
        assignmentTable = cat(1,assignmentTable,table2timetable(currentAssignmentTable,'RowTimes

        % Update display with detections, coverages, and tracks
        theaterDisplay(detBuffer, covcon, confirmedTracks, assignmentTable, truePoses, sensPlatID

        % Empty buffers
        detBuffer = {};
        sensorConfigBuffer = [];

        % Update next track update time
```

```
            nextTrackUpdateTime = nextTrackUpdateTime + 1/trackUpdateRate;
        end

    end
```
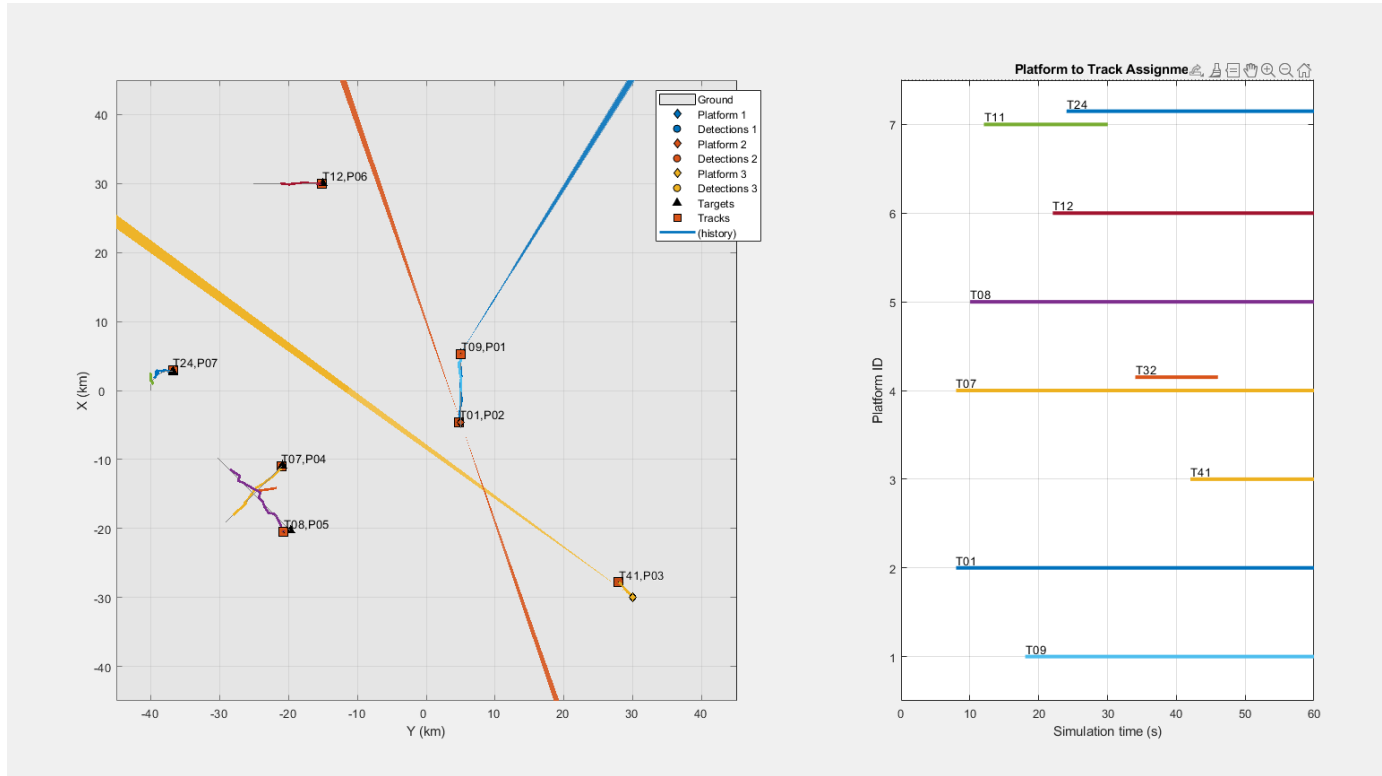

At the end of the scenario, you see that multiple tracks have been dropped and replaced. You can also see the association of tracks to platforms for the duration of the scenario. The plot has seven rows for seven platforms in the scenario. Each track is shown as a horizontal line. Track numbers are annotated at the beginning of the lines. Whenever a track is deleted, its line stops. Whenever a new track is assigned to a platform, a new line is added to the platform's row, when multiple lines are shown at the same time for a single platform, the platform has multiple tracks assigned to it. In these cases, the newer track associated with the platform is considered as *redundant*.

```
endTime = assignmentTable.Time(end);
assignmentTable(endTime,{'TrackID','AssignedTruthID','TotalLength','DivergenceCount','RedundancyC
```

ans=9×6 timetable

Time	TrackID	AssignedTruthID	TotalLength	DivergenceCount	RedundancyCount
60 sec	1	2	27	0	0
60 sec	7	4	27	0	0
60 sec	8	5	26	0	0
60 sec	9	1	22	0	0
60 sec	11	NaN	10	0	0
60 sec	12	6	20	0	0
60 sec	24	7	19	0	1
60 sec	32	NaN	7	0	1
60 sec	41	3	10	0	0

Notice that the platforms which have difficulties in maintaining tracks (platforms 4 and 7) are also the platforms furthest from the radars. This poor tracking performance is attributed to the Gaussian distribution assumption for the measurement noise. The assumption works well for targets at short ranges, but at long ranges, the measurement uncertainty deviates from a Gaussian distribution. The following figure compares the 1-sigma covariance ellipses corresponding to actual target distribution and the distribution of the target given by a radar sensor. The sensor is 5 km away from the target with an angular resolution of 5 degrees. The actual measurement uncertainty has a concave shape resulting from the spherical sensor detection coordinate frame in which the radar estimates the target's position.

```
maxCondNum = 300;
figure;
helperPlotLongRangeCorrection(maxCondNum)
```
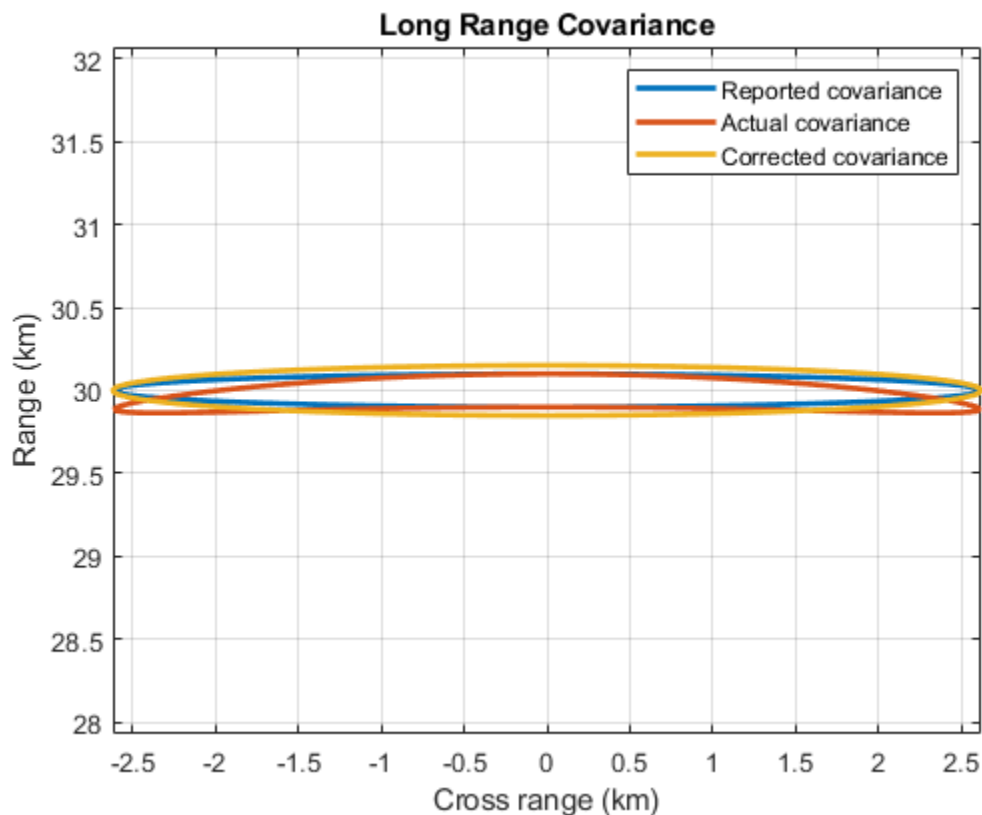

To account for the concave shape of the actual covariance at long ranges, the `longRangeCorrection` supporting function constrains the condition number of the reported measurement noise. The corrected measurement covariance shown above is constrained to a maximum condition number of 300. In other words, no eigenvalue in the measurement covariance can be more than 300 times smaller than the covariance's largest eigenvalue. This treatment expands the measurement noise along the range dimension to better match the concavity of the actual measurement uncertainty.

Simulate with Long-Range Covariance Correction

Rerun the previous simulation using the `longRangeCorrection` supporting function to correct the reported measurement noise at long ranges.

```
[confirmedTracks,correctedAssignmentTable,ctheaterDisplay] = ...
    runMultiPlatFusionSim(recording,tracker,@longRangeCorrection);
```
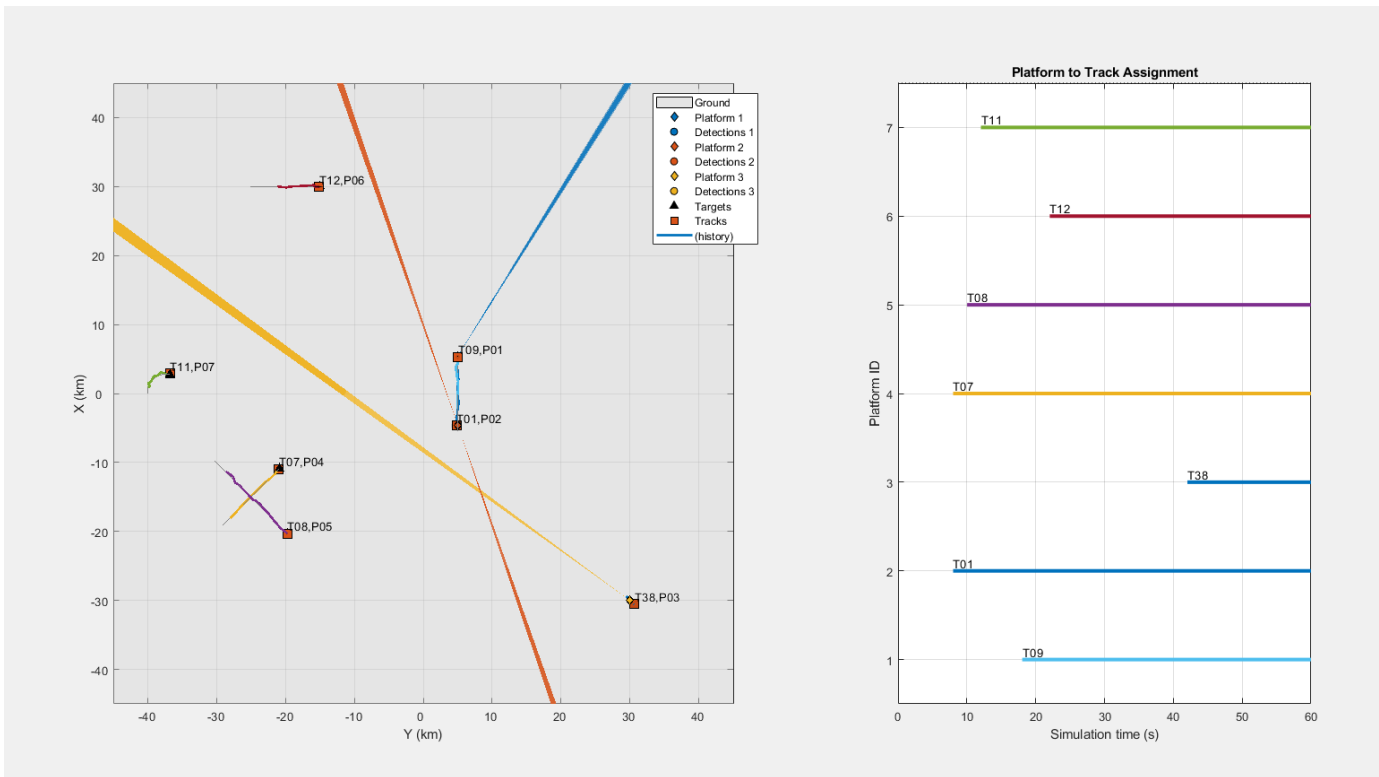


```
endTime = correctedAssignmentTable.Time(end);
correctedAssignmentTable(endTime,{'TrackID','AssignedTruthID','TotalLength','DivergenceCount','Re
```

```
ans=7×6 timetable
    Time      TrackID    AssignedTruthID    TotalLength    DivergenceCount    RedundancyCount
    _____    _____    _____    _____    _____    _____

    60 sec       1             2                27                0                  0
    60 sec       7             4                27                0                  0
    60 sec       8             5                26                0                  0
    60 sec       9             1                22                0                  0
    60 sec      11             7                25                0                  0
    60 sec      12             6                20                0                  0
    60 sec      38             3                10                0                  0
```

The preceding figure shows that by applying the long-range correction, no track-drops or multiple
tracks are generated for the entire scenario. In this case, there is exactly one track for each platform
detected by the surveillance network.

```
allDetections = vertcat(recording.RecordedData.Detections);
ctheaterDisplay(allDetections,covcon,confirmedTracks,correctedAssignmentTable,truePoses, sensPlat
axes(ctheaterDisplay.TheaterPlot.Parent)
legend('off')
xlim([-1000 5000]); ylim([-41000 -36000]); zlim([-5000 0]);
view([-90 90])
```

```
axis square
title('Jet Executing Horizontal Turn')
```
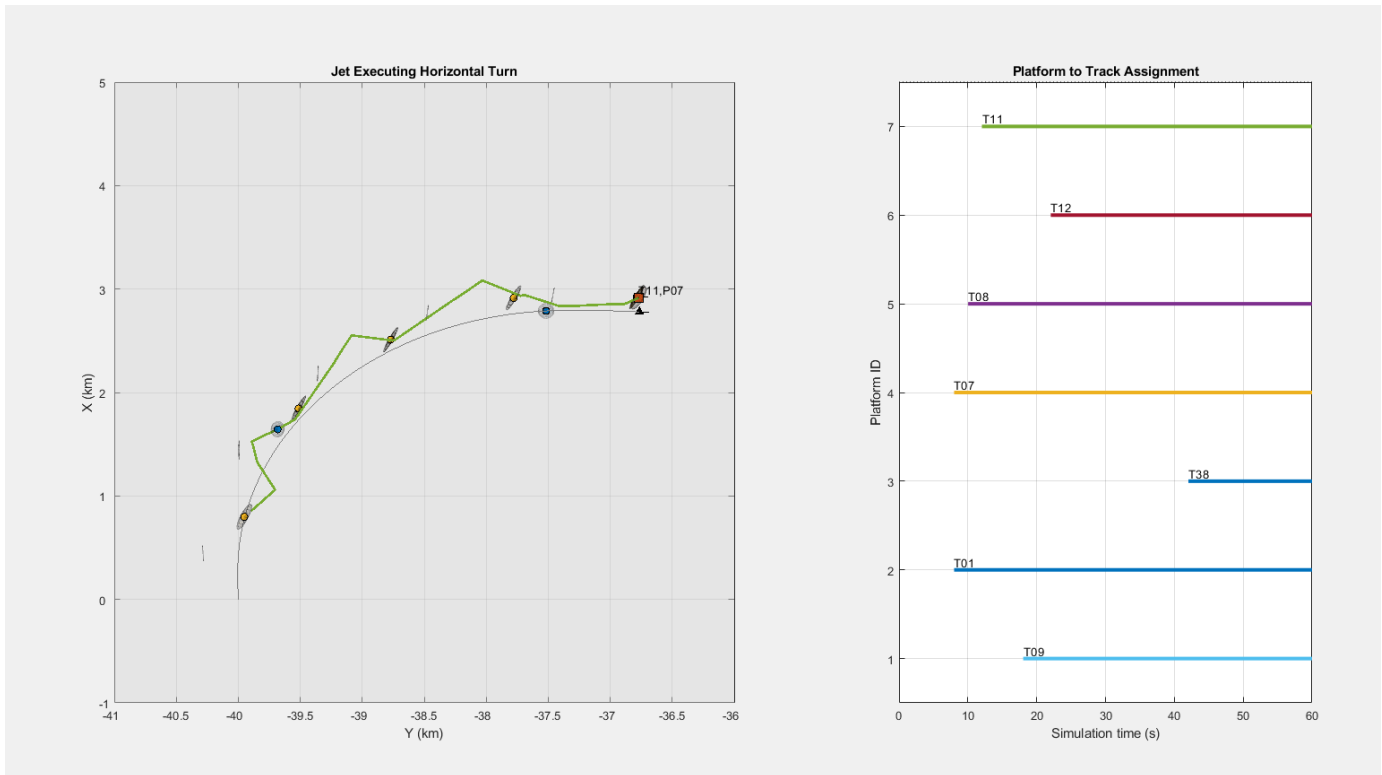

Zoom in the view in which the jet is executing a horizontal turn, the track follows the maneuvering target relatively well, even though the motion model used in this example is constant velocity. Tracking the maneuver could be further improved by using an interacting multiple-model (IMM) filter such as the `trackingIMM` filter.

```
view([-60 25])
```
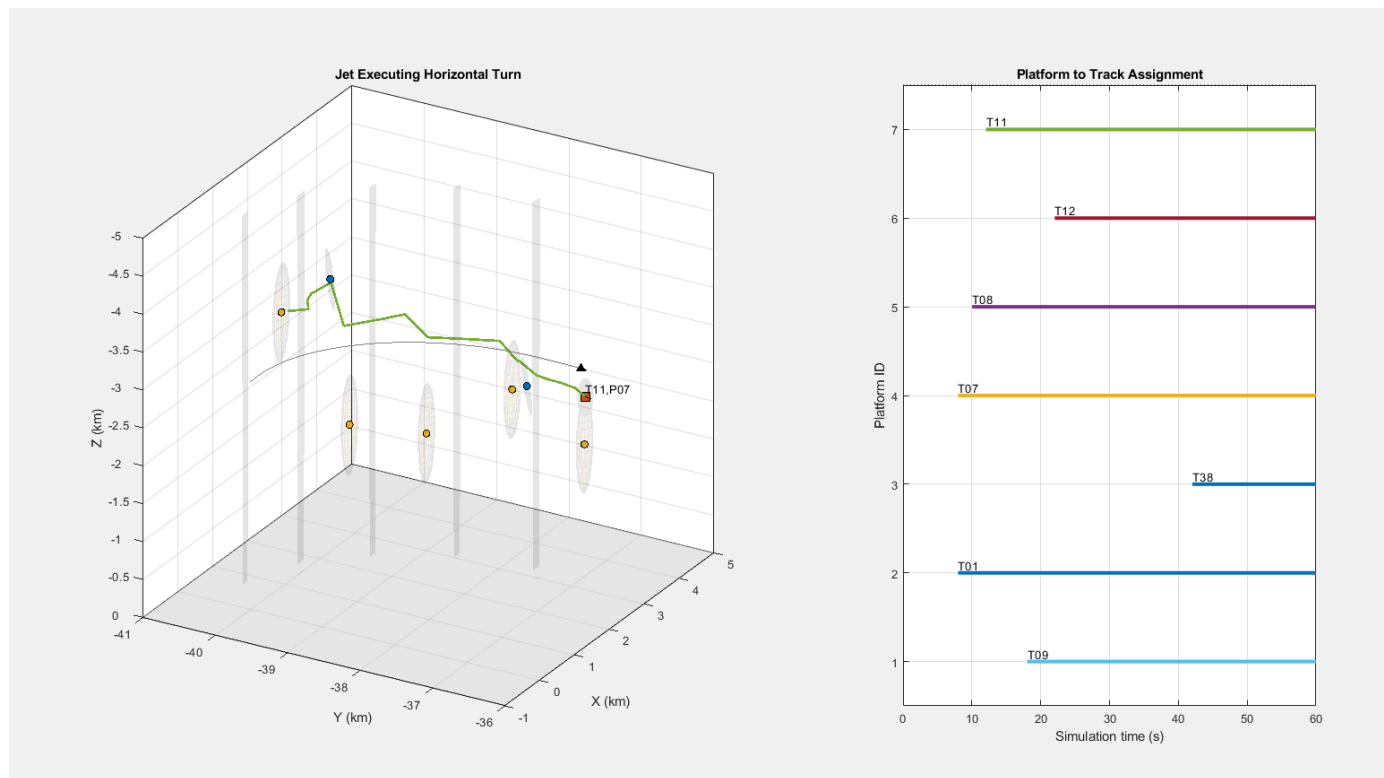
From another view in which the jet is executing a horizontal turn, you can see that the altitude is estimated correctly, despite the inaccurate altitude measurements from the sensors. One of the sensors does not report altitude at all, as seen by the large vertical ellipsoids, while the other two sensors underestimate their uncertainty in the altitude.

```
xlim([-25000 -9000]); ylim([-31000 -19000]); zlim([-9000 -2000]);
view([-45 10])
title('Crossing Airliners')
```

Crossing Airliners / Platform to Track Assignment
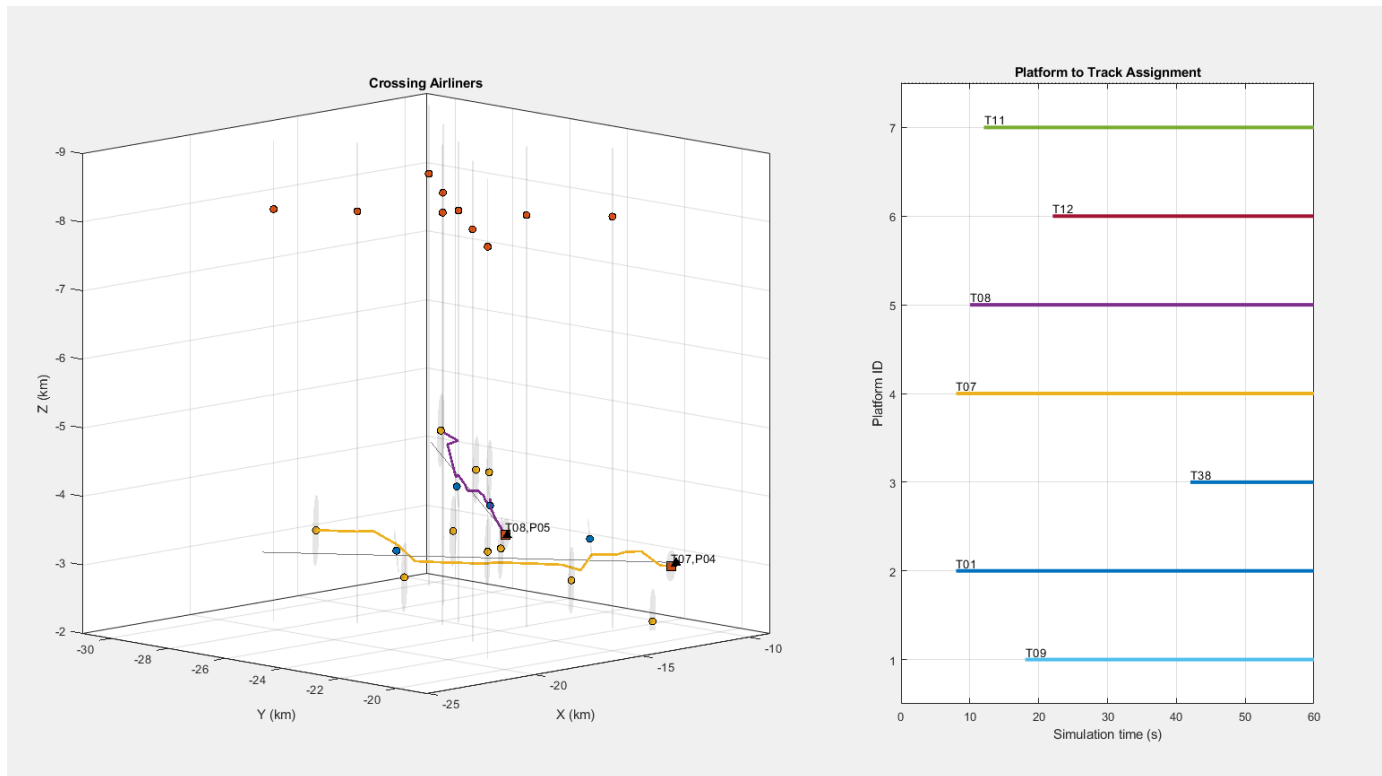
Switching the point of view to focus on the crossing airliners, the same inaccurate altitude measurements are depicted. Note how the red detections are centered at an altitude of 8 km, while the two airliners fly at altitudes of 3 and 4 km, respectively. The use of a very large covariance in the altitude allows the tracker to ignore the erroneous altitude reading from the red detections and keep track of the altitude using the other two radars. Observing the uncertainty covariance of tracks T07 and T08, you can see that they provide a consistent estimate of platforms P04 and P05, respectively.

```
xlim([-10000 10000]); ylim([-0 10000]); zlim([-12000 -5000]);
view([-15 10])
title('Airborne Radar Platforms')
```
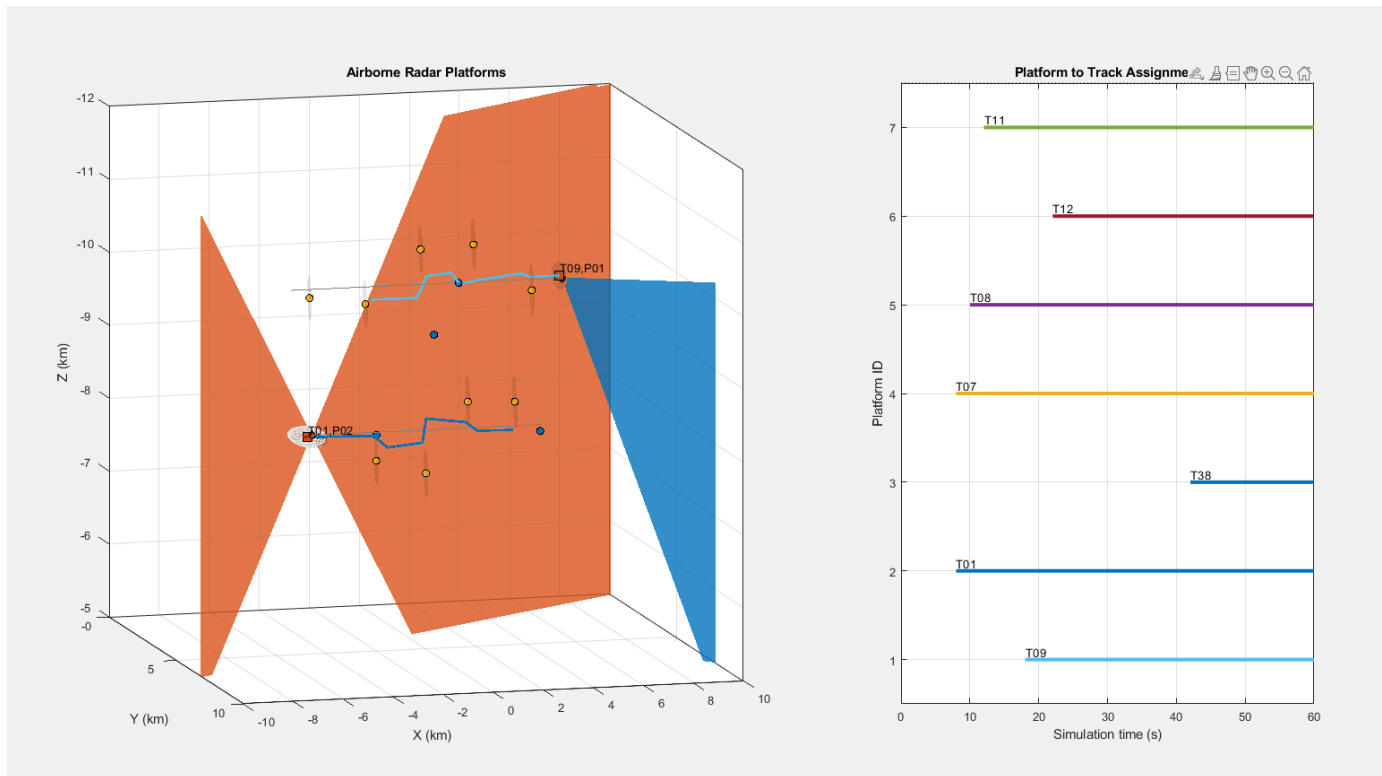
The last plot focuses on the two airborne radar platforms. Each platform is detected by the other platform as well as by the ground radar. The platform trajectories cross each other, separated by 1000 m in altitude, and their tracks are consistent with the ground truth.

Summary

This example shows how to process detections collected across multiple airborne and ground-based radar platforms in a central tracker. In this example, you learned how the measurement noise at long ranges is not accurately modeled by a Gaussian distribution. The concavity of the 1-sigma ellipse of the measurement noise at these long ranges results in poor tracking performance with dropped tracks and multiple tracks assigned to a single platform. You also learned how to correct the measurement noise for detections at long ranges to improve the continuity of the reported tracks.

Supporting Functions

`initFilter` This function modifies the function `initcvekf` to handle higher velocity targets such as the airliners in the scenario.

```
function filter = initFilter(detection)
filter = initcvekf(detection);
classToUse = class(filter.StateCovariance);

% Airliners can move at speeds around 900 km/h. The velocity is initialized
% to 0, but will need to be able to quickly adapt to aircraft moving at
% these speeds. Use 900 km/h as 1 standard deviation for the velocity
% noise.
spd = 900*1e3/3600; % m/s
velCov = cast(spd^2,classToUse);
cov = filter.StateCovariance;
```

```
cov(2,2) = velCov;
cov(4,4) = velCov;
filter.StateCovariance = cov;

% Set filter's process noise to allow for some horizontal maneuver
scaleAccel = cast(10,classToUse);
Q = blkdiag(scaleAccel^2, scaleAccel^2, 1);
filter.ProcessNoise = Q;
end
```

detectableTracks This function returns the IDs for tracks that fell within at least one sensor's field of view. The sensor's field of view and orientation relative to the coordinate frame of the tracks is stored in the array of sensor configuration structs. The configuration structs are returned by the `monostaticRadarSensor` and can be used to transform track positions and velocities to the sensor's coordinate frame.

```
function trackIDs = detectableTracks(tracks,predictedtracks,configs)
% Identify which tracks fell within a sensor's field of view

numTrks = size(tracks,1);
[numsteps, numSensors] = size(configs);
allposTrack = zeros(3,numsteps);
isDetectable = false(numTrks,1);
for iTrk = 1:numTrks
    % Interpolate track positions between current position and predicted
    % positions for each simulation step
    posTrack = tracks(iTrk).State(1:2:end);
    posPreditedTrack = predictedtracks(iTrk).State(1:2:end);
    for iPos = 1:3
        allposTrack(iPos,:) = linspace(posTrack(iPos),posPreditedTrack(iPos),numsteps);
    end
    for iSensor = 1:numSensors
        thisConfig = configs(:,iSensor);
        for k = 1:numsteps
            if thisConfig(k).IsValidTime
                pos = trackToSensor(allposTrack(:,k),thisConfig(k));
                % Check if predicted track position is in sensor field of
                % view
                [az,el] = cart2sph(pos(1),pos(2),pos(3));
                az = az*180/pi;
                el = el*180/pi;
                inFov = abs(az)<thisConfig(k).FieldOfView(1)/2 && abs(el) < thisConfig(k).FieldO
                if inFov
                    isDetectable(iTrk) = inFov;
                    k = numsteps; %#ok<FXSET>
                    iSensor = numSensors; %#ok<FXSET>
                end
            end
        end
    end

end

trackIDs = [tracks.TrackID]';
trackIDs = trackIDs(isDetectable);
end
```

5-211

trackToSensor This function returns the track's position in the sensor's coordinate frame. The track structure is returned by the `trackerGNN` object and the config structure defining the sensor's orientation relative to the track's coordinate frame is returned by the `monostaticRadarSensor` object.

```
function pos = trackToSensor(pos,config)

frames = config.MeasurementParameters;
for m = numel(frames):-1:1
    rotmat = frames(m).Orientation;
    if ~frames(m).IsParentToChild
        rotmat = rotmat';
    end
    offset = frames(m).OriginPosition;
    pos = bsxfun(@minus,pos,offset);
    pos = rotmat*pos;
end
end
```

longRangeCorrection This function limits the measurement noise accuracy reported by the radar to not exceed a maximum condition number. The condition number is defined as the ratio of the eigenvalues of the measurement noise to the largest eigenvalue.

When targets are detected at long ranges from a radar, the surface curvature of the uncertainty of the measurement is no longer well approximated by an ellipsoid but takes on that of a concave ellipsoid. The measurement noise must be increased along the range dimension to account for the concavity, producing a planar ellipse which encompasses the concave ellipsoid. There are several techniques in the literature to address this. Here, the maximum condition number of the measurement noise is limited by increasing the smallest eigenvalues to satisfy the maximum condition number constraint. This has the effect of increasing the uncertainty along the range dimension, producing an ellipse which better encloses the concave uncertainty.

```
function dets = longRangeCorrection(dets,maxCond)
for m = 1:numel(dets)
    R = dets{m}.MeasurementNoise;
    [Q,D] = eig(R);
    Q = real(Q);
    d = real(diag(D));
    dMax = max(d);
    condNums = dMax./d;
    iFix = condNums>maxCond;
    d(iFix) = dMax/maxCond;
    R = Q*diag(d)*Q';
    dets{m}.MeasurementNoise = R;
end
end
```

Simulating Passive Radar Sensors and Radar Interferences

This example shows you how to model and simulate the output of active and passive radar sensors. In this example, you observe how radio frequency (RF) interference impacts the detection performance of a radar. In addition, you use passive radar sensors to estimate the location and type of the RF interference.

Introduction

Assessing tracking performance for radars requires modeling a radio frequency (RF) scenario. This includes:

- Generating RF emissions
- Propagating the emissions and reflecting these emissions from platforms
- Receiving the emissions, calculating interference losses, and generating detections

Create scenario

This example shows you how to perform each of the steps above using a scenario consisting of three platforms:

1 Airborne platform flying north at 500 km/h from the bottom of the scenario
2 Airborne platform flying south at 600 km/h from the top of the scenario
3 Airborne platform flying east at 700 km/h in the middle of the scenario

```
% Create a tracking scenario to manage the movement of the platforms.
scene = trackingScenario;

% Set the duration of the scenario to 10 seconds.
scene.StopTime = 10; % s

% Platform 1: Airborne and northbound at 500 km/h
spd = 500*1e3/3600; % m/s
wp1 = [0 0 -6000];
wp2 = [spd*scene.StopTime 0 -6000];
toa = [0; scene.StopTime];
platform(scene, 'Trajectory', waypointTrajectory('Waypoints', [wp1; wp2], 'TimeOfArrival', toa))

% Platform 2: Airborne and southbound at 600 km/h
spd = 600*1e3/3600; % m/s
wp1 = [30e3+spd*scene.StopTime 0 -6000];
wp2 = [30e3 0 -6000];
toa = [0; scene.StopTime];
platform(scene, 'Trajectory', waypointTrajectory('Waypoints', [wp1; wp2], 'TimeOfArrival', toa))

% Platform 3: Airborne and eastbound at 700 km/h
spd = 700*1e3/3600; % m/s
wp1 = [10e3 1e3 -6000];
wp2 = [10e3 1e3+spd*scene.StopTime -6000];
toa = [0; scene.StopTime];
platform(scene, 'Trajectory', waypointTrajectory('Waypoints', [wp1; wp2], 'TimeOfArrival', toa))
```

Use `theaterPlot` to create a display showing the platforms in the scenario and their trajectories.

```
ax = axes;
theaterDisplay = theaterPlot('Parent',ax,'AxesUnit',["km" "km" "km"], 'XLim',[-10000 40000] , 'Y
```

```
view([90 -90]) % swap X and Y axis
patch('XData',[-10000 -10000 40000 40000],'YData',[-20000 20000 20000 -20000],'EdgeColor','none'

platPlotter = platformPlotter(theaterDisplay,'DisplayName','Platforms','MarkerFaceColor','k');
plotPlatform(platPlotter,vertcat(scene.platformPoses.Position));

trajPlotter = trajectoryPlotter(theaterDisplay,'DisplayName','Trajectories','LineStyle','-');
allTrajectories = cellfun(@(x) x.Trajectory.lookupPose(linspace(0,scene.StopTime,10)), scene.Pla
plotTrajectory(trajPlotter,allTrajectories);
```
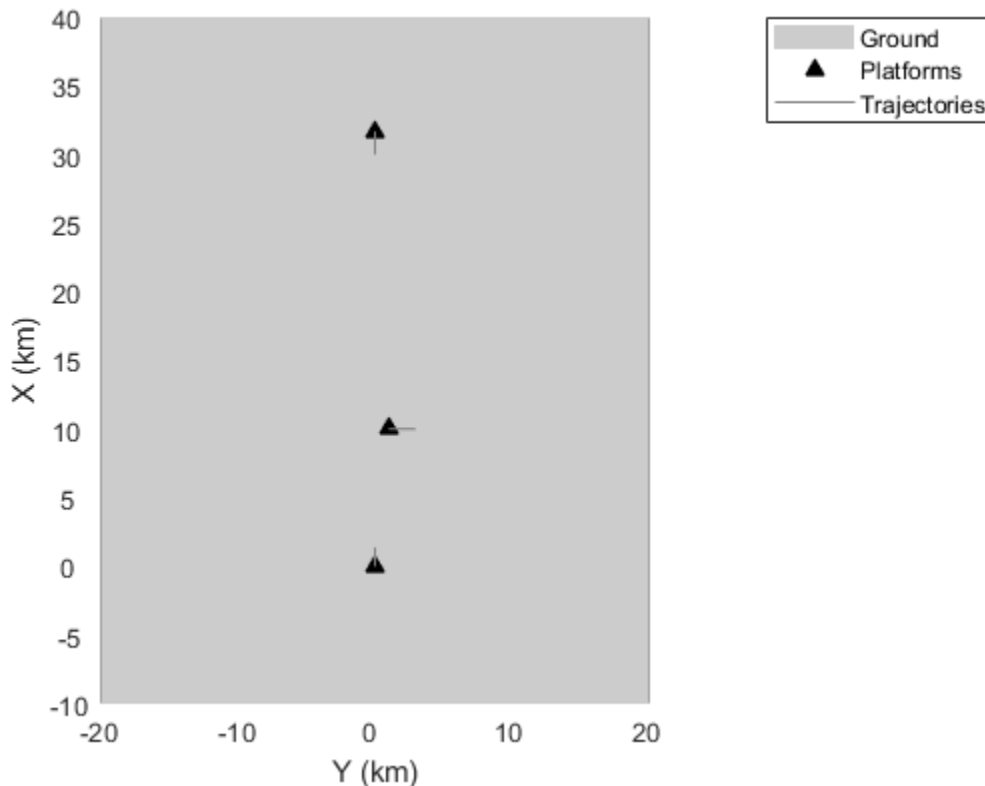

Radar Detection in the Presence of Interfering Emissions

Often, a radar operates in an environment where other undesirable RF emissions interfere with the waveforms emitted by the radar. When this occurs, the radar will experience a degradation in detection performance in the direction of the interfering signals. Attach an RF emitter to the platform at the bottom of the scenario (the first platform) and a radar to the platform at the top of the scenario (the second platform).

Create an RF emitter

Model an RF emission using a `radarEmitter`. The emitter is in a forward-looking configuration with an azimuthal field of view of 20 degrees to include the 2 other platforms in the scenario. The effective isotropic radiated power (EIRP) sets the strength of the interfering signal. The waveform type is a user-defined value used to enumerate the various waveform types that are present in the scenario. For this scenario, use the value 0 to indicate a noise waveform type.

```
% Create the interference emitter.
rfEmitter = radarEmitter(1, 'No scanning', ...
    'FieldOfView', [20 5], ...        % [az el] deg
    'EIRP', 200, ...                  % dBi
    'CenterFrequency', 300e6, ...     % Hz
    'Bandwidth', 30e6, ...            % Hz
    'WaveformType', 0)                % Use 0 for noise-like emissions

rfEmitter =
  radarEmitter with properties:

        EmitterIndex: 1
          UpdateRate: 1
            ScanMode: 'No scanning'

    MountingLocation: [0 0 0]
      MountingAngles: [0 0 0]

         FieldOfView: [2x1 double]
     MechanicalAngle: 0
           LookAngle: 0
        HasElevation: 0

                EIRP: 200
     CenterFrequency: 300000000
           Bandwidth: 30000000
        WaveformType: 0
       ProcessingGain: 0
```

Attach the emitter to the first platform.

```
platEmit = scene.Platforms{1};
platEmit.Emitters = rfEmitter

platEmit =
  Platform with properties:

        PlatformID: 1
           ClassID: 0
          Position: [0 0 -6000]
       Orientation: [0 0 0]
        Dimensions: [1x1 struct]
              Mesh: [1x1 extendedObjectMesh]
        Trajectory: [1x1 waypointTrajectory]
     PoseEstimator: [1x1 insSensor]
           Emitters: {[1x1 radarEmitter]}
            Sensors: {}
         Signatures: {[1x1 rcsSignature]  [1x1 irSignature]  [1x1 tsSignature]}
```

Create a monostatic radar

Equip the second platform with a monostatic radar. Use `monostaticRadarSensor` to model this type of radar. Configure the radar to scan the azimuth sector in front of its platform which includes the other platforms.

```
% Create a monostatic radar.
radar = monostaticRadarSensor(2, 'Sector', ...
```

```
    'UpdateRate', 12.5, ...          % Hz
    'FieldOfView', [2 10]);          % [az el] deg

% Mount the radar so that it scans the sector in front of the second
% platform which includes the interfering signal and the third platform.
radar.MountingAngles = [0 0 0];      % [Z Y X] deg

% Enable the radar's INS input so that it can use the pose estimated by
% the platform's pose estimator to generate detections.
radar.HasINS = true;

% Enable the radar's interference input so that the interference signal
% created by the emitter above can be passed to the radar.
radar.HasInterference = true;

% Set coordinate system for detections to scenario
radar.DetectionCoordinates = 'scenario'

radar =
  monostaticRadarSensor with properties:

                   SensorIndex: 2
                    UpdateRate: 12.5000
                      ScanMode: 'Mechanical'

              MountingLocation: [0 0 0]
                MountingAngles: [0 0 0]

            MaxUnambiguousRange: 100000
      MaxUnambiguousRadialSpeed: 200
                    FieldOfView: [2x1 double]
           MaxMechanicalScanRate: 75
            MechanicalScanLimits: [-45 45]
                MechanicalAngle: 0
                      LookAngle: 0

          DetectionProbability: 0.9000
                 FalseAlarmRate: 1.0000e-06

    Show all properties


% Attach the radar to the second platform.
platRadar = scene.Platforms{2};
platRadar.Sensors = radar;
```

Update the display to show the platforms, the radar, and the emitter in the scenario.

```
emitterColor =  [0.9290 0.6940 0.1250];
radarColor = [0 0.4470 0.7410];
platEmitPlotter = platformPlotter(theaterDisplay,'DisplayName', 'RF emitter','Marker','d','Marker
platRadarPlotter = platformPlotter(theaterDisplay,'DisplayName','Monostatic radar','Marker','d',
platPlotter.DisplayName = 'Targets';
clearData(platPlotter);
covPlotter = coveragePlotter(theaterDisplay,'Alpha',[0.2 0]);
detPlotter = detectionPlotter(theaterDisplay,'DisplayName','Radar detections','MarkerFaceColor',
title('Radar detection with an interfering emitter');
```

```
plotPlatform(platRadarPlotter, platRadar.pose.Position);
plotPlatform(platEmitPlotter, platEmit.pose.Position);
plotPlatform(platPlotter, scene.Platforms{3}.pose.Position);
plotCoverage(covPlotter, coverageConfig(scene), [-1 2], {emitterColor, radarColor});
```
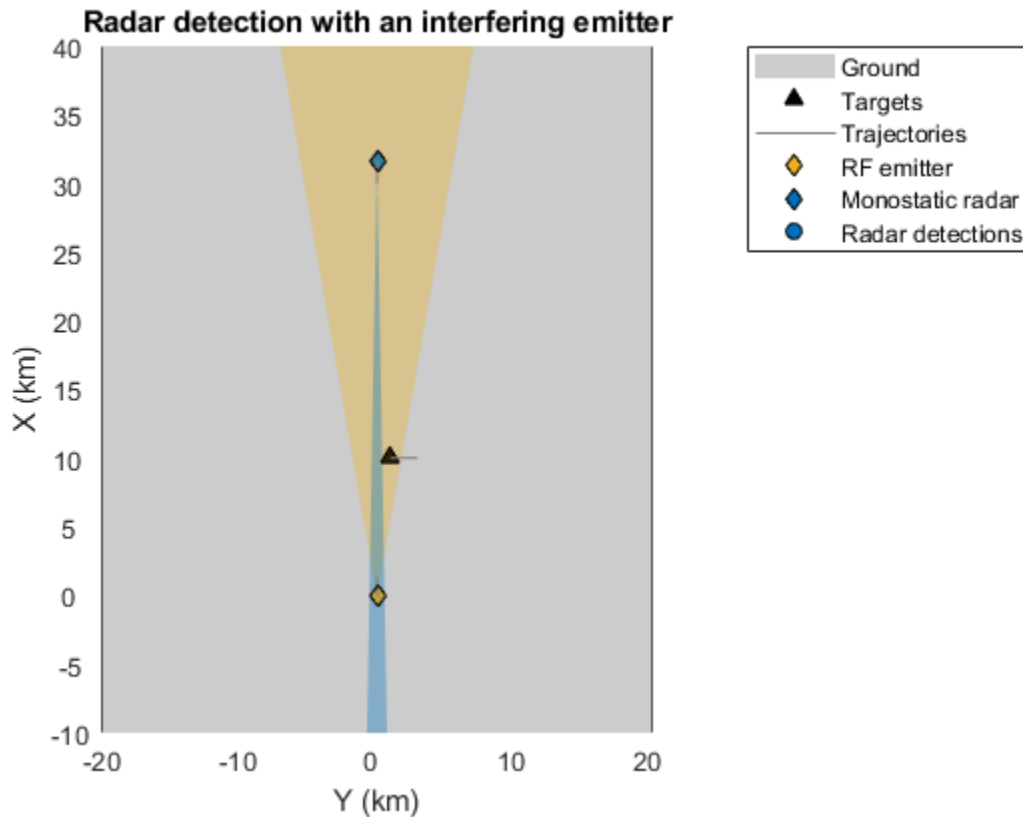

In the preceding figure, the platform carrying the forward-looking radar is shown as a blue diamond, with the radar's current field of view as a blue region originating from the platform. At the bottom of the figure, the platform carrying the interfering RF emission is shown as a yellow diamond, with the emitter's current field of view shown as a corresponding yellow region. Platforms without any emitters or sensors attached to them are referred to as *targets* and are displayed as black triangles.

Simulate monostatic detections

In multi-rate scenarios, you can either find an update rate that is a common divider of all the sensors and emitters rates defined in the scenario or you can use the continuous update which will automatically advance the scenario to the next valid update time when calling `advance`.

```
scene.UpdateRate = 0

scene =
  trackingScenario with properties:

    IsEarthCentered: 0
         UpdateRate: 0
     SimulationTime: 0
           StopTime: 10
```

```
        IsRunning: 1
        Platforms: {1x3 cell}
```

For each step in the following loop use:

1   `advance` to move all of the platforms according to their trajectories.

2   `emit` to update the emissions transmit direction from `platEmit`.

3   `propagate` to propagate the emissions directly to each platform in the scenario that lies within the emitter's field of view. Each platform that receives a direct path emission generates a single-bounce reflection that is also propagated to every other platform as a reflected emission.

4   `detect` to generate detections from the emissions received at `platRadar`.

This propagation of the emitter's emissions from the emitter to the radar sensor are illustrated in the following image.
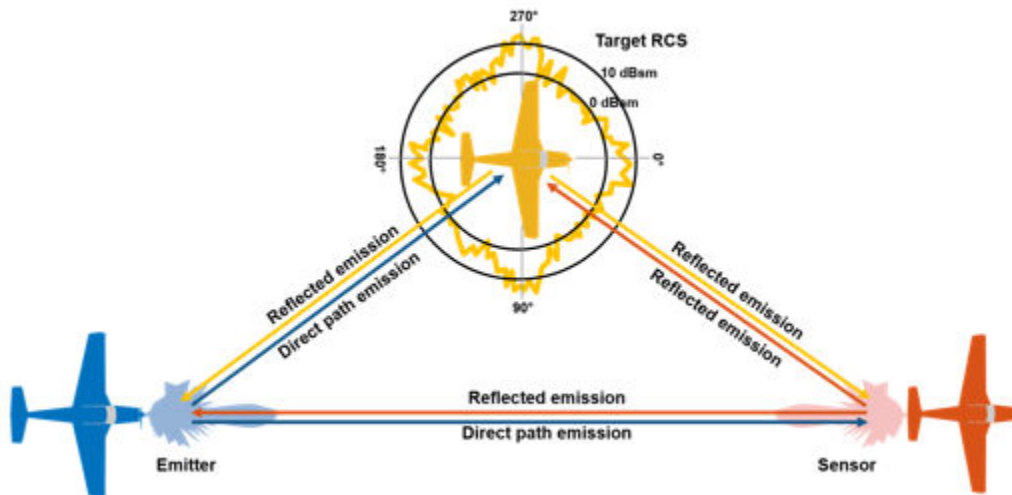

```
% Set the random seed for repeatable results.
rng(2018);

plotDets = {};
while advance(scene)

    % Emit the RF signal.
    txEmiss = emit(scene);

    % Reflect the emitted signal off of the platforms in the scenario.
    reflEmiss = propagate(scene, txEmiss);

    % Generate detections from the monostatic radar sensor.
    [dets, config] = detect(scene, reflEmiss);

    % Reset detections every time the radar completes a sector scan.
    if config.IsScanDone
        % Reset
        plotDets = dets;
```

```
    else
        % Buffer
        plotDets = [plotDets;dets]; %#ok<AGROW>
    end

    % Update display with current platform positions, beam positions and detections.
    plotPlatform(platRadarPlotter, platRadar.pose.Position);
    plotPlatform(platEmitPlotter, platEmit.pose.Position);
    plotPlatform(platPlotter, scene.Platforms{3}.pose.Position);
    plotCoverage(covPlotter, coverageConfig(scene), [-1 2], {emitterColor, radarColor});
    if ~isempty(plotDets)
        allDets = [plotDets{:}];
        % extract column vector of measurement positions
        meas = [allDets.Measurement]';
        plotDetection(detPlotter,meas);
    end
end
```

Radar detection with an interfering emitter
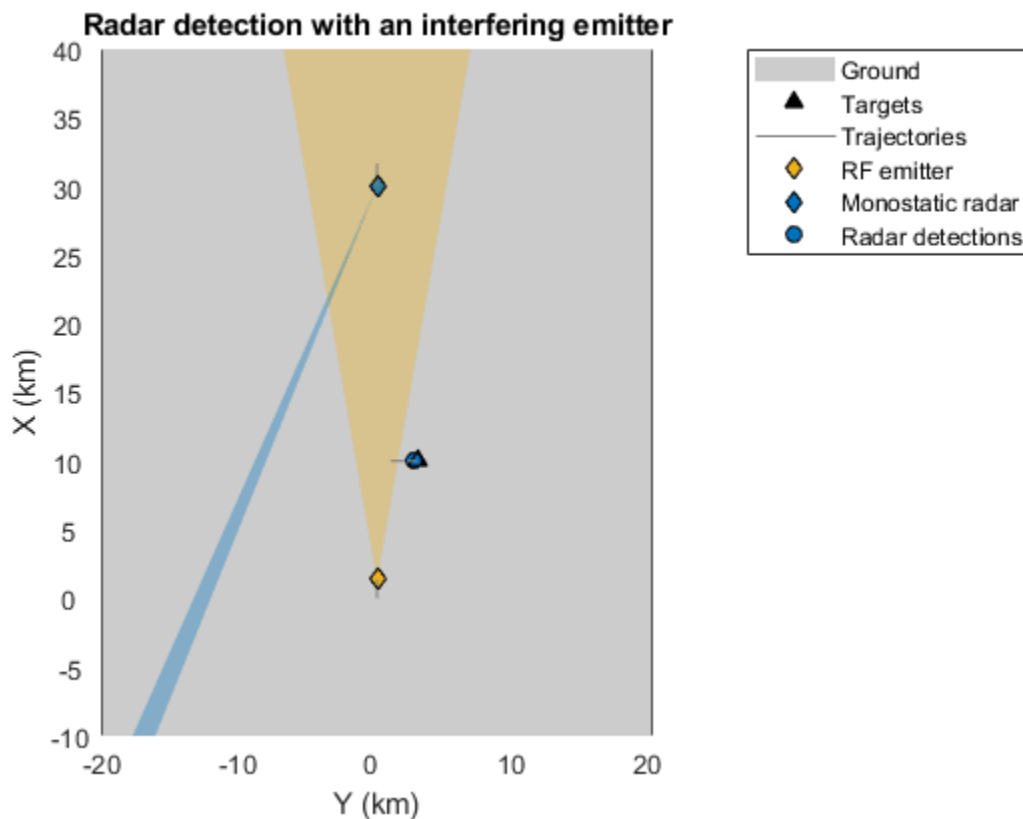
As shown in the preceding figure, the radar (shown in blue) is only able to detect the target in the middle of the scenario. The detections are shown as blue, filled circles, and are made whenever the radar's field of view (e.g. beam) sweeps across the target. However, when the radar's beam sweeps across the emitting platform (shown in yellow), no detections are generated, since the interference generated by this platform prevents detection by the radar.

Passive Detection of RF Emissions

In the preceding section, the radar was unable to detect the location of the emitting platform because the emissions from that platform masked the radar's own emissions. However, such strong emissions can be detected and identified by passive sensors which listen for RF emissions in the scenario. These sensors are often referred to as electronic support measures (ESM). These sensors typically listen across a broad range of frequencies and attempt to identify unique emitters, the direction of arrival of the emissions from those emitters, and whenever possible, the type of waveform used by the emitter.

Create an ESM sensor

Reuse the scenario from the previous section, but replace the monostatic radar on the first platform with an ESM sensor. Use `radarSensor` to model the ESM sensor and ensure that the sensor is configured so that its center frequency and bandwidth includes the RF spectrum of the emitter. Otherwise, it will be unable able to detect the emitter.

```
restart(scene);
esm = radarSensor(1, 'No scanning', ...
    'DetectionMode', 'ESM', ...
    'UpdateRate', 12.5, ...          % Hz
    'MountingAngles', [0 0 0], ...   % [Z Y X] deg
    'FieldOfView', [30 10], ...      % [az el] deg
    'CenterFrequency', 300e6, ...    % Hz
    'Bandwidth', 30e6, ...           % Hz
    'WaveformTypes', 0, ...          % Detect the interference waveform type
    'HasINS', true)

esm =
  radarSensor with properties:

          SensorIndex: 1
           UpdateRate: 12.5000
             ScanMode: 'No scanning'
        DetectionMode: 'ESM'

      MountingLocation: [0 0 0]
        MountingAngles: [0 0 0]

           FieldOfView: [2x1 double]
        MechanicalAngle: 0
             LookAngle: 0

           Sensitivity: -50
        FalseAlarmRate: 1.0000e-06

    Show all properties
```

Replace the radar on the second platform with the ESM sensor.

```
platESM = scene.Platforms{2};
platESM.Sensors = esm;
```

Update the visualization accordingly

```
platRadarPlotter.DisplayName = "ESM sensor";
esmColor = [0.4940 0.1840 0.5560];
```

```
platRadarPlotter.MarkerFaceColor = esmColor;
% use a helper to add an angle only detection plotter
delete(detPlotter);
esmDetPlotter = helperAngleOnlyDetectionPlotter(theaterDisplay,'DisplayName','ESM detections','Co
clearData(covPlotter);

plotCoverage(covPlotter, coverageConfig(scene), [-1 1], {emitterColor, esmColor});
title('Passive detection of RF emissions');
```
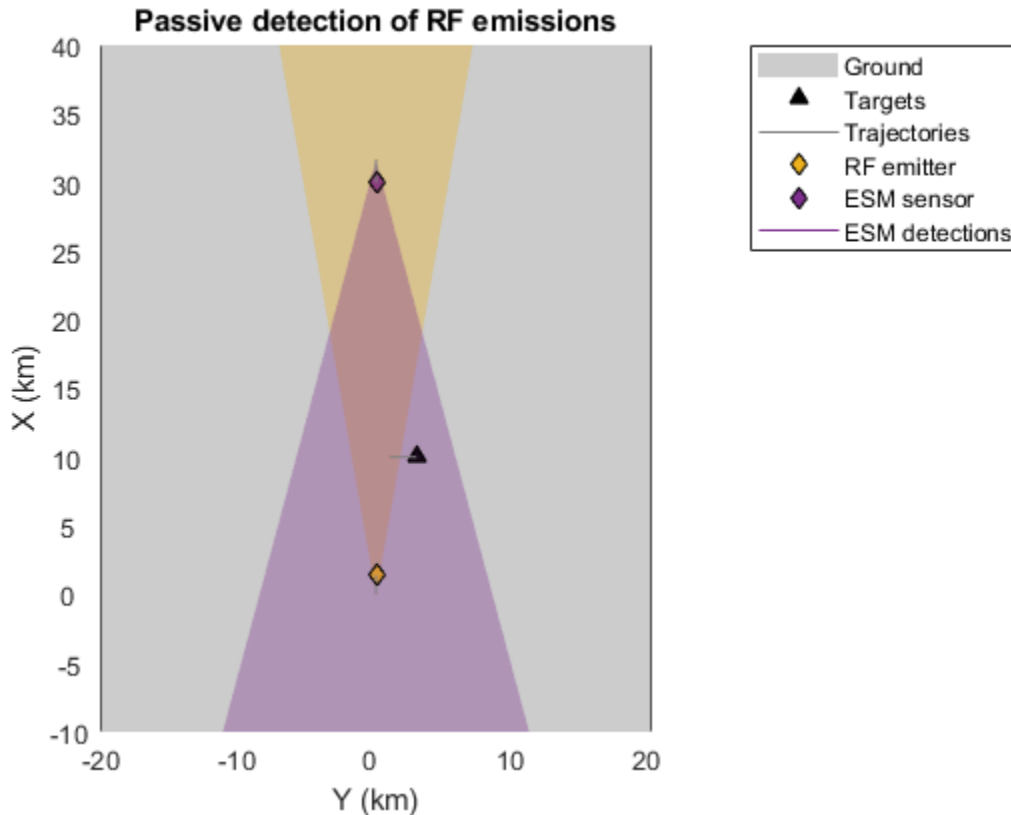

In the preceding figure, the radar is replaced by an ESM sensor mounted on the second platform. The ESM sensor's field of view is shown in magenta and includes both the emitting and target platforms.

Simulate ESM detections

Now simulate detections using the ESM sensor instead of the radar. You will notice that because the ESM sensor is a passive sensor, it is unable to localize the emitting platform, but indicates the direction of arrival the platform's emissions. These angle-only detections are shown as rays originating from the ESM sensor towards the emitting platform.

```
% Set the random seed for repeatable results.
rng(2018);

plotDets = {};
snap = [];
while advance(scene)

    % Emit the RF signal.
```

```
        txEmiss = emit(scene);

        % Reflect the emitted signal off of the platforms in the scenario.
        reflEmiss = propagate(scene, txEmiss);

        % Generate detections from the monostatic radar sensor.
        [dets, config] = detect(scene, reflEmiss);

        % Reset detections every time the radar completes a sector scan.
        if config.IsScanDone
            % Reset
            plotDets = dets;
        else
            % Buffer
            plotDets = [plotDets;dets]; %#ok<AGROW>
        end

        % Update display with current platform positions, beam positions and detections.
        plotPlatform(platRadarPlotter, platRadar.pose.Position);
        plotPlatform(platEmitPlotter, platEmit.pose.Position);
        plotPlatform(platPlotter, scene.Platforms{3}.pose.Position);
        plotCoverage(covPlotter, coverageConfig(scene), [-1 1], {emitterColor, esmColor});
        plotDetection(esmDetPlotter,plotDets);

        % Record the reflected detection at t = 2 sec
        snap = getSnap(ax, scene.SimulationTime, 2, snap);
        drawnow
    end
    title('RF emitter detected by an ESM sensor');
```

RF emitter detected by an ESM sensor
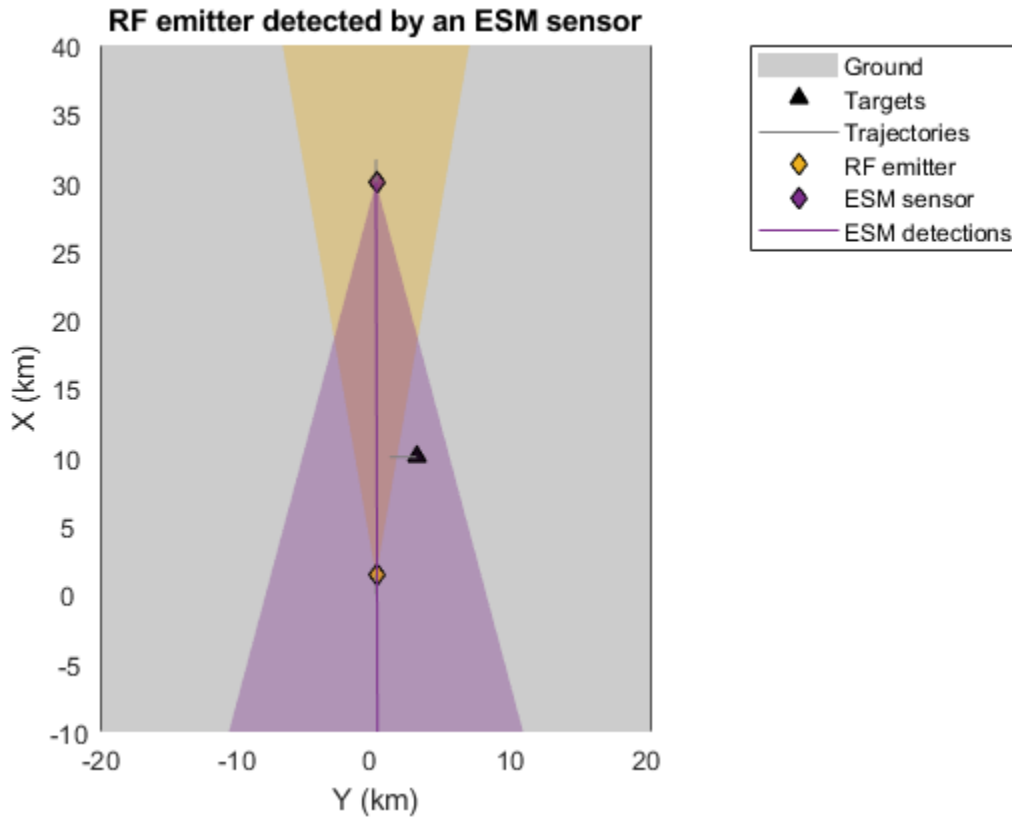
The RF emissions are detected and their direction of arrival is estimated by the ESM sensor. This is shown by the magenta line originating from the sensor and closely passing by the emitter. There is some noise in the estimated angle, which is why the line does not pass directly through the emitter's location.

The ESM sensor classifies the waveform types in its reported detections. For this emitter, it reports the noise waveform type used by the emitter: 0.

```
dets{1}.ObjectAttributes{1}
```

```
ans = struct with fields:
    TargetIndex: 1
   EmitterIndex: 1
   WaveformType: 0
            SNR: 184.8224
```

Notice that the signal-to-noise ratio (SNR) of the emitted signal detected by the sensor is very large, 185 dB. Because the RF emitter has high power, reflections of the emitted waveform off of the target will also be detected by the ESM sensor. This is seen at 2 seconds in the simulation when the target lies within the emitter's field of view.

```
figure; imshow(snap.cdata);
title('Emitter and target detected by an ESM sensor');
```

Emitter and target detected by an ESM sensor
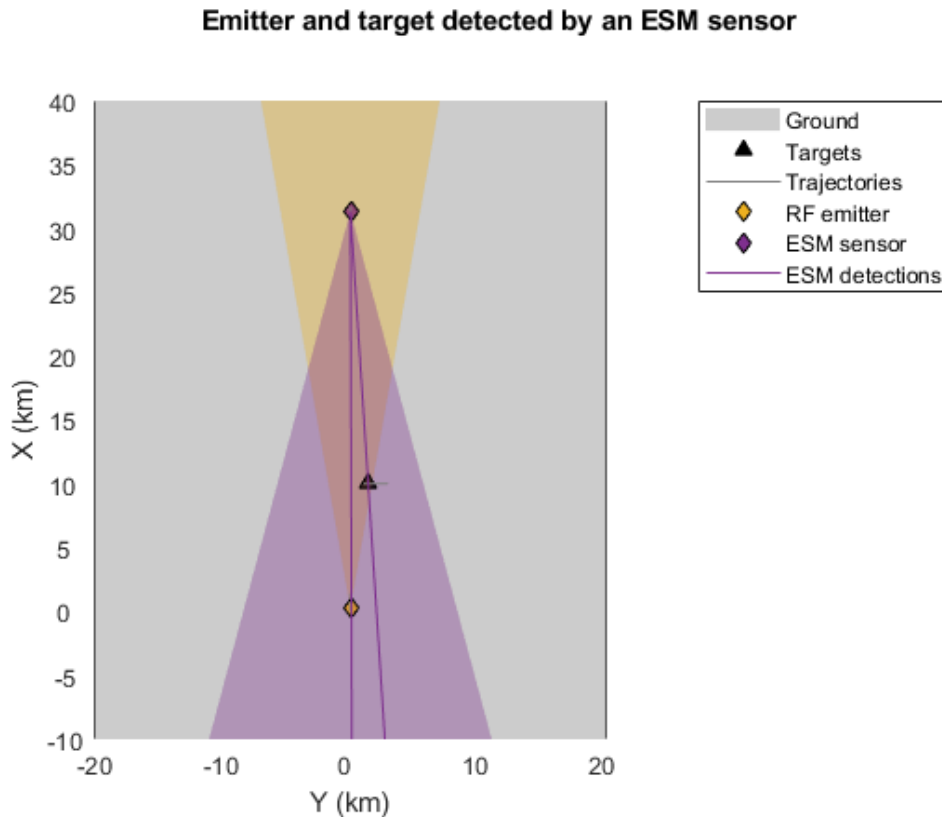
In the preceding figure, emissions are detected from both the emitter and the target, as the target receives energy from the emitter and re-emits that waveform back into the scenario, causing it to be detected by the ESM sensor as well.

Passive Detection of Monostatic Radars

Monostatic radars also emit waveforms into the scenario. Sometimes it is desirable to be able to detect their emissions. To do this, you must model both the emitting and sensing portions of the radar separately. The emitter generates the waveforms which become part of the scenario's RF emissions. These waveforms can then be detected by other sensors, such as the ESM sensor presented in the preceding section.

Reuse the same scenario from the preceding sections. For this scenario, attach a monostatic radar to the platform at the top of the scenario (the second platform), and attach an ESM sensor to the platform at the bottom of the scenario (the first platform). The middle platform remains a target with no emitters or sensors attached to it.

```
restart(scene);
```

Create a monostatic radar by modeling both the emitting and sensing portions of the sensor. Use `radarEmitter` to model the monostatic radar emitter. For this scenario, use 1 to indicate the waveform type used by this radar, where perhaps 1 represents an LFM chirp with some predefined

parameters. The waveform type is an enumeration defined by the user to represent the different kinds of waveforms simulated in the scenario. This enables emitters and sensors to know how to process these waveforms to generate detections. For example, if an emitter has a waveform type of 1 and a sensor includes this type in its waveform list, then the sensor will know how to process the emitter's emissions (e.g. using a matched filter) and will detect the emission and realize the processing gain associated with that waveform.

```
% Create the emitter for the monostatic radar.
radarTx = radarEmitter(2, 'Sector', ...
    'UpdateRate', 12.5, ...          % Hz
    'MountingAngles', [0 0 0], ...   % [Z Y X] deg
    'FieldOfView', [2 10], ...       % [az el] deg
    'CenterFrequency', 300e6, ...    % Hz
    'Bandwidth', 3e6, ...            % Hz
    'ProcessingGain', 50, ...        % dB
    'WaveformType', 1)               % Use 1 to indicate this radar's waveform

radarTx =
  radarEmitter with properties:

            EmitterIndex: 2
              UpdateRate: 12.5000
                ScanMode: 'Mechanical'

        MountingLocation: [0 0 0]
          MountingAngles: [0 0 0]

             FieldOfView: [2x1 double]
     MaxMechanicalScanRate: 75
      MechanicalScanLimits: [-45 45]
         MechanicalAngle: 0
               LookAngle: 0
            HasElevation: 0

                    EIRP: 100
         CenterFrequency: 300000000
               Bandwidth: 3000000
            WaveformType: 1
          ProcessingGain: 50
```

Use `radarSensor` to model the sensing portion of the radar which receives the RF emissions in the scenario, identifies the emissions that corresponds to the monostatic emitter's waveform type, and generates detections from these received emissions. Emissions which do not match the emitter's waveform type are treated as interference.

When using `radarSensor` to model the sensing portion of a monostatic radar, you must set the `DetectionMode` property of the sensor to `Monostatic`. This tells the sensor to use the emitter's configuration when processing the received RF emissions.

```
radarRx = radarSensor(2, ...
    'DetectionMode', 'Monostatic', ...
    'EmitterIndex', radarTx.EmitterIndex, ...
    'HasINS', true,...
    'DetectionCoordinates','Scenario')

radarRx =
  radarSensor with properties:
```

```
               SensorIndex: 2
             DetectionMode: 'Monostatic'
              EmitterIndex: 2

       MaxUnambiguousRange: 100000
 MaxUnambiguousRadialSpeed: 200
                 LookAngle: 0

               Sensitivity: -50
            FalseAlarmRate: 1.0000e-06

  Show all properties
```

```matlab
% Attach to the radar emitter and sensor to the second platform.
platRadar = scene.Platforms{2};
platRadar.Emitters = radarTx;
platRadar.Sensors = radarRx;
```

Reuse the ESM sensor from the preceding section, but set the list of known waveform types for the ESM sensor to include the waveform emitted by the radar. If the radar's waveform type is not known to the ESM sensor, it will not be detected.

```matlab
% Add the radar's waveform to the list of known waveform types for the ESM sensor.
esm.WaveformTypes = [0 1];
```

```matlab
% Attach the ESM sensor to the first platform.
platESM = scene.Platforms{1};
platESM.Emitters = {}; % Remove the emitter.
platESM.Sensors = esm;
```

Update the display to show both monostatic detections and ESM detections.

```matlab
detPlotter = detectionPlotter(theaterDisplay,'DisplayName','Radar detections','MarkerFaceColor',
platRadarPlotter.DisplayName = 'Monostatic radar';
platRadarPlotter.MarkerFaceColor = radarColor;
platEmitPlotter.DisplayName = 'ESM sensor';
platEmitPlotter.MarkerFaceColor = esmColor;
clearData(esmDetPlotter);
clearData(covPlotter);
covcon = coverageConfig(scene);
plotCoverage(covPlotter, covcon([1 3]) , [1 -2], {esmColor, radarColor});
title(ax,'Passive detection of a monostatic radar');
```

Passive detection of a monostatic radar
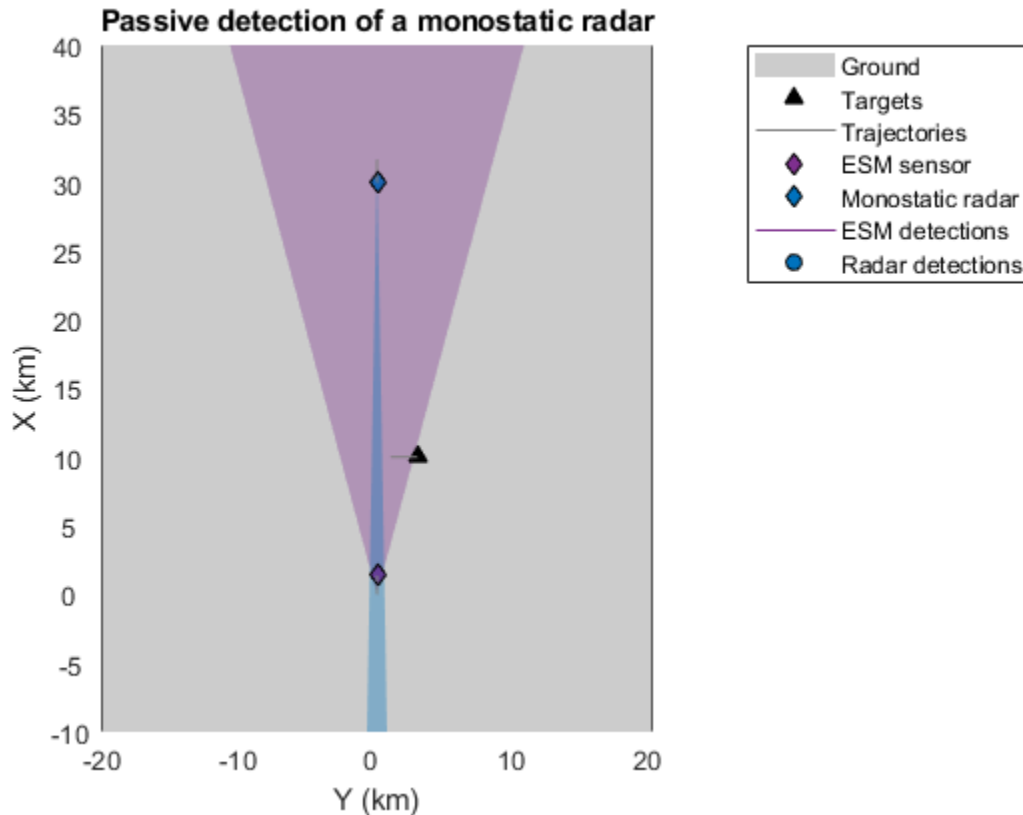
In the preceding figure, the radar is scanning an azimuth sector in front of its platform which includes both the target platform as well as the platform carrying the ESM sensor. The radar will generate detections for both of these platforms when its field of view (shown in blue) sweeps over their location. However, when the radar's beam passes over the location of the ESM sensor, the ESM sensor will detect the radar and will indicate its estimated position by drawing a line originating from the sensor's location.

```
% Set the random seed for repeatable results.
rng(2018);

platforms = scene.Platforms;
numPlat = numel(platforms);


plotDets = {};
snap = [];
while advance(scene)

    % Emit the RF signal.
    [txEmiss, txConfigs] = emit(scene);

    % Reflect the emitted signal off of the platforms in the scenario.
    reflEmiss = propagate(scene, txEmiss);

    % Generate detections from the monostatic radar sensor.
    [dets, config] = detect(scene, reflEmiss, txConfigs);
```

```
% Reset detections every time the radar completes a sector scan.
if txConfigs(end).IsScanDone
    % Reset
    plotDets = dets;
else
    % Buffer
    plotDets = [plotDets;dets];%#ok<AGROW>
end

% Update display with current platform positions, beam positions and detections.
plotPlatform(platRadarPlotter, platRadar.pose.Position);
plotPlatform(platEmitPlotter, platEmit.pose.Position);
plotPlatform(platPlotter, scene.Platforms{3}.pose.Position);
covcon = coverageConfig(scene);
plotCoverage(covPlotter, covcon([1 3]) , [1 -2], {esmColor, radarColor});
plotDetection(esmDetPlotter,plotDets);
plotMonostaticDetection(detPlotter,plotDets);

% Record the reflected detection at t = 5.6 sec
snap = getSnap(ax, scene.SimulationTime, 5.6, snap);
drawnow
end
```
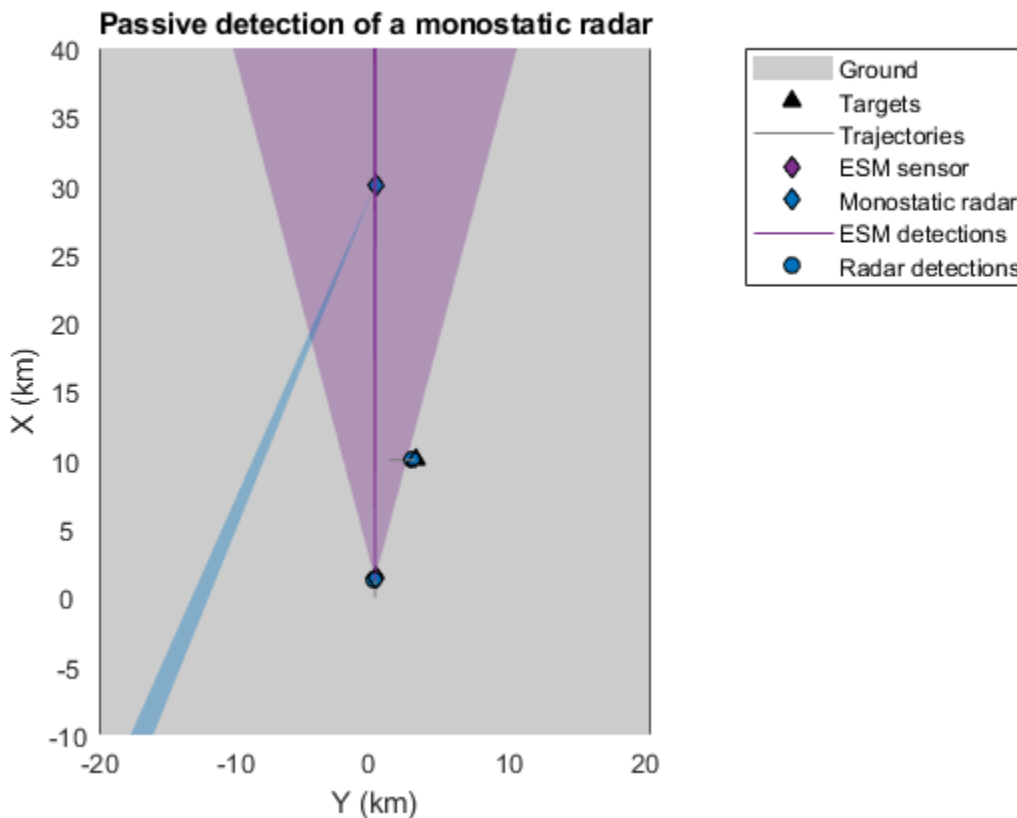

Detections from the monostatic radar modeled using `radarEmitter` and `radarSensor` are shown as filled, blue circles near the target and the platform equipped with the ESM sensor. The ESM sensor is also able to detect the radar, as is indicated by the angle-only detection shown as a line originating from the ESM sensor and passing near the radar platform.

```
figure; imshow(snap.cdata);
title('Radar and target detected by an ESM sensor');
```

Radar and target detected by an ESM sensor
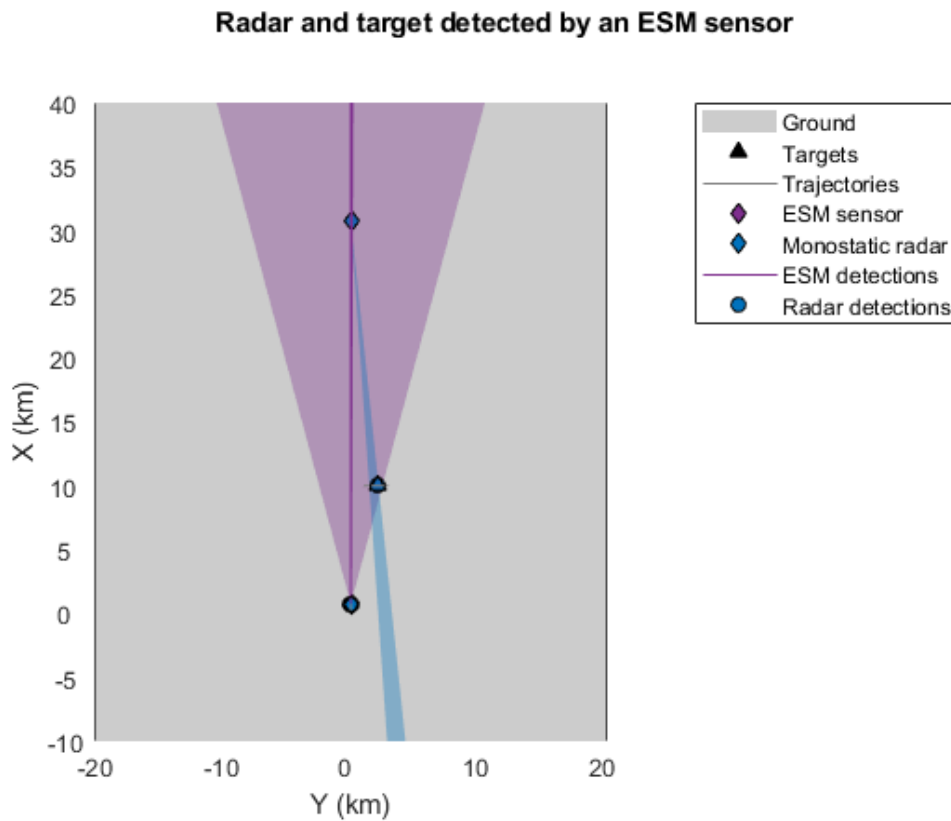
Because of the high power (EIRP) of the radar's emissions, the emitted energy is reflected off of the target towards the ESM platform as well. This results in the ESM sensor detecting the target platform at times when the radar emitter's field of view sweeps past the target platform when the target is still inside of the ESM sensor's field of view.

Summary

In this example, you learned how to simulate the impact of RF interference on a monostatic radar. You also learned how to simulate passive sensors which can detect these interference sources as well as the radar sensors in the scenario.

Supporting Functions

`getSnap` Records a snapshot of an axis at a given snap time.

```
function snap = getSnap(hAx, curTime, snapTime, prevSnap)
if ~isempty(prevSnap)
    snap = prevSnap;
elseif curTime >= snapTime && curTime < snapTime + 0.05
    hAx.Title.Visible = 'off';
```

```
    snap = getframe(hAx.Parent);
    hAx.Title.Visible = 'on';
else
    snap = [];
end
end
```

plotMonostaticDetection Parses detections to plot only monostatic detections with a detectionPlotter.

```
function plotMonostaticDetection(plotter, dets)
if ~isempty(dets)
    % Pass only monostatic detections to the detectionPlotter
    radarDetId = cellfun(@(x) x.SensorIndex == 2, dets);
    if any(radarDetId)
        % extract measurement positions for the monostatic radar
        radarDets = [dets{radarDetId}];
        meas = [radarDets.Measurement]';
        plotDetection(plotter,meas);
    end
end
end
```

Multiplatform Radar Detection Generation

This example shows how to generate radar detections from a multiplatform radar network. The network includes three long-range platforms: two airborne and one ground-based. Such synthetic data can be used to test the performance of tracking architectures for different target types and maneuvers.

The radar platforms and targets are modeled in the scenario as `platforms`. Simulation of the motion of the platforms in the scenario is managed by `trackingScenario`.

```
% Create a tracking scenario to manage the movement of the platforms.
scene = trackingScenario;                % Create tracking scenario
scene.UpdateRate = 0;                    % Use continuous update rate to handle sensors with dif
sceneDuration = 60;                      % Duration of scenario in seconds
scene.StopTime = sceneDuration;
```

Airborne Platform with Rotating Radar Array

Add an airborne platform to the scenario traveling north at 650 km/hr at a cruising altitude of 10 km. Generate the platform trajectory from waypoints using `waypointTrajectory`.

```
ht = 10e3;                                % Altitude in meters
spd = 650*1e3/3600;                       % Speed in m/s
start = [-spd*sceneDuration/2 5e3 -ht];
stop  = [spd*sceneDuration/2 5e3 -ht];
traj  = waypointTrajectory('Waypoints',[start;stop],'TimeOfArrival',[0; sceneDuration]);

% Create the airborne platform with its trajectory.
plat1 = platform(scene,'Trajectory',traj);
```

Add a planar array radar to the platform. Mount the radar in a radome 5 meters above the platform. Model the radar as a mechanically rotating phased array. The radar electronically stacks beams in elevation along the array's boresight. The specifications for the modeled radar are tabulated below:

- Sensitivity: 0 dBsm @ 375 km
- Mechanical Scan: Azimuth only
- Mechanical Scan Limits: 0 to 360 deg
- Electronic Scan: Elevation only
- Electronic Scan Limits: -2 to 45 deg
- Field of View: 1 deg azimuth, 47 deg elevation
- Measurements: Azimuth, elevation, range
- Azimuth Resolution: 1 deg
- Elevation Resolution: 5 deg
- Range Resolution: 30 m

Model the mechanically rotating radar using `monostaticRadarSensor`.

```
sensorIndex = 1; % Identifies originating sensor of each detection
radar = monostaticRadarSensor(sensorIndex, 'Rotator', ...
    'MountingLocation', [0 0 -5], ...    % m
    'UpdateRate', 12.5, ...              % Hz
    'ReferenceRCS', 0, ...               % dBsm
    'ReferenceRange', 375e3, ...         % m
```

```
    'ScanMode', 'Mechanical and electronic', ...
    'MechanicalScanLimits', [0 360;0 0], ... % deg
    'ElectronicScanLimits', [0 0;-2 45], ... % deg
    'FieldOfView',[1;47.1], ...            % deg
    'HasElevation', true, ...
    'AzimuthResolution', 1, ...            % deg
    'ElevationResolution', 5, ...          % deg
    'RangeResolution', 30, ...             % m
    'HasINS', true);

% Attach the radar to its airborne platform.
plat1.Sensors = radar;
```

Airborne Platform with Two Radar Arrays

Add a second airborne platform to the scenario traveling south at 550 km/hr at a cruising altitude of 8 km.

```
ht = 8e3;                               % Altitude in meters
spd = 550*1e3/3600;                     % Speed in m/s
start = [spd*sceneDuration/2 5e3 -ht];
stop  = [-spd*sceneDuration/2 5e3 -ht];
traj  = waypointTrajectory('Waypoints',[start;stop],'TimeOfArrival',[0; sceneDuration]);
plat2 = platform(scene,'Trajectory',traj);
```

Multiple sensors can be mounted on a platform. Add a radar composed of two linear phased arrays mounted 5 meters above the platform. Mount the arrays so that one array looks over the right side of the airframe and the other array looks over the left side of the airframe. Both arrays provide coverage over a 150 degree azimuth sector on either side of the platform. Elevation is not measured by the linear arrays. The specifications for this radar are tabulated below:

- Sensitivity: 0 dBsm @ 350 km

- Mechanical Scan: No

- Electronic Scan: Azimuth only

- Electronic Scan Limits: -75 to 75 deg

- Field of View: 1 deg azimuth, 60 deg elevation

- Measurements: Azimuth, range

- Azimuth Resolution: 1 deg

- Range Resolution: 30 m

Model the linear phased array radar using the monostaticRadarSensor.

```
% Create right facing radar by setting radar's yaw to 90 degrees.
sensorIndex = sensorIndex+1;
rightRadar = monostaticRadarSensor(sensorIndex, 'Sector', ...
    'SensorIndex', sensorIndex, ...
    'MountingLocation', [0 0 -5], ...   % m
    'MountingAngles', [90 0 0], ...     % deg, look over right side
    'UpdateRate', 12.5, ...             % Hz
    'ReferenceRCS', 0, ...              % dBsm
    'ReferenceRange', 350e3, ...        % m
    'ScanMode','Electronic', ...
    'ElectronicScanLimits',[-75 75], ... % deg
    'FieldOfView',[1;60], ...           % deg
```

```
    'HasElevation', false, ...
    'AzimuthResolution', 1, ...          % deg
    'RangeResolution', 30, ...           % m
    'HasINS', true);

% Create an identical radar looking over the left side of the airframe.
leftRadar = clone(rightRadar);
sensorIndex = sensorIndex+1;
leftRadar.SensorIndex = sensorIndex;
leftRadar.MountingAngles(1) = -90; % Look over the left side

% Attach the two linear radar arrays to the airborne platform.
plat2.Sensors = {leftRadar, rightRadar};
```

Ground-Based Platform with Rectangular Radar Array

Add a ground-based radar using a rectangular phased array mounted 5 meters above its trailer. The radar electronically surveys a 60 degree azimuth span and 20 degrees of elevation above the ground using an electronic raster scan pattern.

- Sensitivity: 0 dBsm @ 350 km

- Mechanical Scan: No

- Electronic Scan: Azimuth and elevation

- Electronic Scan Limits: -30 to 30 deg azimuth, -20 to 0 deg elevation

- Field of View: 1 deg azimuth, 5 deg elevation

- Measurements: Azimuth, elevation, range

- Azimuth Resolution: 1 deg

- Elevation Resolution: 5 deg

- Range Resolution: 30 m

Model the rectangular phased array radar using the `monostaticRadarSensor`.

```
% Create an electronically scanning rectangular array radar.
sensorIndex = sensorIndex+1;
radar = monostaticRadarSensor(sensorIndex, 'Raster', ...
    'MountingLocation', [0 0 -5], ...    % m
    'UpdateRate', 25, ...                % Hz
    'ReferenceRCS', 0, ...               % dBsm
    'ReferenceRange', 350e3, ...         % m
    'ScanMode','Electronic', ...
    'ElectronicScanLimits',[-30 30;-20 0],... % deg
    'FieldOfView', [1;5], ...            % deg
    'HasElevation', true, ...
    'AzimuthResolution', 1, ...          % deg
    'ElevationResolution', 5, ...        % deg
    'RangeResolution', 30, ...           % m
    'HasINS', true);

% Attach the rectangular radar array to the trailer platform.
plat3 = platform(scene,'Sensors',radar);
plat3.Trajectory.Position = [-30e3 30e3 0];
plat3.Trajectory.Orientation = quaternion([-60 0 0],'eulerd','zyx','frame');
```

Airborne Targets

Add four airborne targets within the surveillance region.

1 Airliner traveling northeast at 700 km/hr at a 3,000 m altitude

2 Crossing airliner traveling southeast at 900 km/hr at a 4,000 m altitude

3 Airliner traveling east at 600 km/hr at a 9,000 m altitude

4 Jet traveling at 300 km/hr and executing a 90 degree turn at a 3,000 m altitude

```matlab
% Add airliner traveling northeast.
ht = 3e3;                               % Altitude in meters
spd = 700*1e3/3600;                     % Speed in m/s
ang = 45;
rot = [cosd(ang) sind(ang) 0;-sind(ang) cosd(ang) 0; 0 0 1];
offset = [-15e3 -25e3 -ht];
start = offset - [spd*sceneDuration/2 0 0]*rot;
stop  = offset + [spd*sceneDuration/2 0 0]*rot;
traj  = waypointTrajectory('Waypoints',[start;stop],'TimeOfArrival',[0; sceneDuration]);

rcs = rcsSignature('Pattern', [10 10; 10 10], ...
        'Azimuth', [-180 180], 'Elevation', [-90 90], ...
        'Frequency', [0 10e9]);         % Define custom RCS signature of target
platform(scene,'Trajectory',traj,'Signatures',rcs);

% Add crossing airliner traveling southeast.
ht = 4e3;                               % Altitude in meters
spd = 900*1e3/3600;                     % Speed in m/s
offset = [(start(1)+stop(1))/2 (start(2)+stop(2))/2 -ht];
start = offset + [0 -spd*sceneDuration/2 0]*rot;
stop  = offset + [0  spd*sceneDuration/2 0]*rot;
traj  = waypointTrajectory('Waypoints',[start;stop],'TimeOfArrival',[0; sceneDuration]);
rcs = rcsSignature;                     % Default 10 dBsm RCS at all viewing angles
platform(scene,'Trajectory',traj,'Signatures',rcs);

% Add eastbound airliner.
ht = 9e3;                               % Altitude in meters
spd = 600*1e3/3600;                     % Speed in m/s
start = [30e3 -spd*sceneDuration/2-20e3 -ht];
stop  = [30e3  spd*sceneDuration/2-20e3 -ht];
traj  = waypointTrajectory('Waypoints',[start;stop],'TimeOfArrival',[0; sceneDuration]);
platform(scene,'Trajectory',traj);      % Default 10 dBsm RCS at all viewing angles

% Add jet turning with horizontal acceleration of 0.3 G.
ht = 3e3;                               % Altitude in meters
spd = 300*1e3/3600;                     % Speed in m/s
accel = 0.3*9.8;                        % Centripetal acceleration m/s^2
radius = spd^2/accel;                   % Turn radius in meters
t0 = 0;
t1 = t0+5;
t2 = t1+pi/2*radius/spd;
t3 = sceneDuration;
start = [0e4 -4e4 -ht];
wps = [ ...
    0                   0                   0; ... % Begin straight segment
    spd*t1              0                   0; ... % Begin horizontal turn
    spd*t1+radius       radius              0; ... % End of horizontal turn
    spd*t1+radius       radius+spd*(t3-t2)  0];    % End of second straight segment
```

```matlab
traj  = waypointTrajectory('Waypoints',start+wps,'TimeOfArrival',[t0; t1; t2; t3]);
platform(scene,'Trajectory',traj);
```

Generation of Radar Detections

The following loop advances the platform and target positions until the end of the scenario. For each step forward in the scenario, detections are generated from each platform.

The `trackingScenario` can advance at a fixed time interval or automatically determine the next update time. Set the `UpdateRate` to `0` to let `trackingScenario` determine the next update time.

```matlab
rng(2018); % Set random seed for repeatable results

% Create a theaterPlot to show the true and measured positions of the detected targets and platf
theaterDisplay = helperMultiPlatDisplay(scene);

title('Multiplatform Radar Scenario');
legend('show');

% Show 3D view of the scenario.
view(-60,10);

% Log all the detections
detLog = {};
timeLog = [];

while advance(scene)
    % Generate detections from radars on each platform.
    [dets,configs] = detect(scene);

    % Update display with current beam positions and detections.
    theaterDisplay(dets);

    % Log sensor data and ground truth.
    detLog = [detLog; dets]; %#ok<AGROW>
    timeLog = [timeLog; scene.SimulationTime];%#ok<AGROW>
end
```
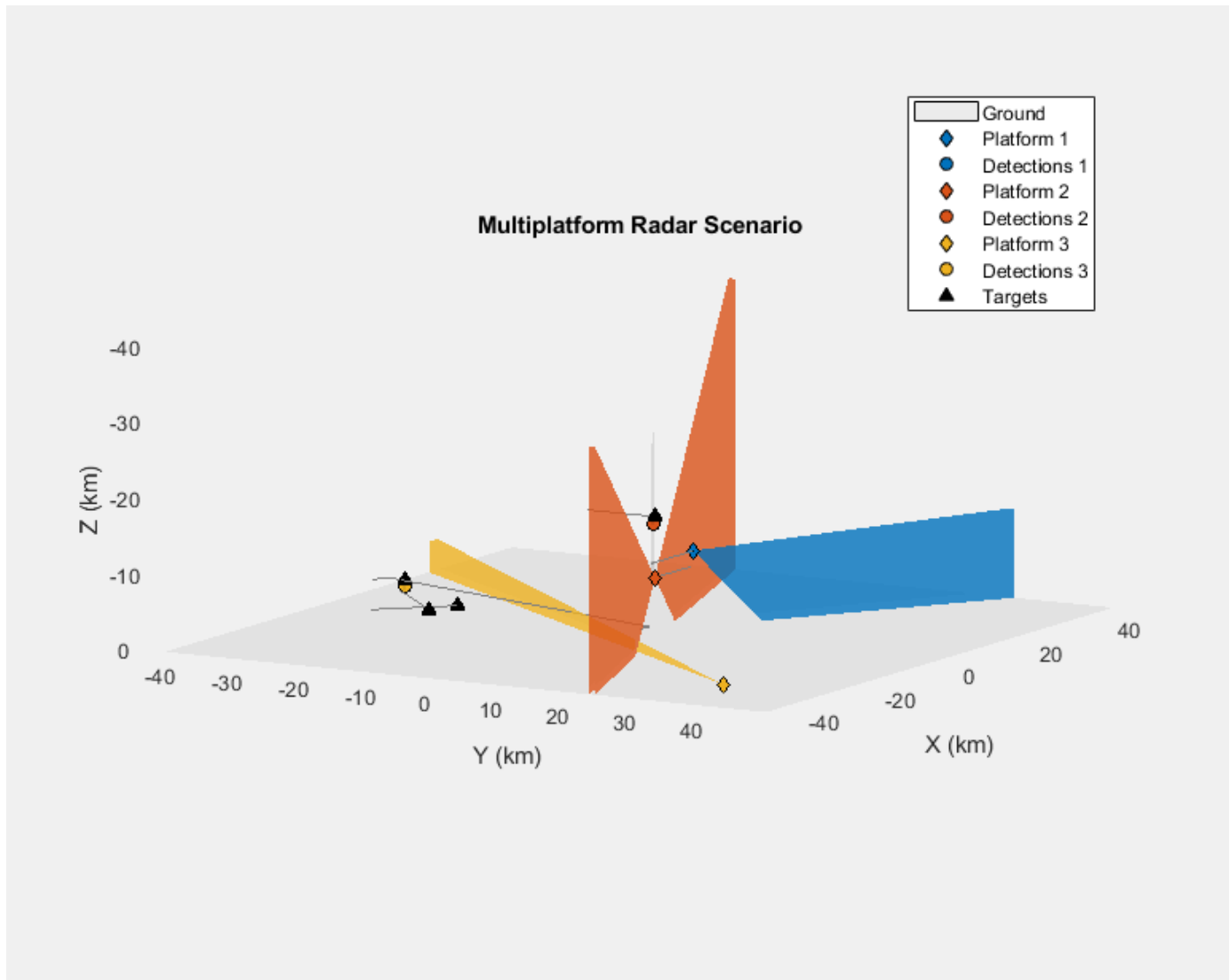
Notice the wide beams from the airborne platforms and the narrow beam from the ground-based radar executing a raster scan. You can visualize the ground truth trajectories in the 2D view below. The four targets are represented by triangles. Around 30 km on the x-axis is the airliner traveling east (left to right). Around 2 km on the x-axis is the jet executing a turn clockwise. Further south are two crossing airliners.

```
view(-90,90); % 2D view
```
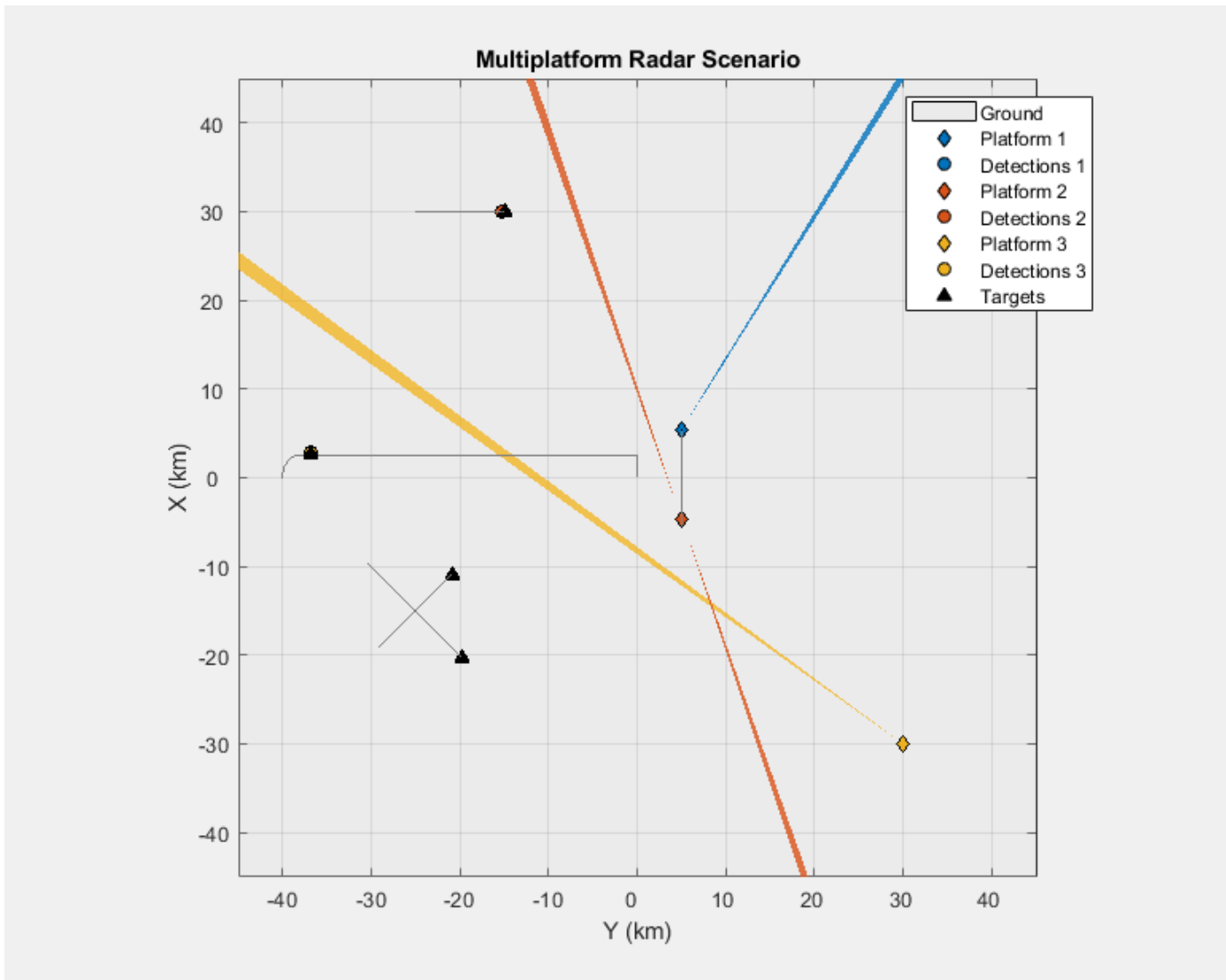
Plot the logged detections with their measurement uncertainties. Each color corresponds to the platform generating the detections. The legend from the previous display applies to all the following plots. Notice that the radars generate false alarms, which are detection far away from the target trajectories.

```
theaterDisplay(detLog);
title([num2str(numel(detLog)) ' Detections Logged from ' num2str(numel(timeLog)) ' Simulation Ste
legend('hide');
```

110 Detections Logged from 1501 Simulation Steps
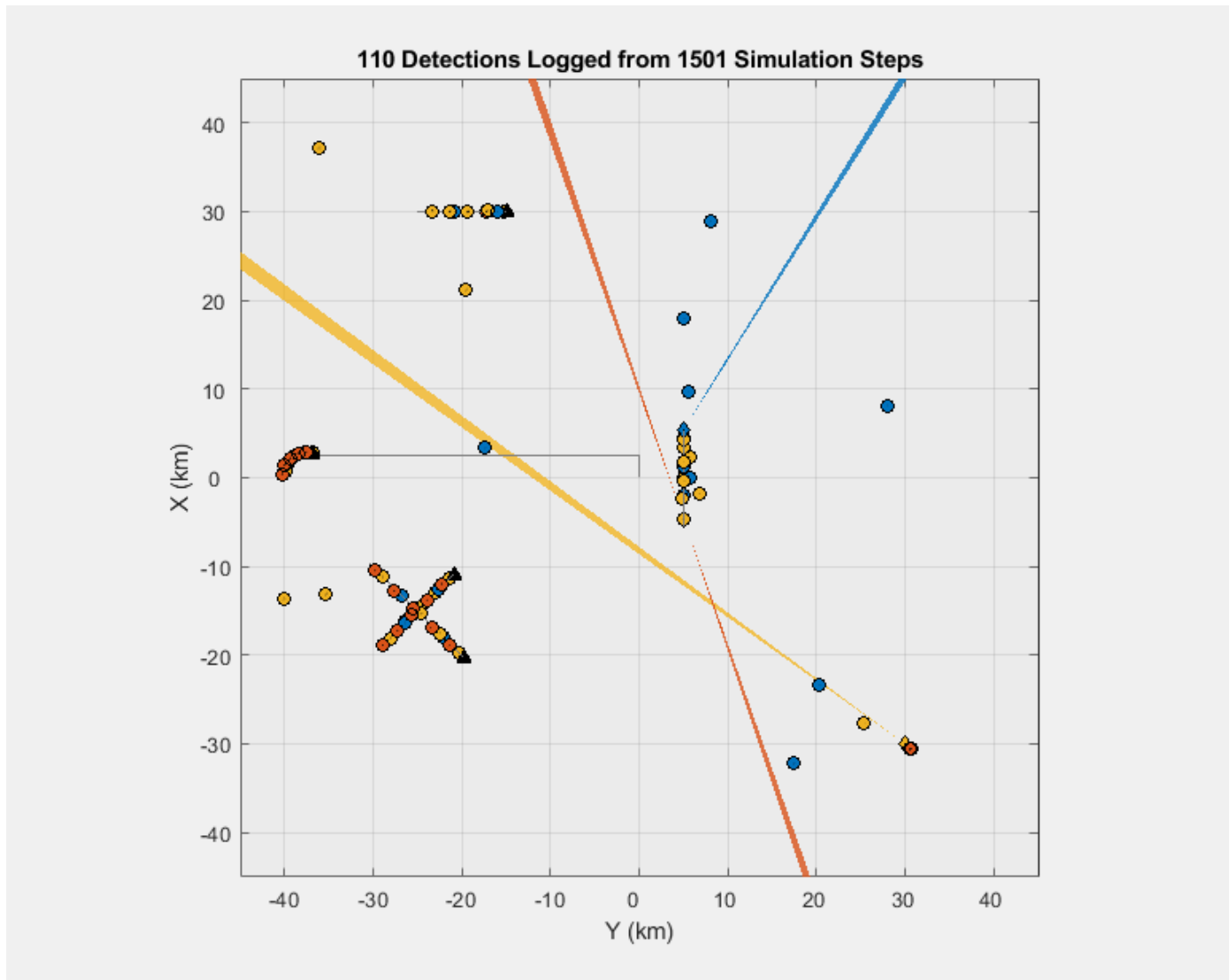
The following 3D view shows how these detections are distributed in elevation. For platforms with 3D sensors (the blue and yellow platforms), the detections closely follow the target trajectories. The 2D-view platform's detections (the red platform) are offset in elevation from the target trajectories because its radar is unable to measure in elevation. The 1-sigma measurement uncertainty is shown for each detection as a gray ellipsoid centered on the measured target positions (shown as filled circles).

```
view([-60 25]); % 3D view
```

110 Detections Logged from 1501 Simulation Steps
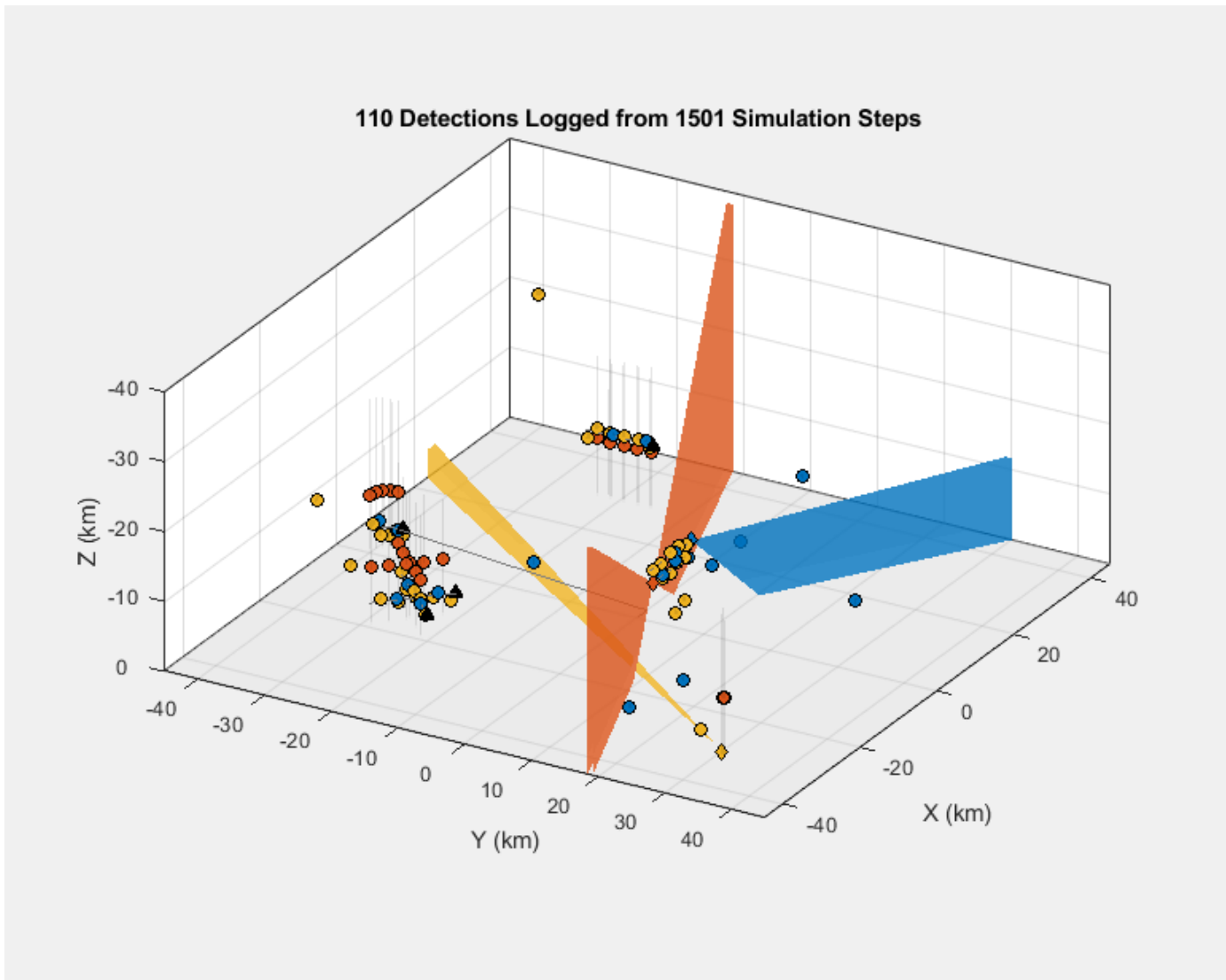
Zoom in on the jet executing the 90 degree horizontal turn. The 1-sigma measurement uncertainty is reported by the radar according to the radar's resolution and the signal-to-noise ratio (SNR) for each detection. Targets at longer ranges or with smaller SNR values will have larger measurement uncertainties than targets at closer ranges or with larger SNR values. Notice that the blue detections have smaller measurement uncertainties than the yellow detections. This is because the blue detections originate from the airborne platform (Platform 1) that is much closer to the target than the ground-based platform (Platform 3) generating the yellow detections.

```
xlim([-3000 9000]); ylim([-44000 -32000]); zlim([-12000 0000]);
axis('square');
title('Jet Executing Horizontal Turn');
```

Jet Executing Horizontal Turn
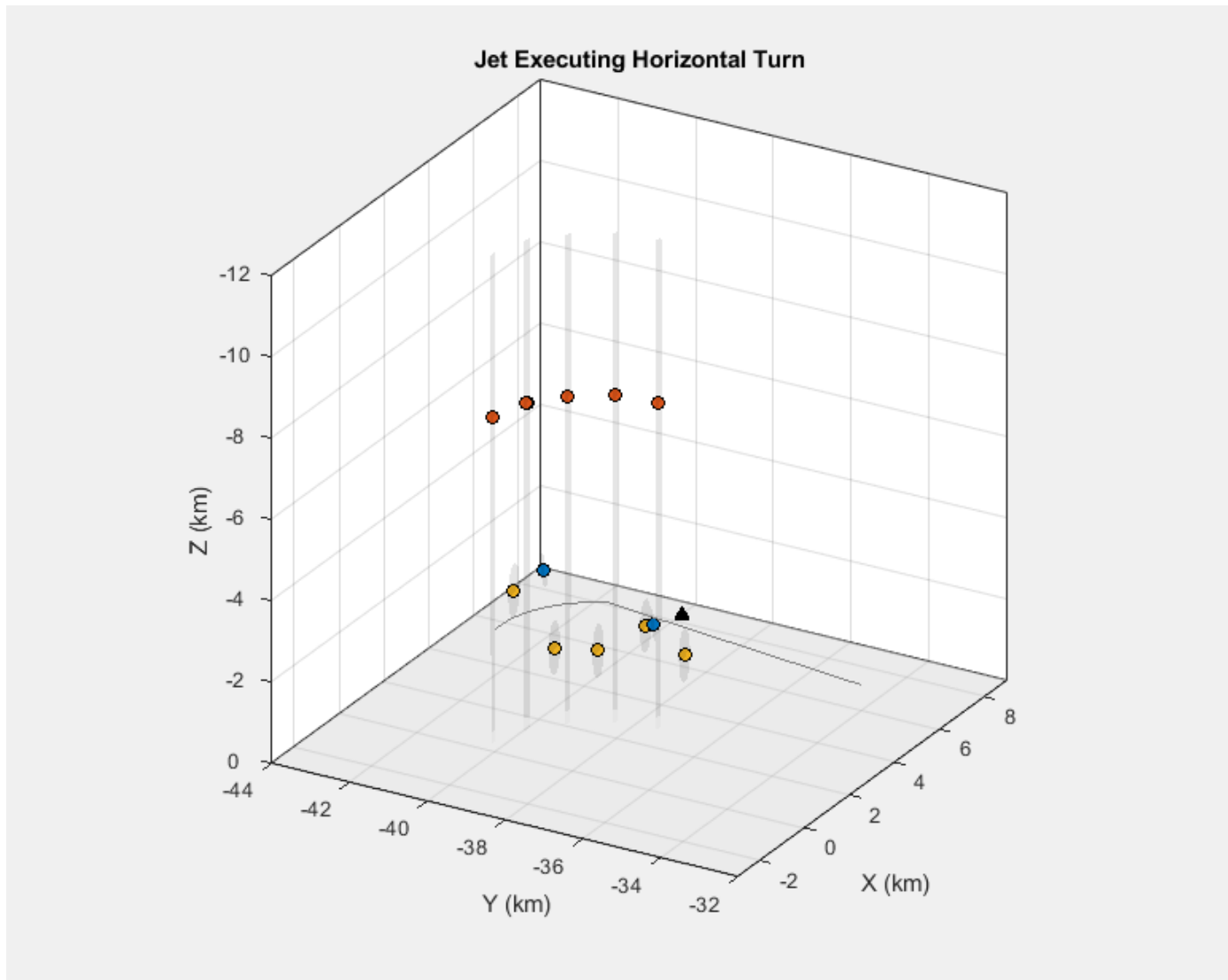
Notice the large uncertainty in elevation of the red detections generating from the airborne platform (Platform 2) that uses two linear arrays. The ellipsoids have small axes in the range and azimuth directions but have very large axes along the elevation direction. This is because the linear arrays on this platform are unable to provide estimates in elevation. In this case, the platform's radar reports detections at 0 degrees with an uncertainty in elevation corresponding to the elevation field of view.

Zoom in on the two crossing airliners. The blue airborne radar with the rotating array generates the fewest number of detections (only 4 detections for these two targets), but these detections are the most precise (smallest ellipses). The small number of detections from this platform is due to its radar's 360 mechanical scan, which limits how frequently its beam can revisit a target in the scenario. The other platforms have radars with smaller scan regions, allowing them to revisit the targets at a higher rate.

```
view([-55 20]);
xlim([-22000 -10000]); ylim([-31000 -19000]);
title('Crossing Airliners');
```
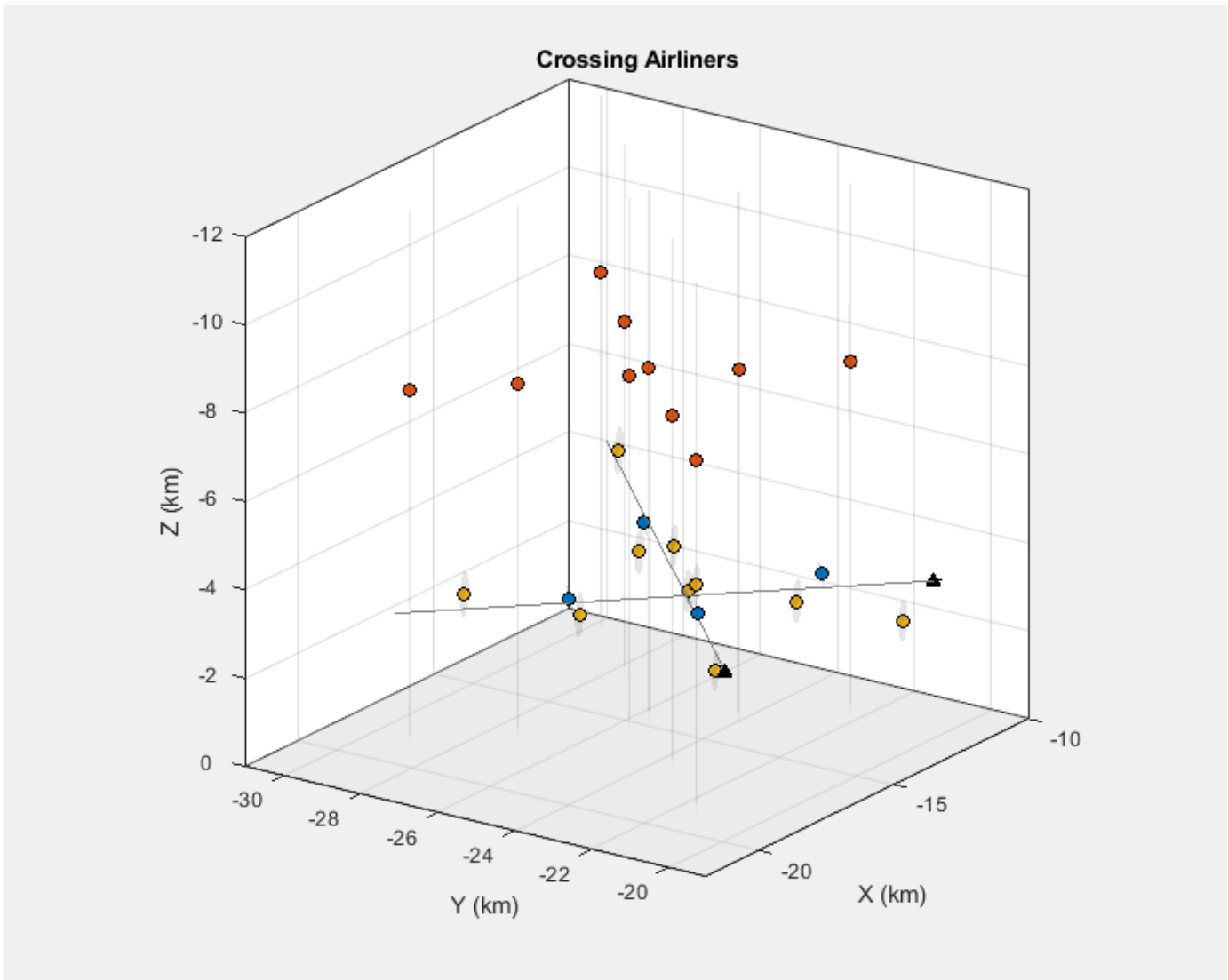
Zoom in on the airliner traveling east. Same observations on the number of detections and accuracy from different radar platforms apply.

```
view([-70 10]);
xlim([24000 36000]); ylim([-26000 -14000]); zlim([-15000 -3000])
title('Airliner Traveling East');
```

Airliner Traveling East
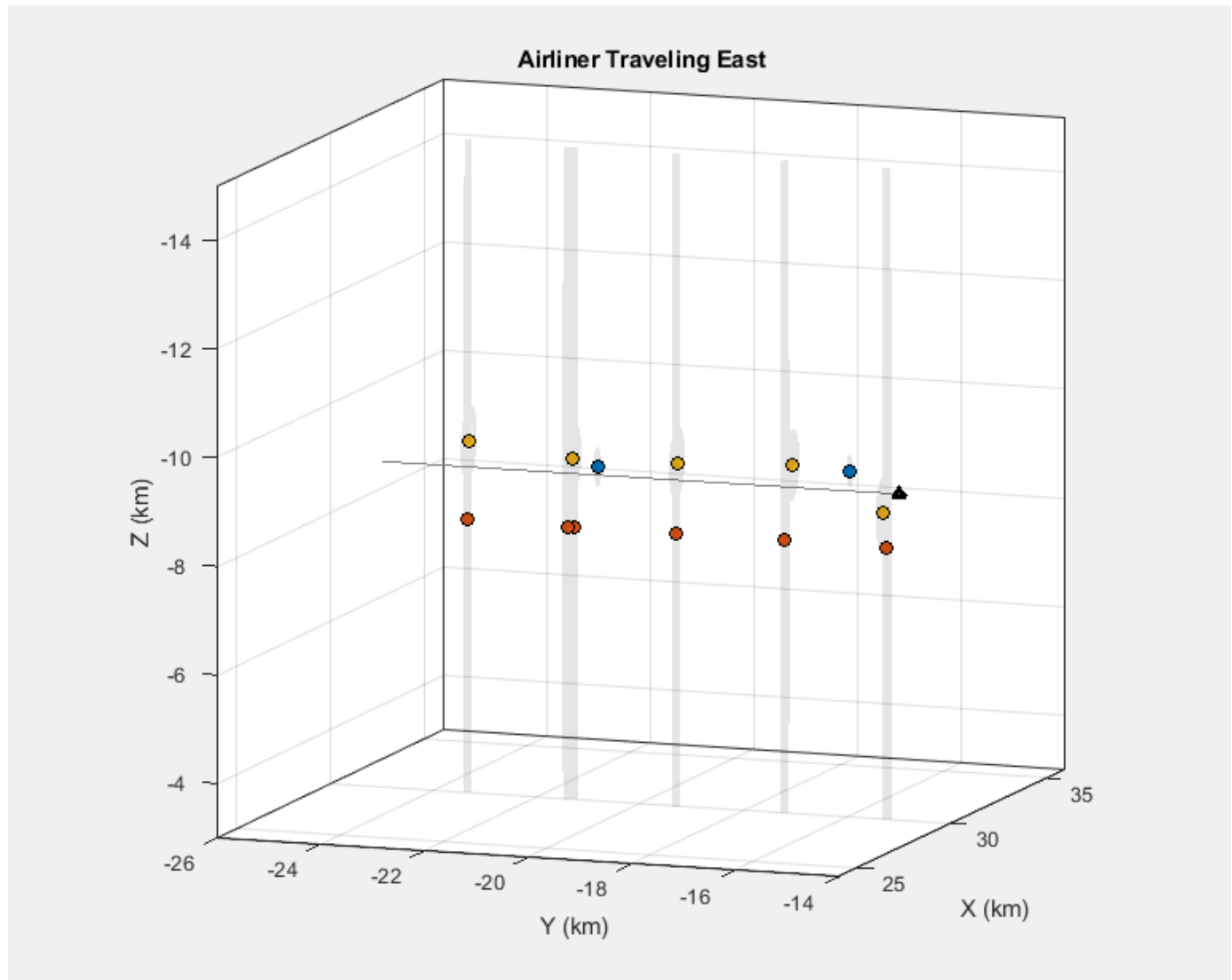
Summary

This example shows how to model a radar surveillance network and simulate detections generated by multiple airborne and ground-based radar platforms. In this example, you learned how to define scenarios, including targets and platforms that can be stationary or in motion. You also learned how to visualize the ground truth trajectories, sensor beams, detections, and associated measurement uncertainties. You can process this synthetic data through your tracking and fusion algorithms to assess their performance for this scenario. You can also modify this example to exercise your multi-target tracker against different target types and maneuvers.

Tracking Using Distributed Synchronous Passive Sensors

This example illustrates the tracking of objects using measurements from spatially-distributed and synchronous passive sensors. In the "Passive Ranging Using a Single Maneuvering Sensor" on page 5-256, you learned that passive measurements provide incomplete observability of a target's state and how a single sensor can be maneuvered to gain range information. Alternatively, multiple stationary sensors can also be used to gain observability. In this example, you will learn how to track multiple objects by fusing multiple detections from passive synchronous sensors.

Introduction

In the synchronized multisensor-multitarget tracking problem, detections from multiple passive sensors are collected synchronously and are used to estimate the following:

- Number of targets in the scenario
- Position and velocity of those targets

This example demonstrates the use of the *Static Fusion Before Tracking* [1] architecture for tracking using passive measurements. The *Static Fusion* part of the architecture aims to triangulate the most likely set of detections and output fused detections containing estimated positions of targets. As measurements need to be fused together by static fusion, the sensors must report measurements synchronously.
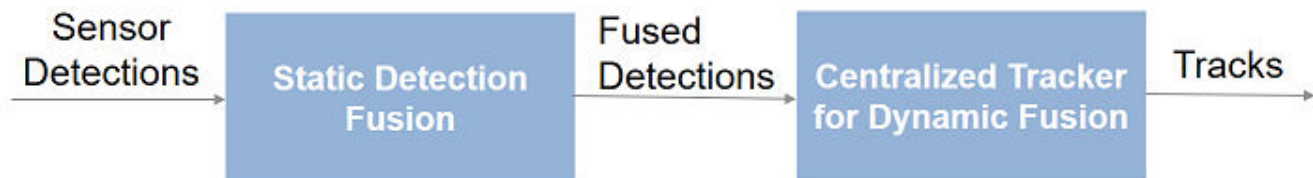
Static Fusion Before Tracking

With measurements containing only line-of-sight (LOS) information, at least 2 sensors are needed to find the position. However, with 2 sensors, the problem of ghosting (intersections at points with no targets) occurs when multiple targets lie in the same plane. With 2 targets and 2 sensors, it is impossible to identify the correct pair from a single frame of measurements as demonstrated in the figure below:
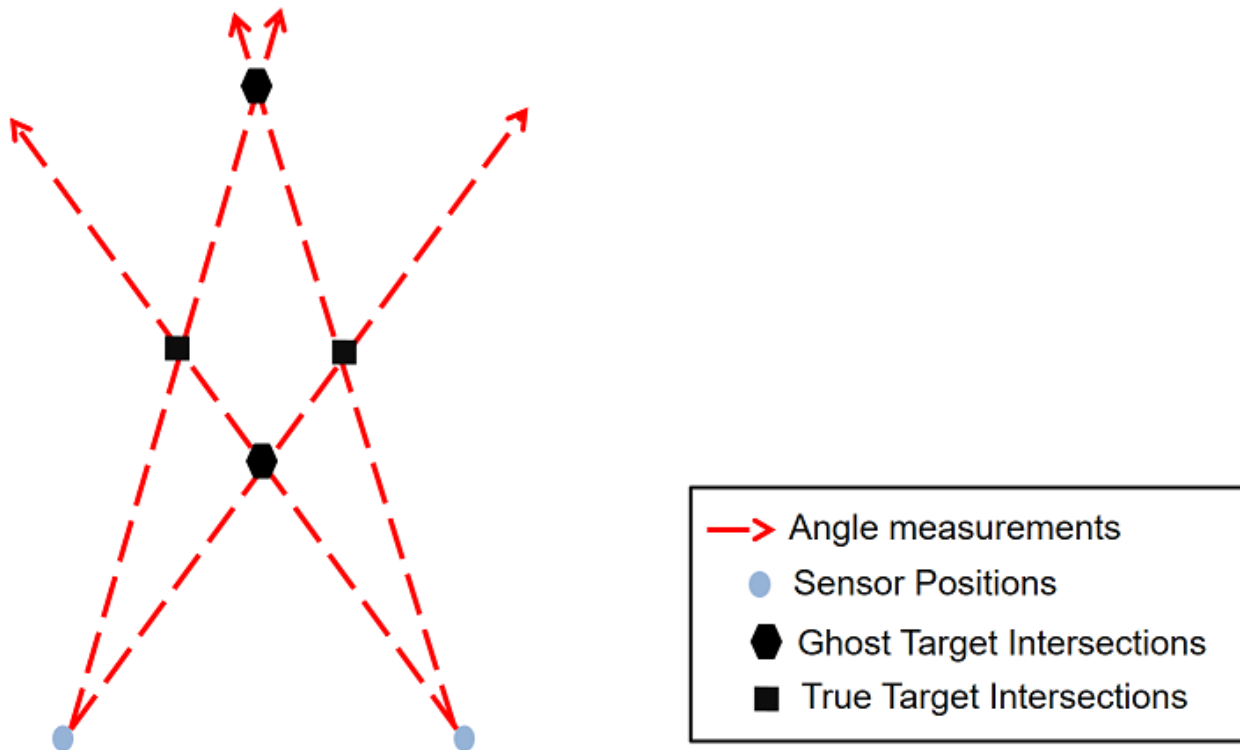
Therefore, one must use 3 or more sensors to reduce the problem of ghosting. Due to the presence of measurement noise and false measurements, it is difficult to eliminate the problem of ghosting completely. Ghost triangulations returned by static association are likely to be discarded by the dynamic association blocks as the geometry of targets and sensors changes during the scenario.

Define Scenario

The relative placement of sensors and targets in the scenario used here is taken from an example in [1]. The scenario consists of five equally-spaced targets observed by three to five passive sensors. The passive detections are modeled using `radarEmitter` and `radarSensor` with `DetectionMode` set to `ESM`. The `HasNoise` property of the sensors is set to `false` to generate noise-free detections along with false alarms. Noise to measurements is added in this example via a user-controlled variable. This is to simulate the effect of sensor noise on static fusion. Each sensor has a field of view of 180 degrees in azimuth and a `FalseAlarmRate` of 1e-3 per azimuth resolution cell. This results in 2 to 3 false alarms per scan. The scenario definition is wrapped inside a helper function `helperGenerateFusionScenarioData`.

```
[detectionBuffer,truthLog,theaterDisplay] = helperGenerateStaticFusionScenarioData;
showScenario(theaterDisplay);
```

Sensor and Target Trajectories
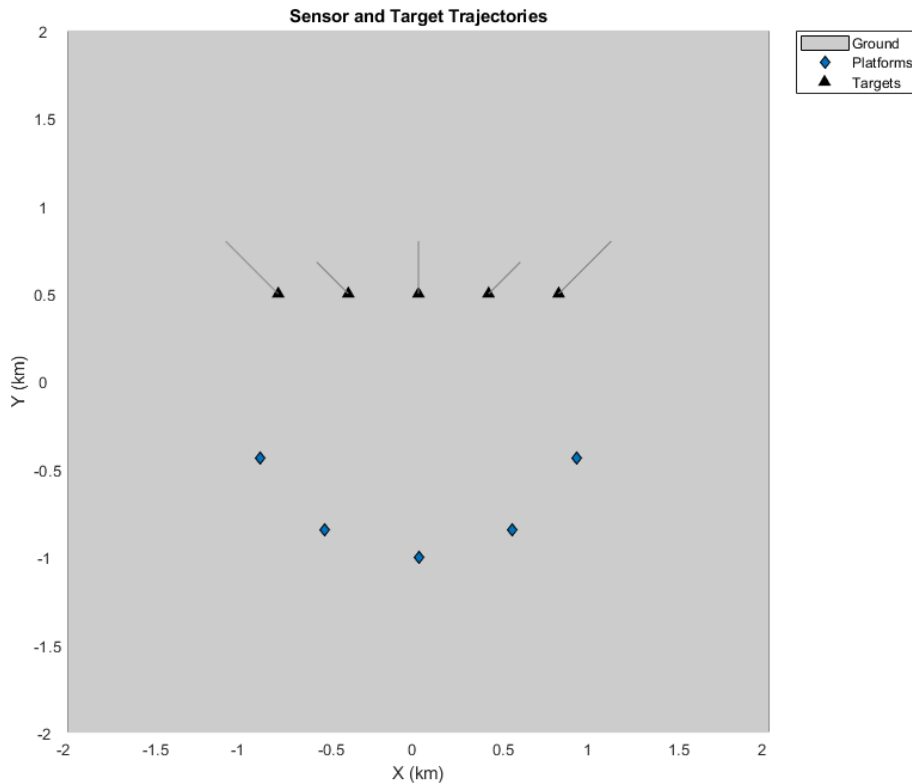
showGrabs(theaterDisplay,[]);

Track with Three Sensors

In this section, only measurements from the inner three sensors are considered and measurement noise covariance for each sensor is set to 0.01 degrees squared.

The detections from each sensor are passed to a `staticDetectionFuser`. The `MeasurementFusionFcn` for passive triangulation is specified as `triangulateLOS`. The `MeasurementFusionFcn` allows specifying a function to fuse a given combination of detections (at most one detection from each sensor) and return the fused position and its error covariance. The parameters `FalseAlarmRate`, `Volume` and `DetectionProbability` are specified to reflect the parameters of sensors simulated in this scenario. These parameters are used to calculate the likelihood of feasible associations. The `UseParallel` property, when set to `true`, allows the fuser to evaluate the feasible associations using parallel processors.

The tracking is performed by GNN data association by using a `trackerGNN`.

The tracking performance is evaluated using `trackAssignmentMetrics` and `trackErrorMetrics`.

Setup

```
% Number of sensors
numSensors = 3;
```

```
% Create a detection fuser using triangulateLOS function as the
% MeasurementFusionFcn and specify parameters of sensors.
fuser = staticDetectionFuser('MeasurementFusionFcn',@triangulateLOS,...
    'MaxNumSensors',numSensors,...
    'UseParallel',true,...
    'FalseAlarmRate',1e-3,...
    'Volume',0.0716,...
    'DetectionProbability',0.99);

% Tracking using a GNN tracker
tracker = trackerGNN('AssignmentThreshold',50,...
    'ConfirmationThreshold',[3 5],'DeletionThreshold',[4 5]);

% Use assignment and error metrics to compute accuracy.
trackingMetrics = trackAssignmentMetrics('DistanceFunctionFormat','custom',...
    'AssignmentDistanceFcn',@trueAssignment,'DivergenceDistanceFcn',@trueAssignment);
errorMetrics = trackErrorMetrics;
```

Run simulation with three sensors

```
% Measurement noise
measNoise = 0.01;

time = 0;
dT = 1; % 1 Hz update rate of scenario.

% Loop through detections and track targets
for iter = 1:numel(detectionBuffer)
    % Truth information
    time = time + dT;
    sensorPlatPoses = truthLog{iter}(1:numSensors);
    targetPlatPoses = truthLog{iter}(6:end);
    groundTruth = [sensorPlatPoses;targetPlatPoses];

    % Generate noisy detections using recorded detections
    thisBuffer = detectionBuffer{iter};
    availableDetections = vertcat(thisBuffer{1:numSensors});
    noiseDetections = addNoise(availableDetections,measNoise);

    % Fuse noisy detections using fuser
    fusedDetections = fuser(noiseDetections);

    % Run a tracker on fused detections
    confTracks = tracker(fusedDetections,time);

    % Update track and assignment metrics
    trackingMetrics(confTracks,targetPlatPoses);
    [trackIDs,truthIDs] = currentAssignment(trackingMetrics);
    errorMetrics(confTracks,trackIDs,targetPlatPoses,truthIDs);

    % Update theater display
    detsToPlot = [noiseDetections(:);fusedDetections(:)];
    theaterDisplay(confTracks,detsToPlot,groundTruth);
end
axes(theaterDisplay.TheaterPlot.Parent);
```
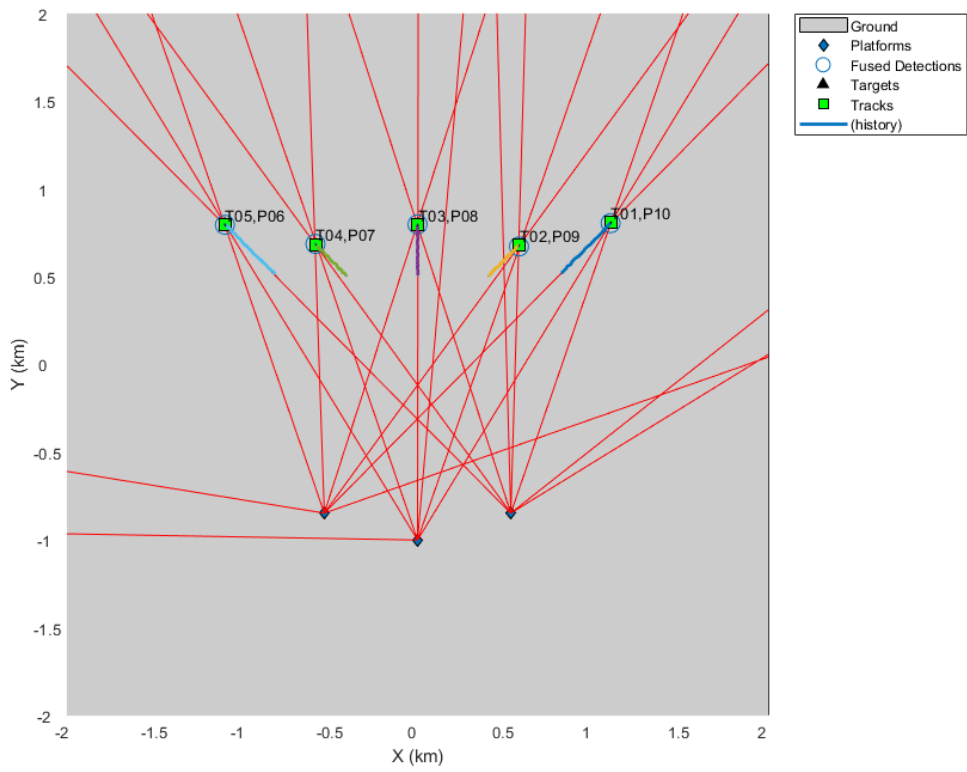
```
ylim([0 1.5]);
```
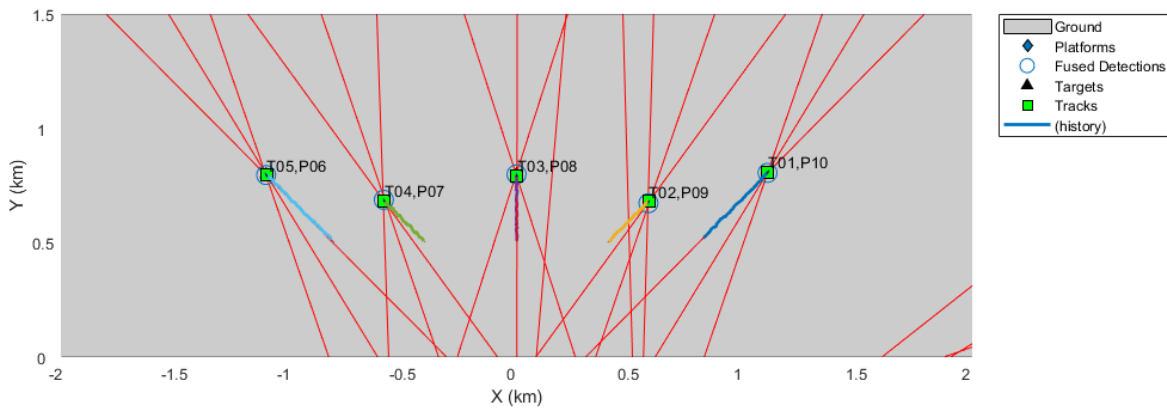
Results from tracking using three sensors with 0.01 degrees-squared of noise covariance can be summarized using the assignment metrics. Note that all tracks were assigned to the correct truths and no false tracks were confirmed by the tracker. These results indicates good static association accuracy.

```
assignmentTable = trackMetricsTable(trackingMetrics);
assignmentTable(:,{'TrackID','AssignedTruthID','TotalLength','FalseTrackStatus'})
```

ans =

  5×4 table

TrackID	AssignedTruthID	TotalLength	FalseTrackStatus
1	10	58	false
2	9	58	false
3	8	58	false
4	7	58	false
5	6	58	false

The error in estimated position and velocity of the targets can be summarized using the error metrics. The errors in position and velocity are within 7 meters and 2 meters/sec respectively for all targets and the normalized errors are close to 1. The error metrics indicate good dynamic association and tracking performance.

```
disp(cumulativeTrackMetrics(errorMetrics));
```

TrackID	posRMS	velRMS	posANEES	velANEES
1	5.6264	1.2647	1.8461	0.56551
2	3.452	1.3206	1.2788	0.59367
3	4.689	1.5054	2.011	0.80126
4	3.9549	1.2235	1.562	0.60194
5	4.7398	1.6053	1.7627	0.74181

Effect of measurement accuracy

The fusion of passive detections to eliminate ghosting is highly dependent on the accuracy of passive measurements. As measurement noise increases, the distinction between ghost associations and true associations becomes less prominent, resulting in a significant drop in the accuracy of static association. With closely spaced targets, incorrect association of fused detections to tracks may also occur. In the next section, a helper function `helperRunStaticFusionSimulation` is used to re-run the scenario with a measurement noise covariance of 2 degrees squared.

Run the scenario again with a high measurement noise

```
numSensors = 3;
measNoise = 2; %standard deviation of sqrt(2) degrees
[trackingMetrics,errorMetrics] = helperRunStaticFusionSimulation(detectionBuffer,truthLog,numSens
axes(theaterDisplay.TheaterPlot.Parent);
```
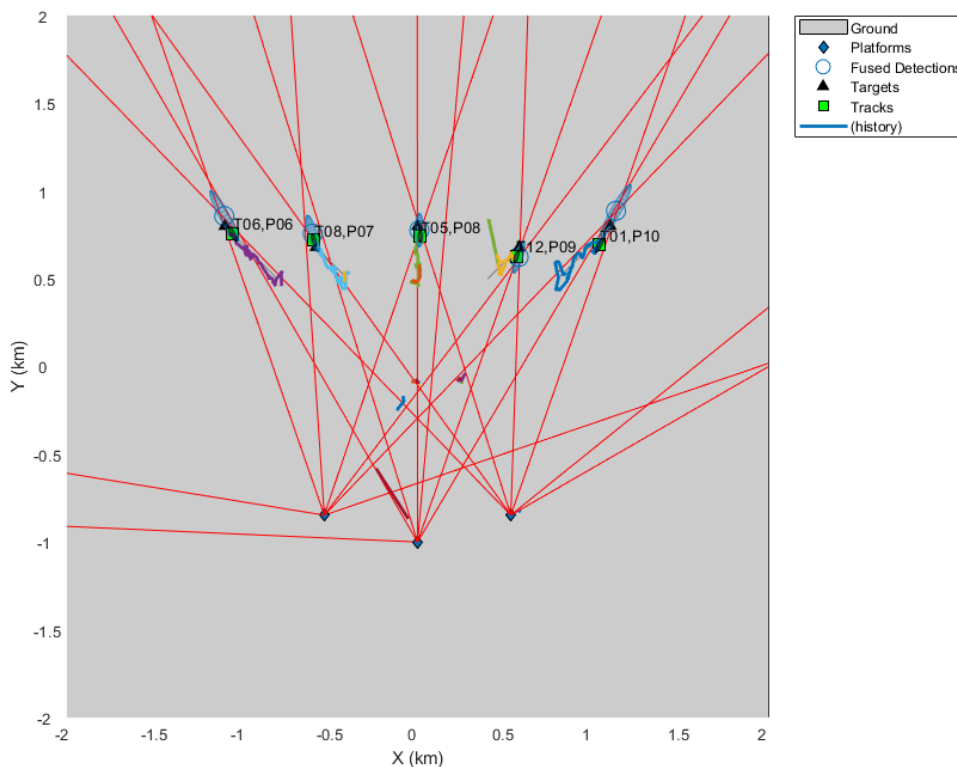
```
ylim([0 1.5]);
```
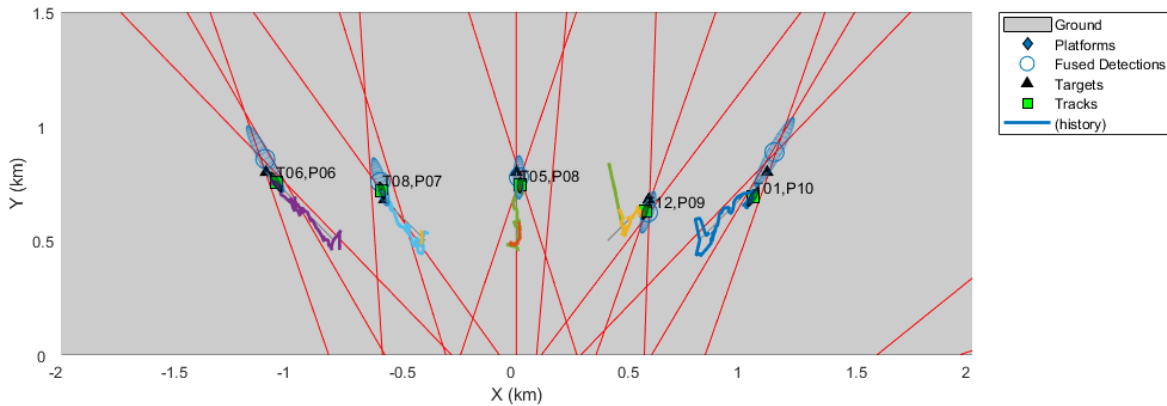

Note that a large number of tracks were confirmed and then dropped in this simulation. Poor static association accuracy leads to ghost target triangulations more often, which results in tracker deleting these tracks due to multiple misses.

```
assignmentTable = trackMetricsTable(trackingMetrics);
assignmentTable(:,{'TrackID','AssignedTruthID','TotalLength','FalseTrackStatus'})
```

ans =

  12×4 table

TrackID	AssignedTruthID	TotalLength	FalseTrackStatus
1	10	58	false
2	NaN	11	true
3	NaN	8	true
5	8	57	false
6	6	58	false
8	7	57	false
12	9	51	false
13	NaN	7	true
22	NaN	6	false
23	NaN	3	true

```
       27              NaN              9               false
       32              NaN              2               false
```

The estimated error for each truth is higher. Notice that the track jumps in the theater display above.

```
disp(cumulativeTruthMetrics(errorMetrics));
```

TruthID	posRMS	velRMS	posANEES	velANEES
6	620.97	10.426	810.2	8.9916
7	336.03	5.3142	32235	2.5579
8	81.688	3.0383	6.0661	0.56988
9	39.44	7.1249	1.5144	1.6229
10	328.32	8.499	1929.2	3.8069

The association accuracy can be improved by increasing the number of sensors. However, the computational requirements increase exponentially with the addition of each sensor. The static fusion algorithm spends most of the time computing the feasibility of each triangulation. This part of the algorithm is parallelized when the `UseParallel` property of the `staticDetectionFuser` is set to `true`, which provides a linear speed-up proportional to the number of processors. To further accelerate execution, you can also generate C/C++ code which will also run in parallel execution on multiple processors. You can learn the basics of code generation using MATLAB® Coder™ at Getting Started with MATLAB Coder.

Accelerate MATLAB Code Through Parallelization and Code Generation

To accelerate MATLAB Code for simulation, the algorithm must be restructured as a MATLAB function, which can be compiled into a MEX file or a shared library. For this purpose, the static fusion algorithm is restructured into a function. To preserve the state of the fuser between multiple calls, it is defined as a `persistent` variable.

```
type('mexFuser');
```

```
function [superDets,info] = mexFuser(detections)

%#codegen
persistent fuser

if isempty(fuser)
    fuser = staticDetectionFuser('MeasurementFusionFcn',@triangulateLOS,...
        'MaxNumSensors',5,...
        'UseParallel',true,...
        'FalseAlarmRate',1e-3,...
        'Volume',0.0716,...
        'DetectionProbability',0.99);
end

[superDets,info] = fuser(detections);
```

MATLAB® Coder™ requires specifying the properties of all the input arguments. An easy way to do this is by defining the input properties by example at the command line using the `-args` option. For more information, see "Define Input Properties by Example at the Command Line" (MATLAB Coder). To allow variable number of detections, you will use the `coder.typeof` function to allocate data types and sizes for the inputs.

```
% Get a sample detection from the stored buffer
sampleDetection = detectionBuffer{1}{1}{1};

% Use the coder.typeof function to allow variable-size inputs for
% detections.
maxNumDets = 500;
inputDets = coder.typeof({sampleDetection},[maxNumDets,1],[1 0]);

h = msgbox({'Generating code for function. This may take a few minutes...';...
    'This message box will close when done.'},'Codegen Message');

% Use the codegen command to generate code by specifying input arguments
% via example by using the |-args| option.
codegen mexFuser -args {inputDets};

close(h);
```

You can verify the speed-up achieved by code generation by comparing the time taken by them for fusing one frame of detections

```
testDetections = addNoise(vertcat(detectionBuffer{1}{1:5}),1);
tic;mexFuser(testDetections);t_ML = toc;
tic;mexFuser_mex(testDetections);t_Mex = toc;
disp(['MATLAB Code Execution time = ',num2str(t_ML)]);
disp(['MEX Code Execution time = ',num2str(t_Mex)]);
```

```
MATLAB Code Execution time = 23.767
MEX Code Execution time = 0.83915
```

Track with Five Sensors

In this section, detections from all five sensors are used for tracking and a measurement noise of 2 degrees squared is used.

```
measNoise = 2; % Same noise as 3 sensors
numSensors = 5;
[trackingMetrics,errorMetrics] = helperRunStaticFusionSimulation(detectionBuffer,truthLog,numSens
axes(theaterDisplay.TheaterPlot.Parent);
```
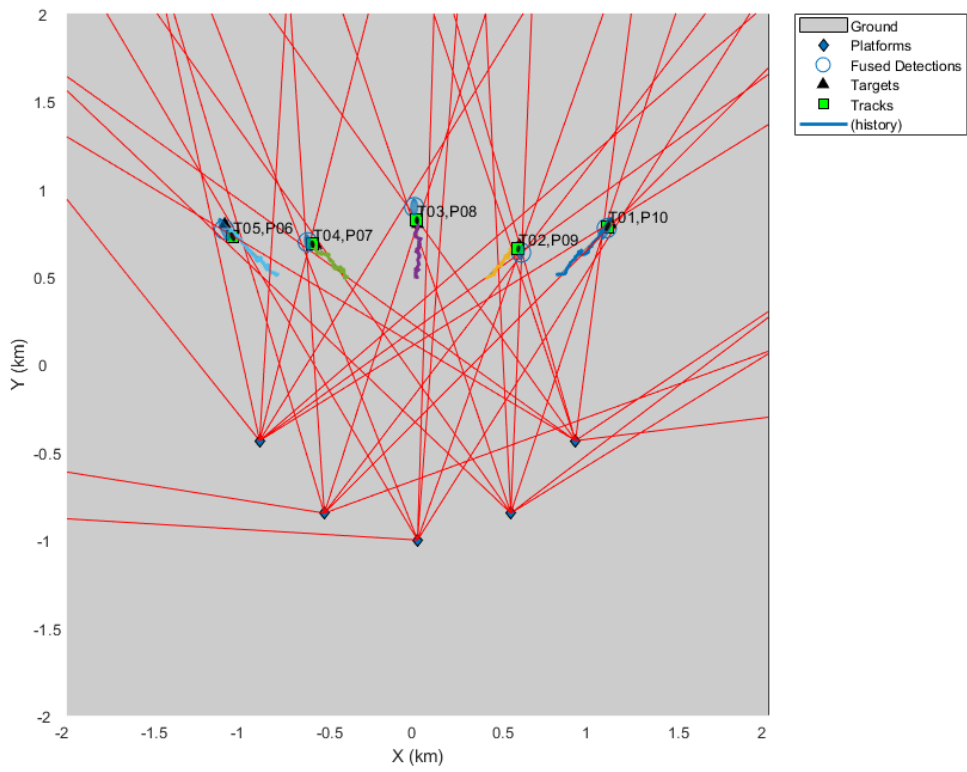
```
ylim([0 1.5]);
```
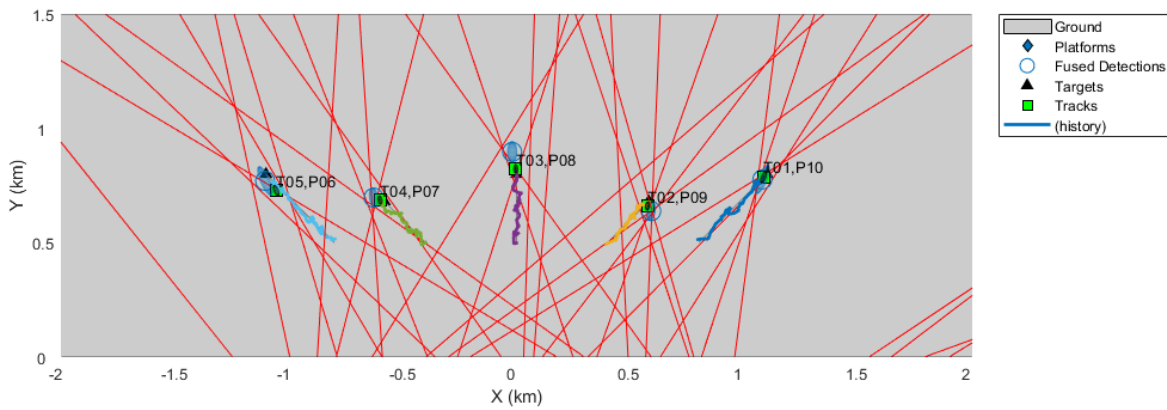
The assignment results from tracking using five sensors show that all truths were assigned a track during the entire simulation. There were also no track drops in the simulation as compared to 6 track drops in the low-accuracy three sensor simulation.

```
assignmentTable = trackMetricsTable(trackingMetrics);
assignmentTable(:,{'TrackID','AssignedTruthID','TotalLength','FalseTrackStatus'})
```

```
ans =

  5×4 table

    TrackID    AssignedTruthID    TotalLength    FalseTrackStatus
    _____    _____    _____    _____

       1             10               58              false
       2              9               58              false
       3              8               58              false
       4              7               58              false
       5              6               58              false
```

The estimated errors for positions are much lower for each true target as compared to the three sensor simulation. Notice that the estimation results for position and velocity do degrade as compared to three sensors with high-accuracy measurements.

```
disp(cumulativeTruthMetrics(errorMetrics))
```

TruthID	posRMS	velRMS	posANEES	velANEES
6	28.555	3.1714	2.381	0.74884
7	18.141	2.6749	1.8022	0.55938
8	17.165	2.5338	1.6855	0.64878
9	17.697	2.0996	2.1846	0.52041
10	25.686	3.4936	2.0517	0.66066

Summary

This example showed how to track objects using a network of distributed passive sensors. You learned how to use `staticDetectionFuser` to statically associate and fuse detections from multiple sensors. The example demonstrated how this architecture is depends parameters like number of sensors in the network and the accuracy of sensor measurements. The example also showed how to accelerate performance by utilizing parallel computing and automatically generating C code from MATLAB code.

Supporting Functions

trueAssignment Use ObjectAttributes of track to assign it to the right truth.

```
function distance = trueAssignment(track,truth)
tIDs = [track.ObjectAttributes.TargetIndex];
tIDs = tIDs(tIDs > 0);
if numel(tIDs) > 1 && all(tIDs == truth.PlatformID)
    distance = 0;
else
    distance = inf;
end
end
```

addNoise Add noise to detections

```
function dets = addNoise(dets,measNoise)
for i = 1:numel(dets)
    dets{i}.Measurement(1) = dets{i}.Measurement(1) + sqrt(measNoise)*randn;
    dets{i}.MeasurementNoise(1) = measNoise;
end
end
```

References

[1] Bar-Shalom, Yaakov, Peter K. Willett, and Xin Tian. "Tracking and Data Fusion: A Handbook of Algorithms." (2011).

Passive Ranging Using a Single Maneuvering Sensor

This example illustrates how to track targets using passive angle-only measurements from a single sensor. Passive angle-only measurements contain azimuth and elevation of a target with respect to the sensor. The absence of range measurements makes the problem challenging as the targets to be tracked are fully observable only under certain conditions.

In this example, you learn about some possible solutions to this problem by using a passive infrared sensor mounted on a maneuvering platform.

Introduction

The absence of range measurements from a target implies incomplete observability of the target state. The following figure depicts that angle-only measurements obtained by an observer traveling at a constant velocity results in multiple possible trajectories of a (presumed constant velocity) target.
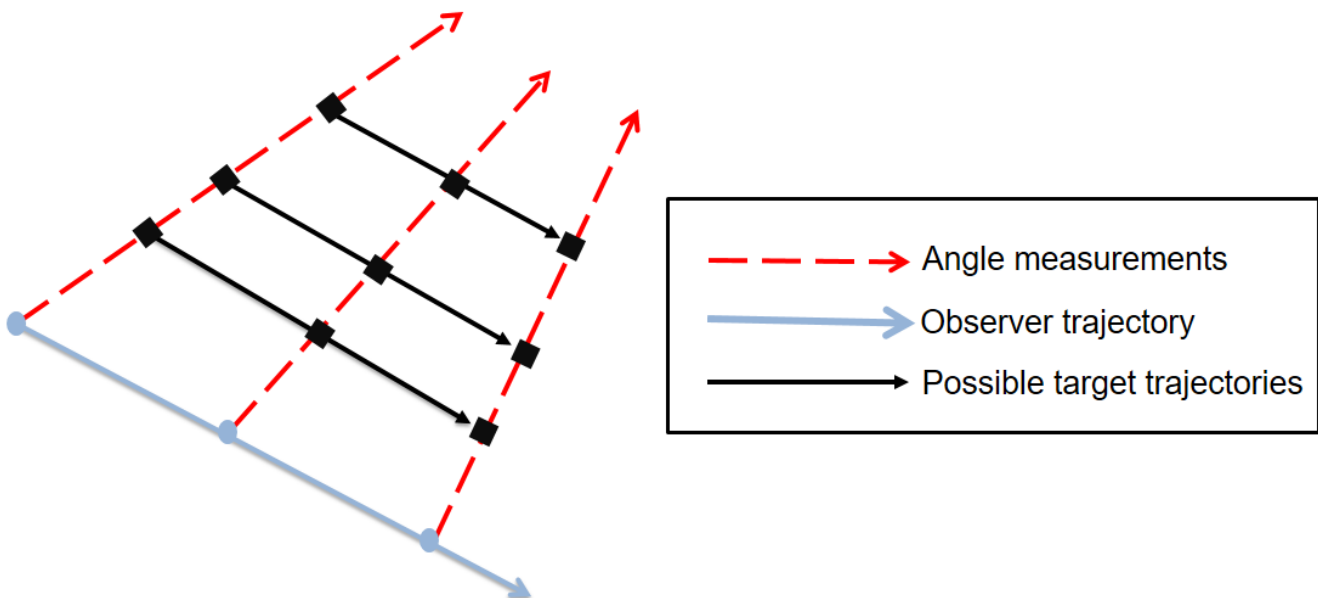
Theoretical results imply that the target state is unobservable until the following conditions are met [1].

- The sensor must out-maneuver the target i.e. the sensor motion must be at least 1 order higher than the target. For example, if a target is traveling at constant velocity, the observer must have at least a constant acceleration.
- The component of sensor maneuver perpendicular to line of sight must be non-zero.

Define Scenario

A helper function `helperCreatePassiveRangingScenario` is used to define a single infrared sensor mounted on a platform. The sensing platform, often termed as ownship, travels at constant velocity in the beginning and then performs a maneuver to observe the range of the target. The target is assumed to be non-maneuvering and travels at a constant velocity in the scenario.

```
% Setup
exPath = fullfile(matlabroot,'examples','fusion','main');
addpath(exPath);
[scene,ownship,theaterDisplay] = helperCreatePassiveRangingScenario;
showScenario(theaterDisplay);
```
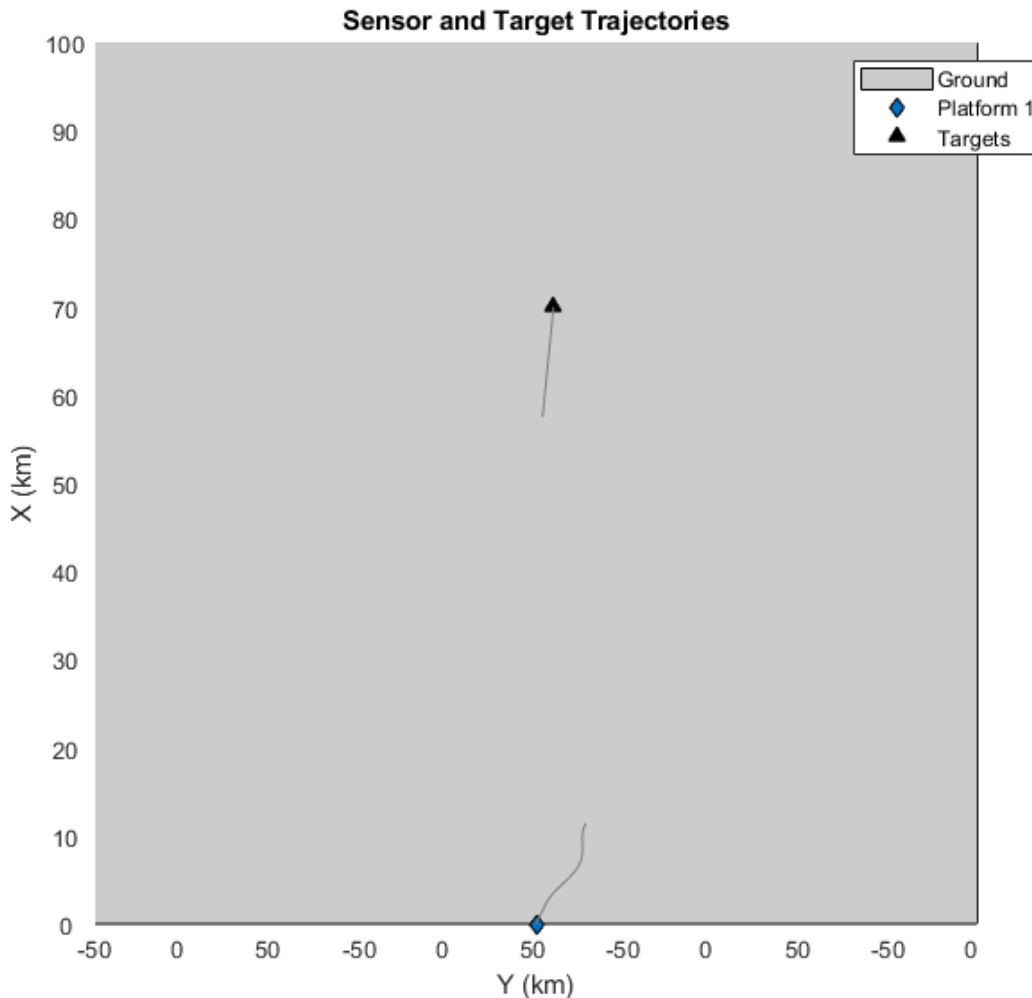

Track Using an EKF in Cartesian Coordinates

The problem of tracking a target using angle-only measurements can be formulated using an extended Kalman filter with a non-linear measurement model in Cartesian coordinates. In this section, a constant velocity `trackingEKF`, describing the state in global Cartesian coordinates is used to track the target.

```
% Set random seed for reproducible results
rng(50);

% Create a |trackerGNN| to track the targets using the
% FilterInitializationFcn as @initCartesianEKF.
```

```matlab
tracker = trackerGNN('FilterInitializationFcn',@initCartesianEKF,...
    'AssignmentThreshold',50,...
    'MaxNumTracks',5);

[tem, tam] = helperPassiveRangingErrorMetrics(ownship,false);

theaterDisplay.ErrorMetrics = tem;

tracks = [];

% Advance scenario, simulate detections and track
while advance(scene)
    % Get time information from tracking scenario.
    truths = platformPoses(scene);
    time = scene.SimulationTime;

    % Generate detections from the ownship
    detections = detect(ownship,time);

    % Pass detections to tracker
    if ~isempty(detections)
        tracks = tracker(detections,time);
    elseif isLocked(tracker)
        tracks = predictTracksToTime(tracker,'confirmed',time);
    end
    % Update error and assignment metrics
    tam(tracks,truths);
    [trackIDs, truthIDs] = currentAssignment(tam);
    tem(tracks,trackIDs,truths,truthIDs);

    % Update display
    theaterDisplay(tracks,detections,tracker);
end
```

The track is initialized using a high covariance in range to account for the missing measurement. Notice the covariance in range of the track just after initialization in the theater plot. As angular estimates are fairly accurate, the target uncertainty is described by a thin covariance ellipse.

```matlab
showGrabs(theaterDisplay,1);
```
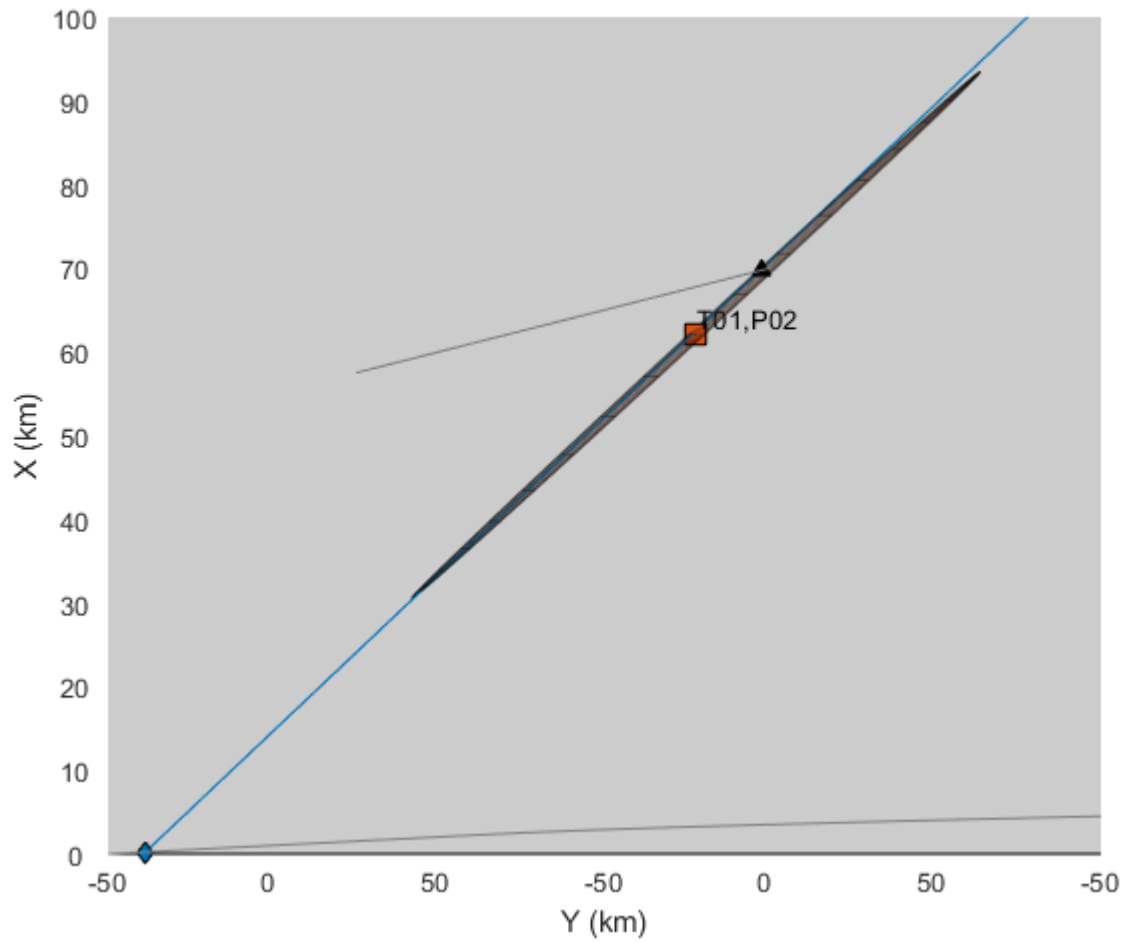
The following figure shows the filter performance after the sensor starts maneuvering in the scenario. The range and range-rate estimate plots are created using `errorbar` and show the σ (standard deviation) bounds of the track's estimate.

```
showGrabs(theaterDisplay,2);
```
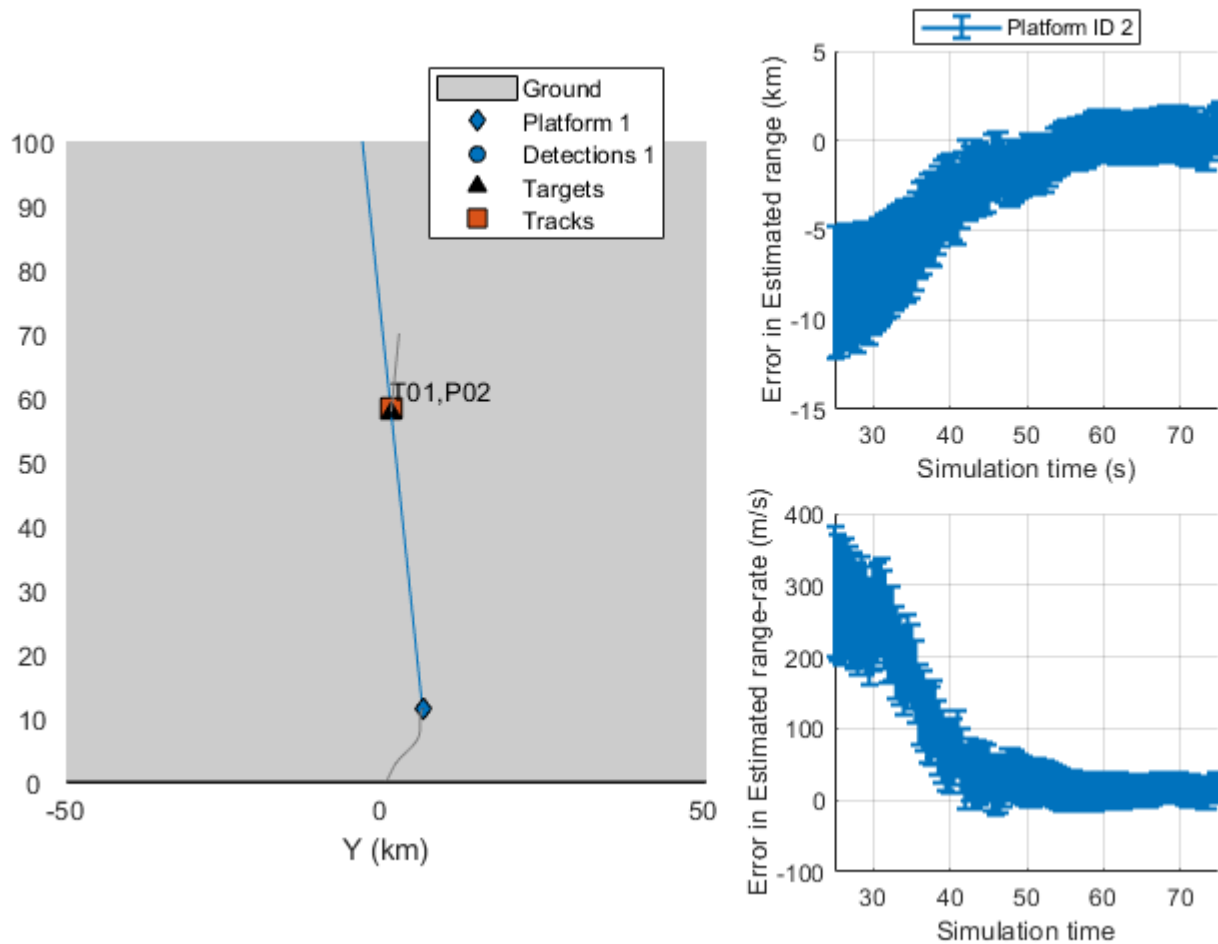
Stability of EKF in Cartesian Coordinates

Tracking in Cartesian coordinates using an extended Kalman filter is appealing due to the ease of problem formulation. The state dynamics are represented by a set of linear equations and the non-linearities are embedded using two (relatively) simple equations for azimuth and elevation.

That said, the behavior of the extended Kalman filter is shown to be erratic and can often be unstable. This is because the states (position and velocity) and unobservable range are highly coupled. When range is unobservable, i.e., during the constant velocity phase of the sensing platform motion, the filter "falsely" estimates the range of the target depending on the history of measurement and associated noise values. This can result in a premature collapse of the covariance ellipse, which can cause the filter to take a very long time to converge (or even diverge) [2] even after sufficient observability conditions are met.

Use a different random seed to observe premature convergence of an extended Kalman filter in Cartesian coordinates.

```
% set random seed
rng(2015);
```

```
% Reset the theaterDisplay to capture snapshots with new scenario.
release(theaterDisplay);
theaterDisplay.GrabFigureFcn = @(fig,scene)helperGrabPassiveRangingDisplay(fig,scene,'CartEKF2')
helperRunPassiveRangingSimulation(scene,theaterDisplay,@initCartesianEKF);
```

After 2 seconds, the covariance in the range has already collapsed, making the filter falsely confident about its estimate of the range. Notice that target lies outside the covariance ellipse of the track's state and so does the zero-error line in the range estimation plot.

```
showGrabs(theaterDisplay,3);
```
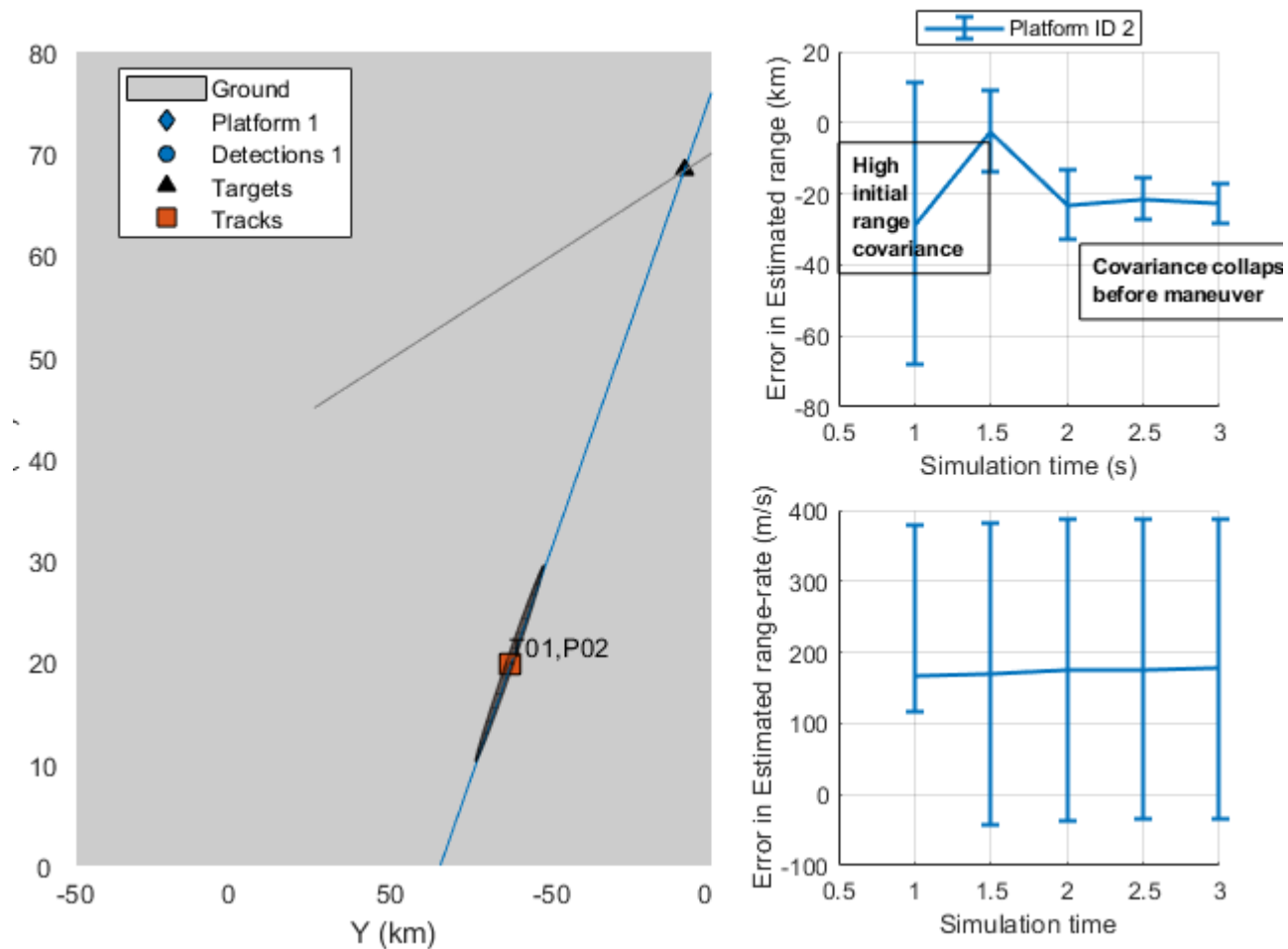

Because the filter remains confident about it estimate of the range, it takes much longer to converge closer to actual range values.

```
showGrabs(theaterDisplay,4);
```
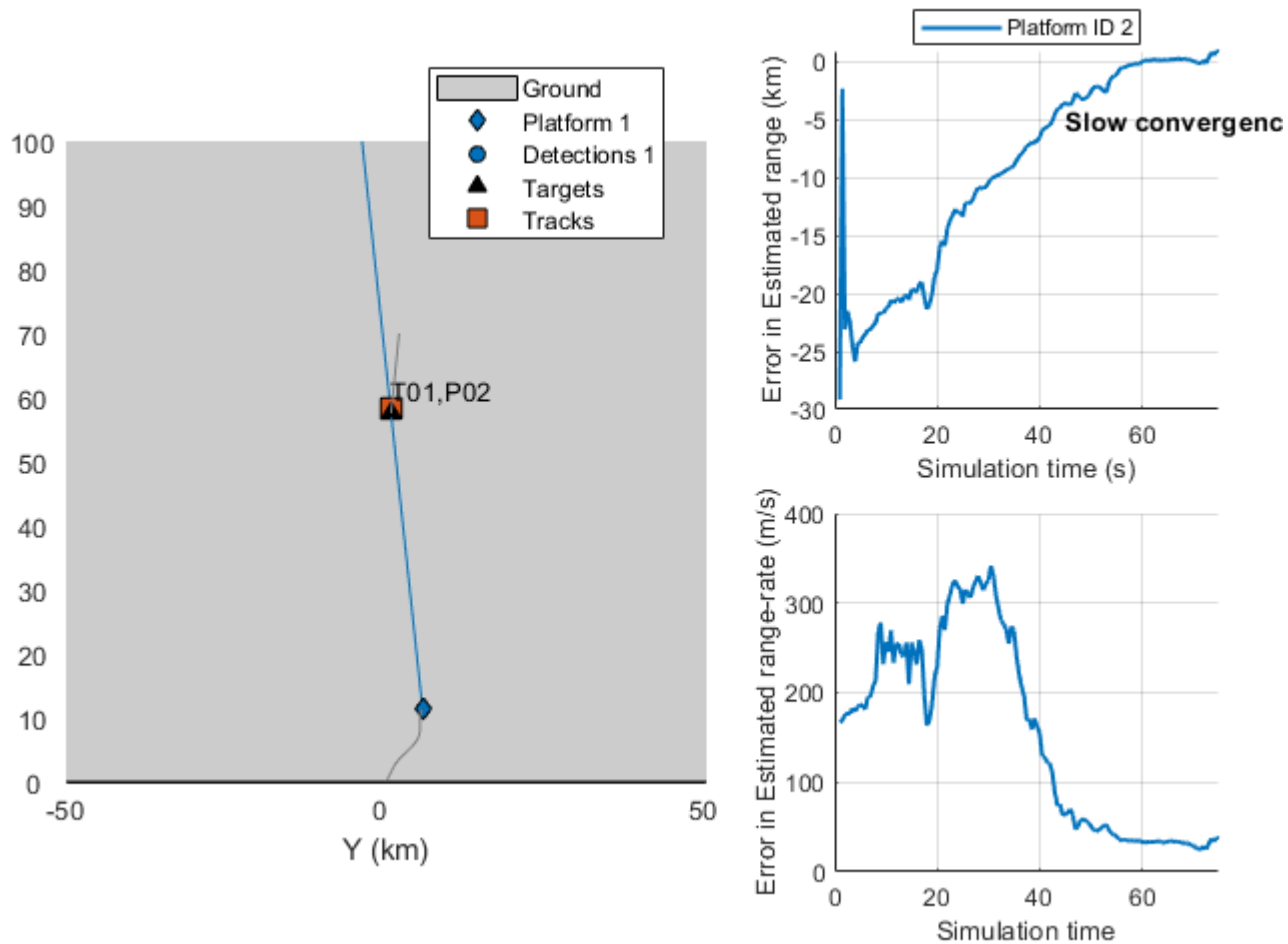
Track Using an EKF in Modified Spherical Coordinates (MSC-EKF)

The modified spherical coordinates (MSC) present a stable coordinate system for tracking using angle-only measurements. By decoupling the state into observable and unobservable parts, the filter overcomes the limitations imposed by the EKF in Cartesian coordinates. The state in modified spherical coordinates is defined in a relative manner i.e. [target - observer] and hence an input from the observer is required to predict the state in future. This can be seen in the following equations where higher order terms refer to observer motion not captured by a constant velocity model.

$$x = x_t - x_o$$

$$\dot{x} = \dot{x}_t - \dot{x}_o$$

$$\dot{x} = Ax_t - (Ax_0 + \mathcal{O}(t^2) + \mathcal{O}(t^3) + ..)$$

$$\dot{x} = A(x_t - x_0) + \text{ higher order terms}$$

```
% Set the same random seed to compare with the same detections
rng(2015);
```

```matlab
% restart the scene
restart(scene);
release(theaterDisplay);
theaterDisplay.GrabFigureFcn = @(fig,scene)helperGrabPassiveRangingDisplay(fig,scene,'MSCEKF');

[tem, tam] = helperPassiveRangingErrorMetrics(ownship,true);
theaterDisplay.ErrorMetrics = tem;

% Create a tracker using |trackerGNN| and MSC-EKF filter initialization.
tracker = trackerGNN('FilterInitializationFcn',@initMSCEKF,...
    'AssignmentThreshold',50);

% The tracks carry a state which is non-linearly dependent on position.
% Inform the theaterDisplay that tracks have non-linear state and position
% can be extracted using a function_handle.
theaterDisplay.HasNonLinearState = true;
theaterDisplay.NonLinearTrackPositionFcn = @getTrackPositionsMSC;

% Initialization for MSC-EKF.
prevPose = pose(ownship,'true');
lastCorrectionTime = 0;
allTracks = [];

% Advance scenario, simulate detections and track
while advance(scene)

    time = scene.SimulationTime;
    truths = platformPoses(scene);

    % Generate detections from ownship
    detections = detect(ownship,time);

    % Update the input from the ownship i.e. it's maneuver since last
    % correction time.
    currentPose = pose(ownship,'true');
    dT = time - lastCorrectionTime;
    observerManeuver = calculateManeuver(currentPose,prevPose,dT);

    for i = 1:numel(allTracks)
        % Set the ObserverInput property using |setTrackFilterProperties|
        % function of the tracker
        setTrackFilterProperties(tracker,allTracks(i).TrackID,'ObserverInput',observerManeuver);
    end

    % Pass detections to tracker
    if ~isempty(detections)
        lastCorrectionTime = time;
        % Store the previous pose to calculate maneuver
        prevPose = currentPose;
        [tracks,~,allTracks] = tracker(detections,time);
    elseif isLocked(tracker)
        tracks = predictTracksToTime(tracker,'confirmed',time);
    end

    % Update error and assignment metrics
    tam(tracks,truths);
    [trackIDs, truthIDs] = currentAssignment(tam);
    tem(tracks,trackIDs,truths,truthIDs);
```

```
        % Update display
        theaterDisplay.NonLinearStateInput = currentPose.Position(:);
        theaterDisplay(detections,tracks,tracker);
end
```

The MSC-EKF tries to maintain the covariance in range till the sensor has not made a maneuver. This is essentially due to decoupled observable and unobservable parts in the state.

```
showGrabs(theaterDisplay,5);
```
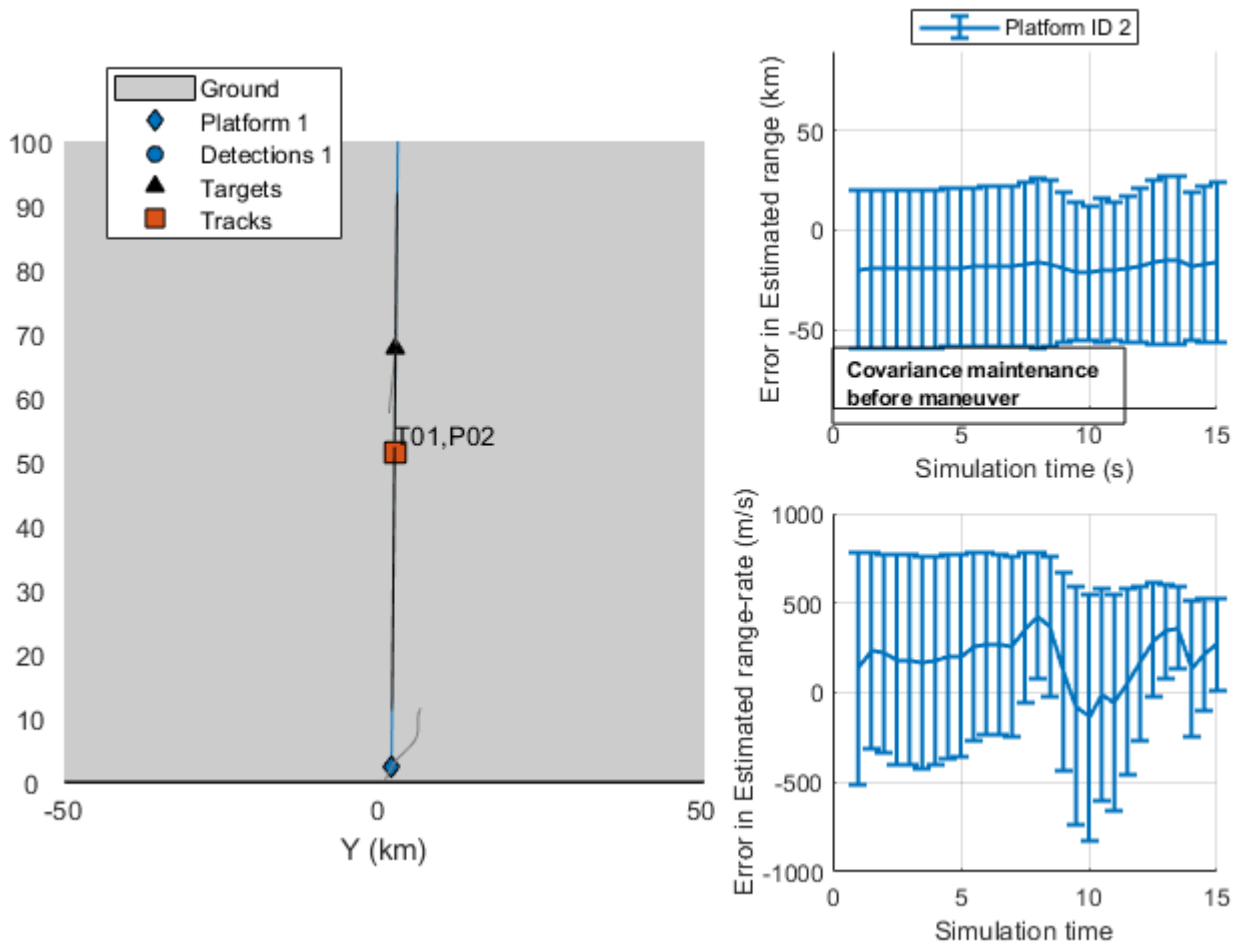

The filter converges closer to true values faster than the EKF in Cartesian coordinates and the σ bounds provide a true, unbiased estimate of the error.

```
showGrabs(theaterDisplay,6);
```
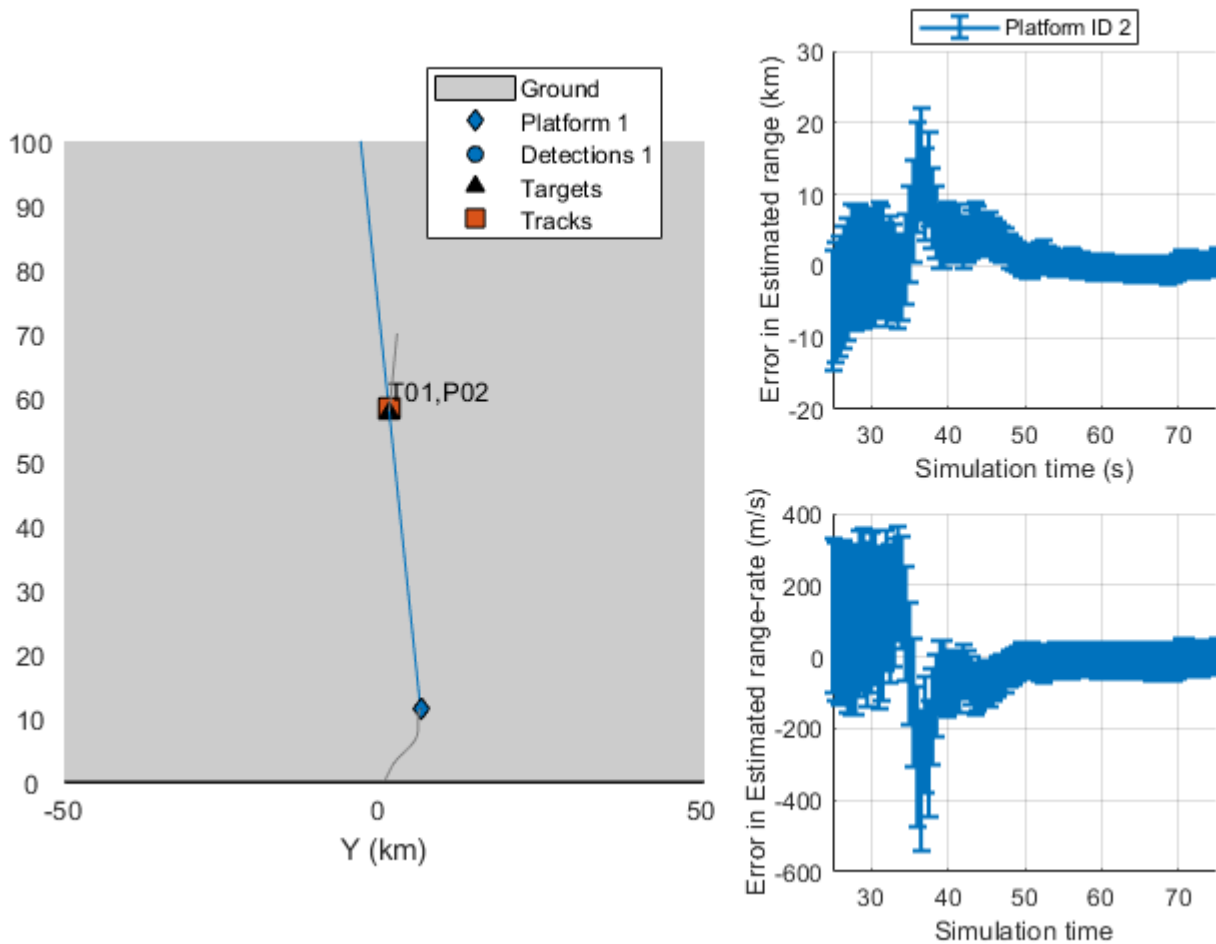
Track Using a Range-Parameterized MSC-EKF

The MSC-EKF approach uses linearization techniques to project covariances in the prediction step. The covariance in range at initialization is typically high and the state transition dynamics is highly non-linear, which can cause issues with filter convergence.

This section demonstrates the use of a Gaussian-sum filter, `trackingGSF`, to describe the state with a bank of filters, each initialized at different range assumptions. This technique is commonly termed as range-parameterization [3].

```
% Set random seed
rng(2015);

% restart the scene
restart(scene);
release(theaterDisplay);
theaterDisplay.GrabFigureFcn = @(fig,scene)helperGrabPassiveRangingDisplay(fig,scene,'MSCRPEKF')

% Create error and assignment metrics.
[tem, tam] = helperPassiveRangingErrorMetrics(ownship,true);
theaterDisplay.ErrorMetrics = tem;
```

```matlab
% Create a tracker using |trackerGNN| and range-parameterized MSC-EKF
% filter initialization.
tracker = trackerGNN('FilterInitializationFcn',@initMSCRPEKF,...
    'AssignmentThreshold',50);

theaterDisplay.HasNonLinearState = true;
theaterDisplay.NonLinearTrackPositionFcn = @getTrackPositionsMSC;

% Initialization for MSC-RPEKF
prevPose = pose(ownship,'true');
lastCorrectionTime = 0;
allTracks = [];

% Advance scenario, simulate detections and track
while advance(scene)

    time = scene.SimulationTime;
    truths = platformPoses(scene);

    % Generate detections from ownship
    detections = detect(ownship,time);

    % Update the input from the ownship i.e. it's maneuver since last
    % correction time.
    currentPose = pose(ownship,'true');
    dT = time - lastCorrectionTime;
    observerManeuver = calculateManeuver(currentPose,prevPose,dT);

    % Get each filter from the trackingGSF property TrackingFilters using
    % the |getTrackFilterProperties| function of the tracker.
    for i = 1:numel(allTracks)
        trackingFilters = getTrackFilterProperties(tracker,allTracks(i).TrackID,'TrackingFilters
        % Set the ObserverInput for each tracking filter
        for m = 1:numel(trackingFilters{1})
            trackingFilters{1}{m}.ObserverInput = observerManeuver;
        end
    end

    % Pass detections to tracker
    if ~isempty(detections)
        lastCorrectionTime = time;
        % Store the previous pose to calculate maneuver
        prevPose = currentPose;
        [tracks,~,allTracks] = tracker(detections,time);
    elseif isLocked(tracker)
        tracks = predictTracksToTime(tracker,'confirmed',time);
    end

    % Update error and assignment metrics
    tam(tracks,truths);
    [trackIDs, truthIDs] = currentAssignment(tam);
    tem(tracks,trackIDs,truths,truthIDs);

    % Update display
    theaterDisplay.NonLinearStateInput = currentPose.Position(:);
    theaterDisplay(detections,tracks,tracker);
end
```

The following figures show the range-parameterized filter after track is initialized and the tracking performance of the filter. The range-parameterization process allows each filter to carry a relatively small covariance in range and hence is less susceptible to linearization issues. The time-to-converge for range-parametrized filter in this scenario is similar to the MSC-EKF. However, the filter demonstrated less transient behavior as the sensor moves into the second major maneuvering phase of the trajectory, i.e, 35 to 40 seconds into simulation. Notice, the MSC-EKF range estimation plot above, which produced an error of 10+ km in range. The estimation error was about 5 km for the range-parametrized filter.

```
showGrabs(theaterDisplay,[7 8]);
```

Range-Parameterized Representation of State
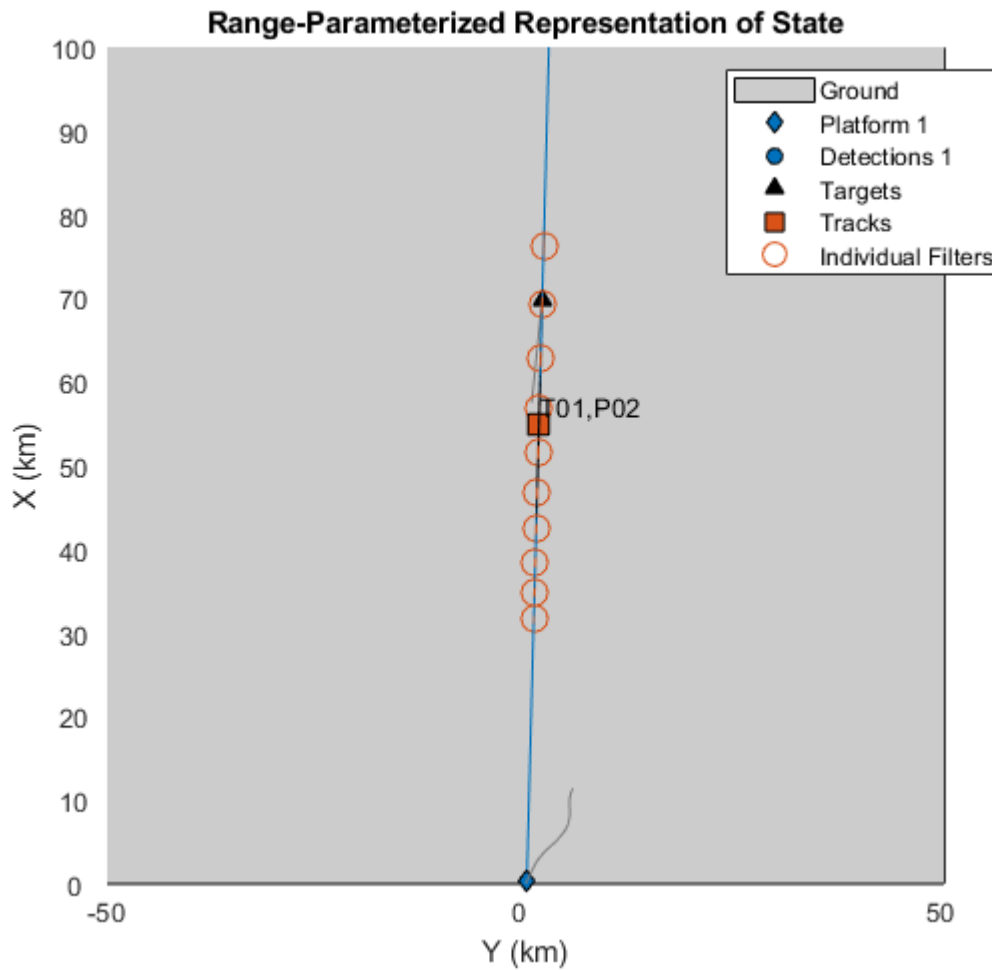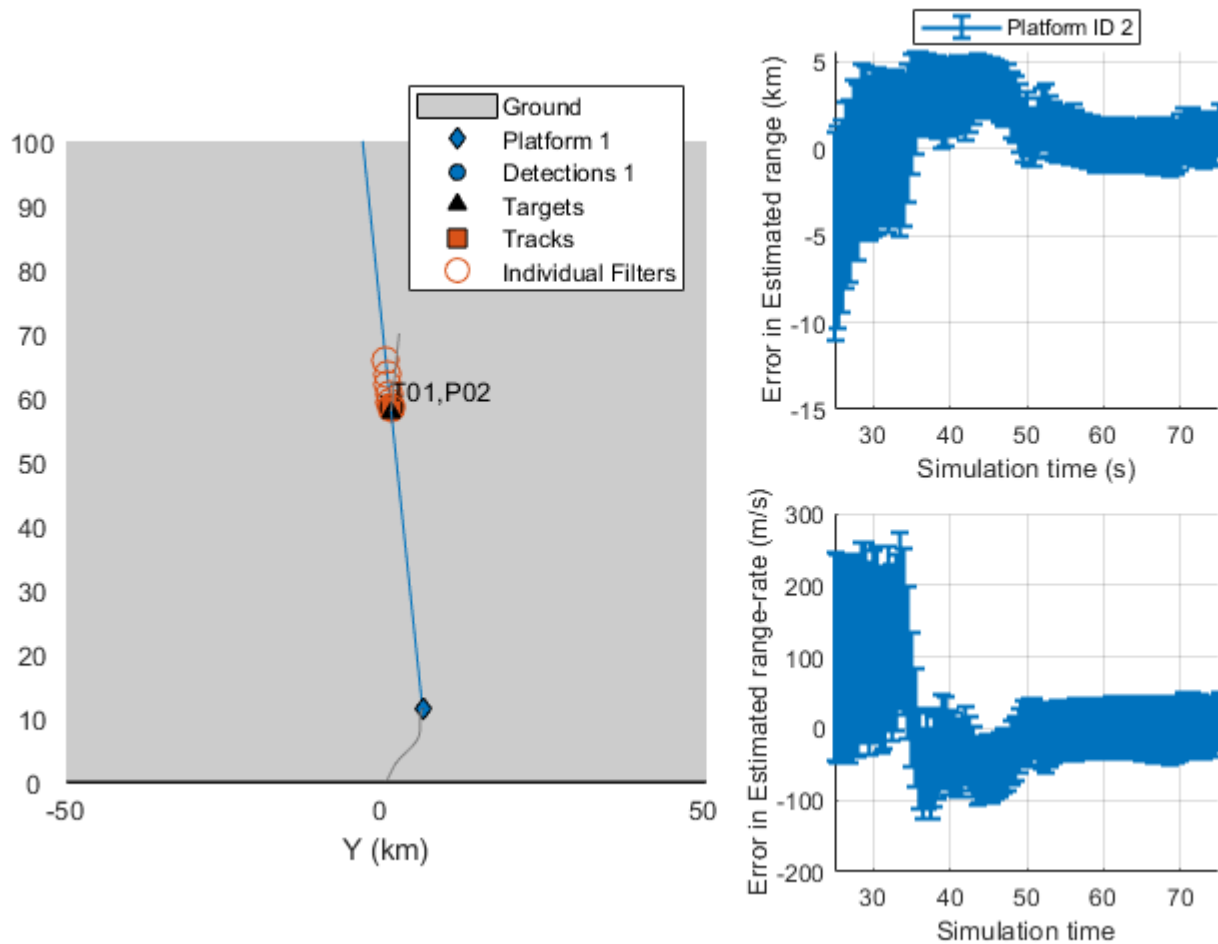
Close displays

```
showGrabs(theaterDisplay,[]);
```

Multi-Target Scenarios

The demonstrated approach using MSC-EKF and range-parameterized MSC-EKF is applicable for more than 1 target. To observe each target, the sensor must out-maneuver each one of them. As filters like MSC-EKF can maintain range-covariance during non-maneuvering stages, each track's estimate should converge closer to the targets as the sensor makes maneuvers with respect to them.

MSC-EKF

```
% Set random seed
rng(2015);

% Create a two-target scenario and theater display.
[sceneTwo,~,theaterDisplayTwo] = helperCreatePassiveRangingScenario(2);

% Use the helper function to run the simulation using @initMSCEKF.
helperRunPassiveRangingSimulation(sceneTwo,theaterDisplayTwo,@initMSCEKF);
```

```
% Show results
showGrabs(theaterDisplayTwo,1);
```
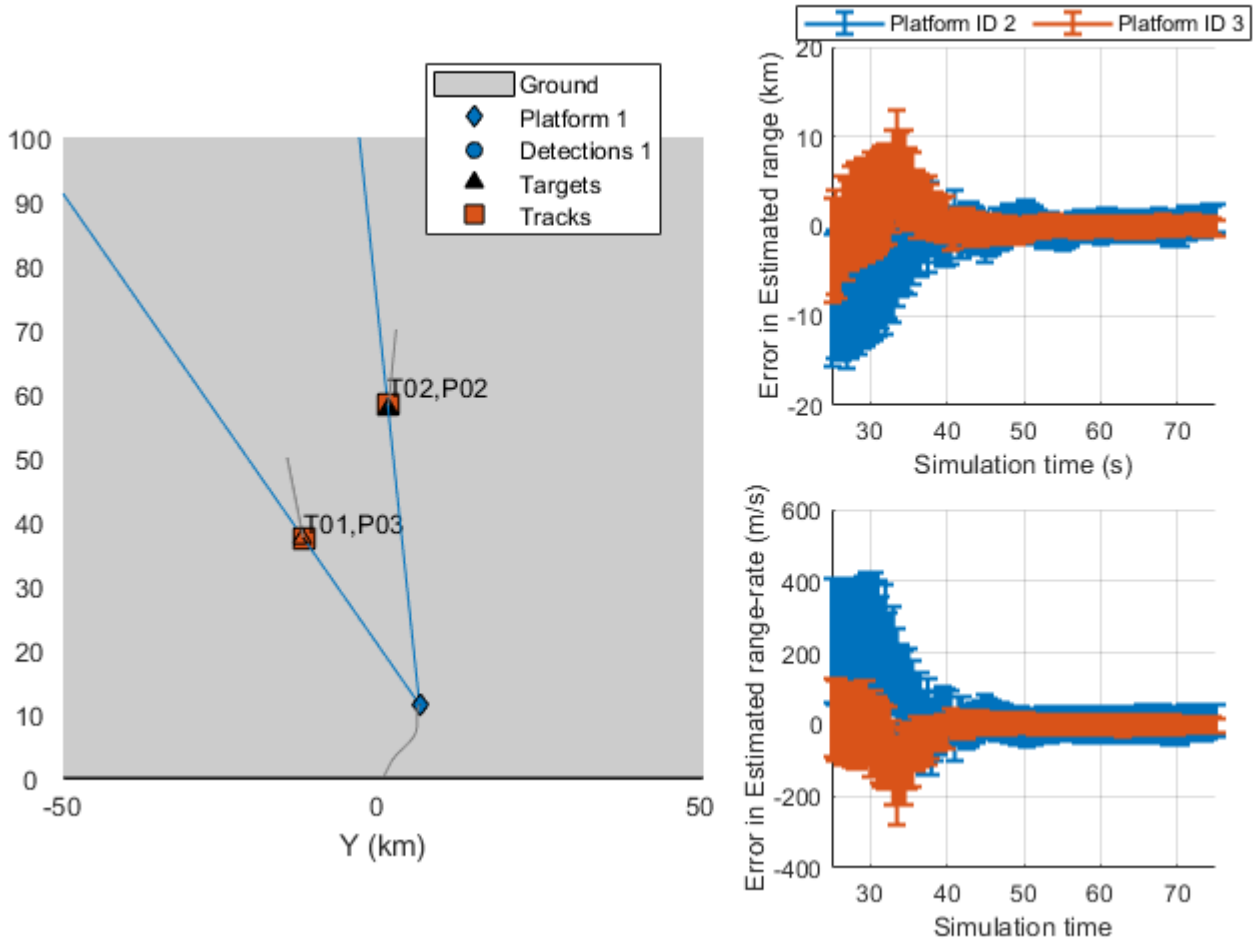

Range-parameterized MSC-EKF

```
% Set random seed
rng(2015);
release(theaterDisplayTwo);

% Run the simulation using range-parameterized MSC-EKF
helperRunPassiveRangingSimulation(sceneTwo,theaterDisplayTwo,@initMSCRPEKF);

% Show results
showGrabs(theaterDisplayTwo,2);

rmpath(exPath);
```
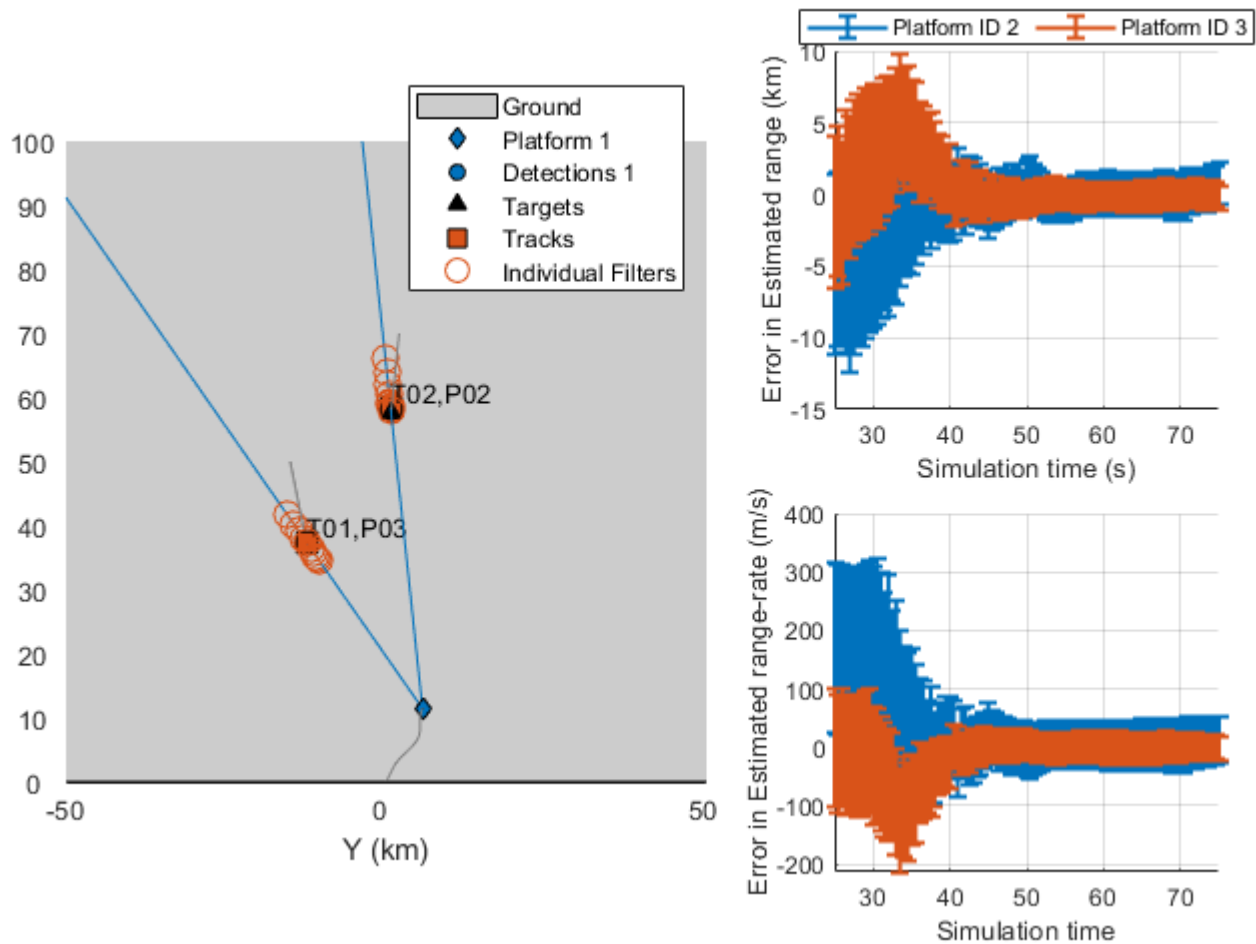
Summary

This example illustrates the challenges associated with single-sensor angle-only tracking problem and demonstrates how to use different tracking algorithms to estimate target's range and range-rate. You learned how to use `initcvekf`, `initcvmscekf` and `trackingGSF` to define customized Cartesian-EKF, MSC-EKF and range-parameterized MSC-EKF for passive ranging. You also learned how to use the defined filters with a GNN tracker, `trackerGNN`.

Supporting Functions

Filter initialization Functions

The following section lists all the `FilterInitializationFcn` used for the `trackerGNN` object in this Example.

initCartesianEKF Initialize trackingEKF from an angle-only detection

```
function filter = initCartesianEKF(detection)

% Create a full detection with high covariance in range estimate.
rangeEstimate = 5e4;
```

```
rangeCov = 16e8;
fullDetection = detection;
fullDetection.Measurement = [fullDetection.Measurement;rangeEstimate];
fullDetection.MeasurementNoise = blkdiag(fullDetection.MeasurementNoise,rangeCov);

% Update the MeasurementParameters to include range.
fullDetection.MeasurementParameters(1).HasRange = true;

% Use the initcvekf function to initialize a trackingEKF using the
% fullDetection.
fullFilter = initcvekf(fullDetection);

% |initcvekf| defines the StateCovariance in velocity with 100. This
% defines a standard deviation uncertainty in velocity as 10 m/s. Scale
% the velocity covariance with 400 i.e. an equivalent velocity standard
% deviation of 200 m/s
velCov = fullFilter.StateCovariance(2:2:end,2:2:end);
fullFilter.StateCovariance(2:2:end,2:2:end) = 400*velCov;

% fullFilter can only be corrected with [az el r] measurements.
% Create a |trackingEKF| using the State and StateCovariance from
% fullFilter.
filter = trackingEKF(@constvel,@cvmeas,fullFilter.State,...
    'StateCovariance',fullFilter.StateCovariance,...
    'StateTransitionJacobianFcn',@constveljac,...
    'MeasurementJacobianFcn',@cvmeasjac,...
    'HasAdditiveProcessNoise',false);

% Unit standard deviation acceleration noise
filter.ProcessNoise = eye(3);

end
```

initMSCEKF Initialize a MSC-EKF from an angle-only detection

```
function filter = initMSCEKF(detection)
% Use the second input of the |initcvmscekf| function to provide an
% estimate of range and standard deviation in range.
rangeEstimate = 5e4;
rangeSigma = 4e4;
filter = initcvmscekf(detection,[rangeEstimate,rangeSigma]);
% The initcvmscekf assumes a velocity standard deviation of 10 m/s, which
% is linearly transformed into azimuth rate, elevation rate and vr/r.
% Scale the velocity covariance by 400 to specify that target can move 20
% times faster.
filter.StateCovariance(2:2:end,2:2:end) = 400*filter.StateCovariance(2:2:end,2:2:end);
filter.ProcessNoise = eye(3);
end
```

initMSCRPEKF Initialize a range-parameterized MSC-EKF from an angle-only detection

```
function filter = initMSCRPEKF(detection)
% A range-parameterized MSC-EKF can be defined using a Gaussian-sum filter
% (trackingGSF) containing multiple |trackingMSCEKF| as TrackingFilters.

% Range-parametrization constants
rMin = 3e4;
rMax = 8e4;
```

5-271

```
numFilters = 10;
rho = (rMax/rMin)^(1/numFilters);
Cr = 2*(rho - 1)/(rho + 1)/sqrt(12);
indFilters = cell(numFilters,1);
for i = 1:numFilters
    range = rMin/2*(rho^i + rho^(i-1));
    rangeSigma = Cr*range;
    % Use initcvmscekf function to create a trackingMSCEKF with provided
    % range and rangeSigma.
    indFilters{i} = initcvmscekf(detection,[range rangeSigma]);
    % Update the velocity covariance of each filter.
    indFilters{i}.StateCovariance(2:2:end,2:2:end) = 400*indFilters{i}.StateCovariance(2:2:end,2
end
filter = trackingGSF(indFilters);

end
```

Utility functions `calculateManeuver`

```
function maneuver = calculateManeuver(currentPose,prevPose,dT)
% Calculate maneuver i.e. 1st order+ motion of the observer. This is
% typically obtained using sensors operating at much higher rate.
v = prevPose.Velocity;
prevPos = prevPose.Position;
prevVel = prevPose.Velocity;
currentPos = currentPose.Position;
currentVel = currentPose.Velocity;

% position change apart from constant velocity motion
deltaP = currentPos - prevPos - v*dT;
% Velocity change
deltaV = currentVel - prevVel;

maneuver = zeros(6,1);
maneuver(1:2:end) = deltaP;
maneuver(2:2:end) = deltaV;
end
```

getTrackPositionsMSC

```
function [pos,cov] = getTrackPositionsMSC(tracks,observerPosition)

if isstruct(tracks) || isa(tracks,'objectTrack')
    % Track struct
    state = [tracks.State];
    stateCov = cat(3,tracks.StateCovariance);
elseif isa(tracks,'trackingMSCEKF')
    % Tracking Filter
    state = tracks.State;
    stateCov = tracks.StateCovariance;
end

% Get relative position using measurement function.
relPos = cvmeasmsc(state,'rectangular');

% Add observer position
pos = relPos + observerPosition;
pos = pos';
```

```matlab
if nargout > 1
    cov = zeros(3,3,numel(tracks));

    for i = 1:numel(tracks)
        % Jacobian of position measurement
        jac = cvmeasmscjac(state(:,i),'rectangular');
        cov(:,:,i) = jac*stateCov(:,:,i)*jac';
    end
end

end
```

References

[1] Fogel, Eli, and Motti Gavish. "Nth-order dynamics target observability from angle measurements." IEEE Transactions on Aerospace and Electronic Systems 24.3 (1988): 305-308.

[2] Aidala, Vincent, and Sherry Hammel. "Utilization of modified polar coordinates for bearings-only tracking." IEEE Transactions on Automatic Control 28.3 (1983): 283-294.

[3] Peach, N. "Bearings-only tracking using a set of range-parameterised extended Kalman filters." IEE Proceedings-Control Theory and Applications 142.1 (1995): 73-80.

Benchmark Trajectories for Multi-Object Tracking

This example shows how to generate and visualize trajectories of multiple aircraft using `trackingScenario` and `waypointTrajectory`.

Introduction

The six aircraft trajectories modeled in this example are described in [1]. The aircraft fly in an arrangement intended to be received by a radar located at the origin.

Choice of Interpolant

Conceptually speaking, a trajectory is a curve through space which an object travels as a function of time. To define the curve, you may think of a curve through space that passes through a set of points called *waypoints* connected by an interpolating function called an *interpolant*. An interpolant allows you to define the path between waypoints via a continuous function. Common interpolants are polynomial based (for example, piecewise linear or cubic splines). For a rapidly changing trajectory, more waypoints are required to keep the interpolated curve as close to the true curve as possible; however, we can reduce the number of required points by choosing interpolants carefully.

Many motion models used in track filters consist of "constant velocity," "constant turn," or "constant acceleration" profiles. To accommodate these motion models, the interpolant used in the `waypointTrajectory` object is based on a piecewise clothoid spline (horizontally) and a cubic spline (vertically). The curvature of a clothoid spline varies linearly with respect to distance traveled; this lets us model straight and constant turns with ease, having one extra degree of freedom to transition smoothly between straight and curved segments. Similarly, objects in the air experience the effects of gravity, following a parabolic (quadratic) path. Having a cubic spline to model vertical elevation allows us to model the path with a similar extra degree of freedom.

Once the physical path through space of an object is known (and set), the speed of the object as a function of distance traveled is determined via cubic Hermite interpolation. This is useful for modeling trajectories of objects that accelerate through turns or straight segments.

The benchmark trajectories we are using consist of straight, constant-g turns, and turns with acceleration.

Waypoint Construction

The following file contains tables of waypoints and velocities (in units of meters and meters per second) that can be used to reconstruct six aircraft trajectories. Load it into MATLAB and examine the table containing the first trajectory.

```
load('benchmarkTrajectoryTables.mat', 'trajTable');
trajTable{1}
```

```
ans =

  14x3 table

    Time          Waypoints                        Velocities
    _____    _____        _____

      0      72947    29474    -1258        -258.9    -129.69         0
     60      57413    21695    -1258        -258.9    -129.66         0
```

62	56905	21417	-1258	-245.3	-153.89	0
78.1	54591	17566	-1258	-20.635	-288.86	0
80	54573	17016	-1258	-2.8042	-289.59	0
83	54571	16147	-1258	-0.061	-289.56	0
110	54571	8329	-1258	0	-289.56	0
112.7	54634	7551.5	-1258	58.979	-283.56	0
120	55718	5785.5	-1258	226.41	-180.59	0
129	58170	5172.8	-1258	284.74	52.88	0
132	59004	5413.9	-1258	274.26	93.05	0
137.8	60592	5962.2	-1258	273.62	94.76	0
147.8	63328	6909.9	-1258	273.62	94.76	0
185	73508	10435	-1258	273.62	94.76	0

Scenario Generation

The table contains a set of waypoints and velocities that the aircraft passes through at the corresponding time.

To use the control points, you can create a scenario with six platforms and assign a trajectory to each:

```
scene = trackingScenario('UpdateRate',10);

for n=1:6
    plat = platform(scene);
    traj = trajTable{n};
    plat.Trajectory = waypointTrajectory(traj.Waypoints, traj.Time, 'Velocities', traj.Velocities
end
```

Trajectory Visualization

Once you have the scenario and plotter set up, you can set up a `theaterPlot` to create an animated view of the locations of the aircraft as time progresses.

```
helperPlot = helperBenchmarkPlotter(numel(scene.Platforms));

while advance(scene)
    % extract the pose of each of the six aircraft
    poses = platformPoses(scene);

    % update the plot
    update(helperPlot, poses, scene.SimulationTime);
end
```
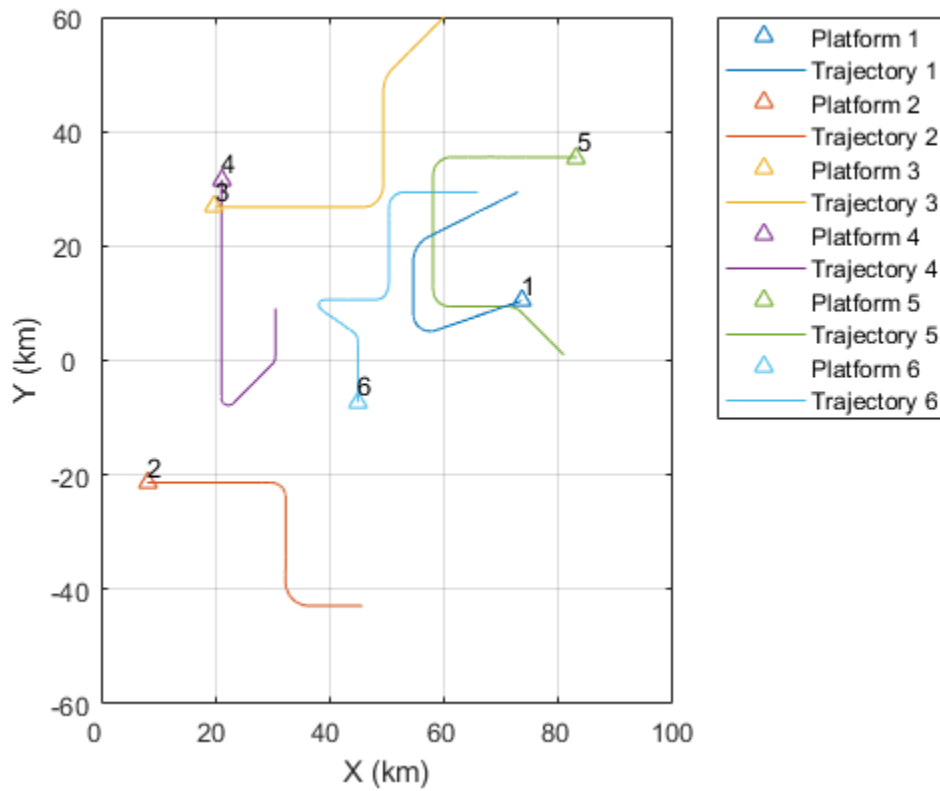
The trajectories plotted above are three-dimensional. You can rotate the plot so that the elevation of the trajectories is readily visible. You can use the `view` and `axis` commands to adjust the plot. Because the trajectories use a NED (north-east-down) coordinate system, elevation above ground has a negative z component.

```
view(60,10);
axis square
grid minor
set(gca,'ZDir','reverse');
```
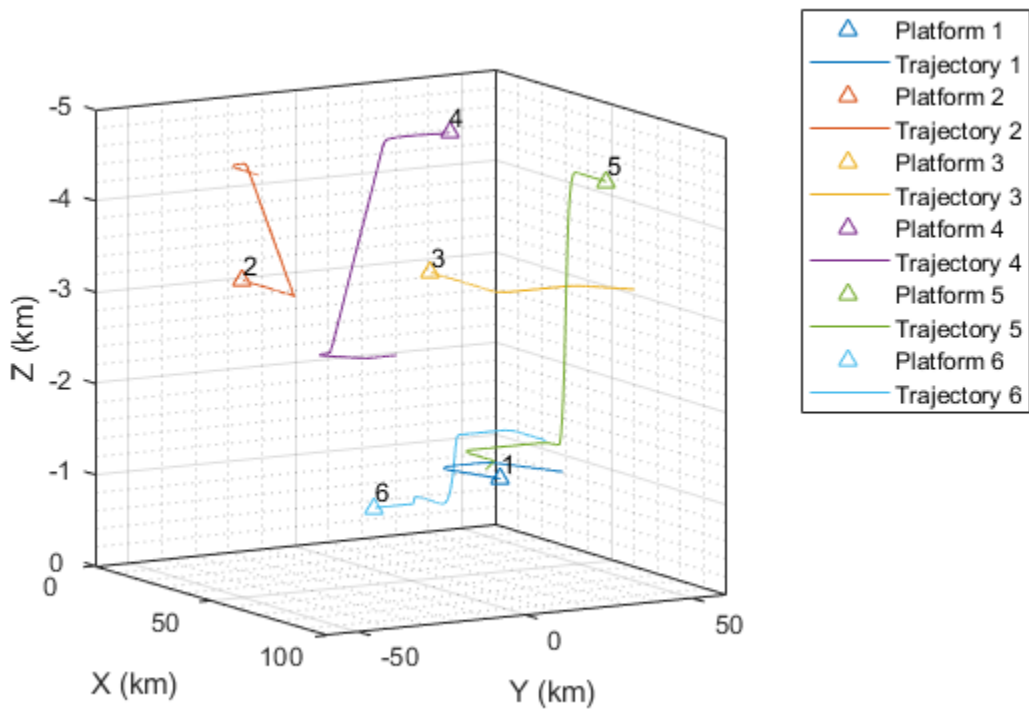
Trajectory 1

It may be instructive to view the control points used to generate the trajectories. The following figure shows the first trajectory, which is representative of a large aircraft.

The control points used to construct the path are plotted on the leftmost plot. Only a few waypoints are needed to mark the changes in curvature as the plane takes a constant turn.

The plots on the right show the altitude, magnitude of velocity (speed), and magnitude of acceleration, respectively. The speed stays nearly constant throughout despite the abrupt change in curvature. This is an advantage of using the clothoid interpolant.

```
[time, position, velocity, acceleration] = cumulativeHistory(helperPlot);
helperTrajectoryViewer(1, time, position, velocity, acceleration, trajTable);
```
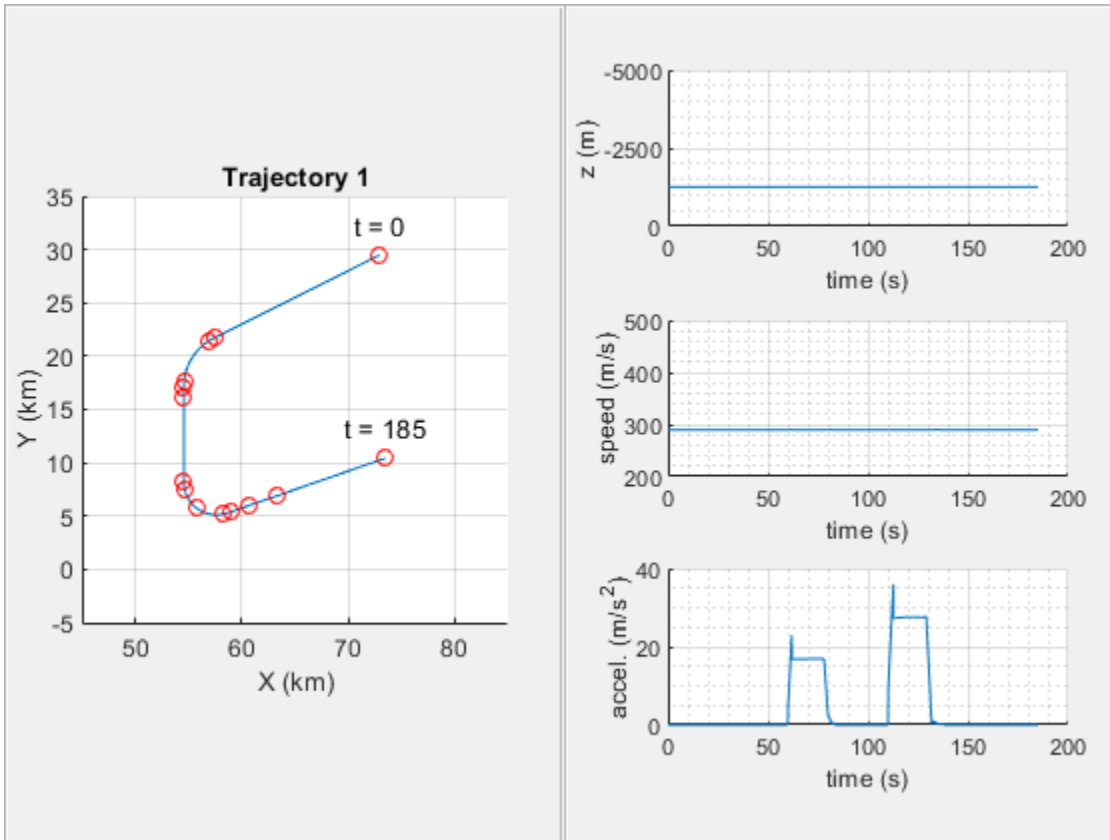
Trajectory 2

The second trajectory, shown below, represents the trajectory of a small maneuverable aircraft. It consists of two turns, having several changes in acceleration immediately after the first turn and during the second turn. More waypoints are needed to adjust for these changes, however the rest of the trajectory requires fewer points.

```
helperTrajectoryViewer(2, time, position, velocity, acceleration, trajTable);
```
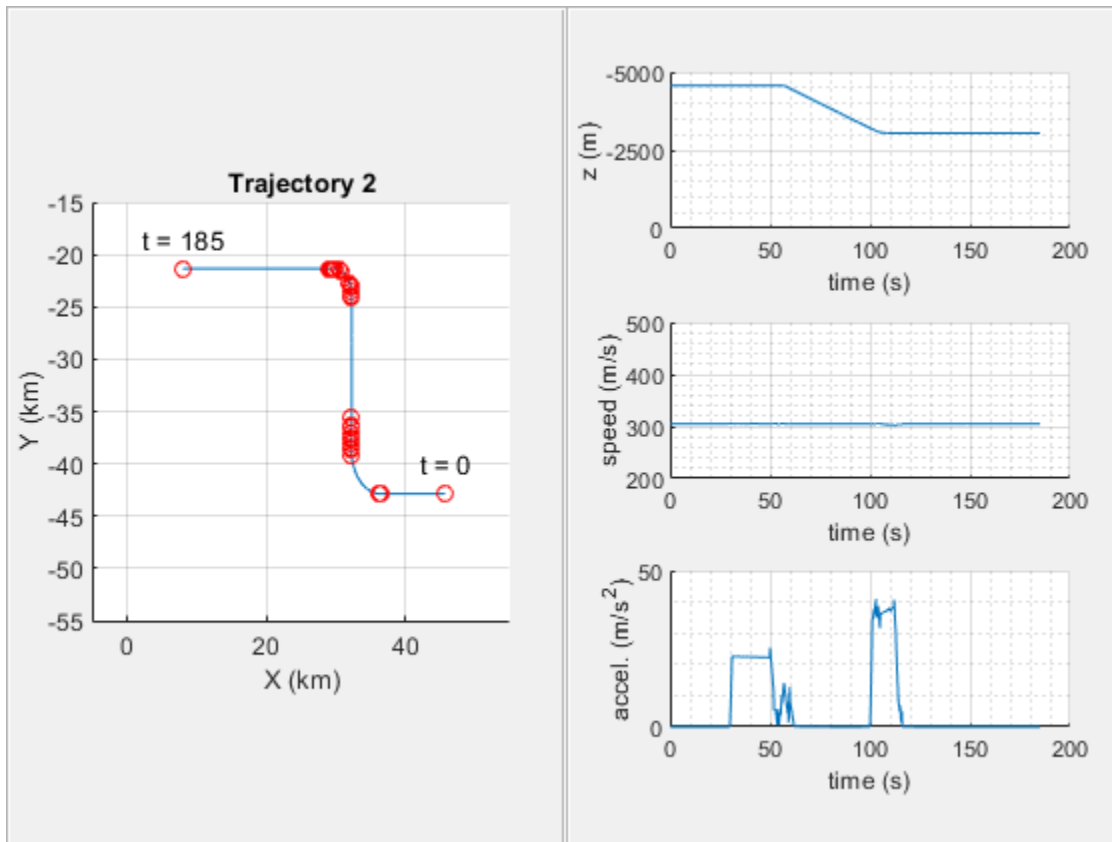
Trajectory 3

The third trajectory, shown below is representative of a higher speed aircraft. It consists of two constant turns, where the aircraft decelerates midway throughout the second turn. You can see the control points that were used to mark the changes in velocity and acceleration in the x-y plot on the left.

```
helperTrajectoryViewer(3, time, position, velocity, acceleration, trajTable);
```
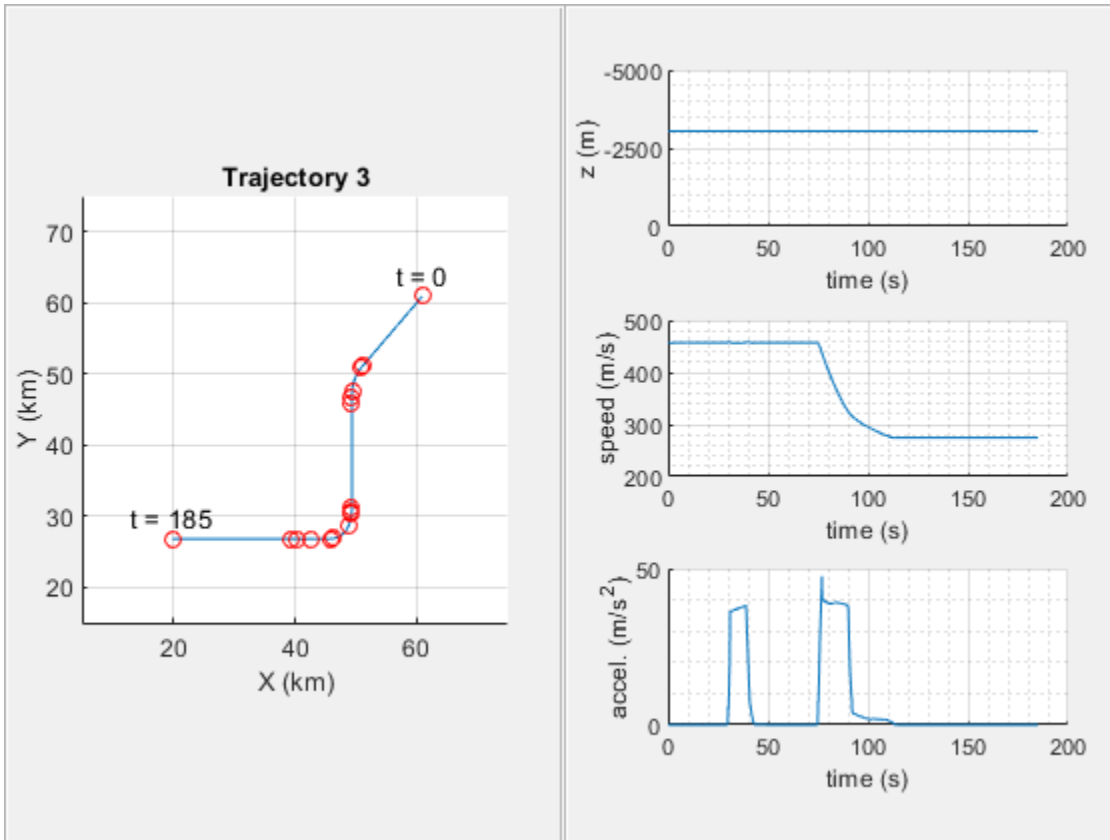
Trajectory 4

The fourth trajectory, also representative of a higher speed aircraft, is shown below. It consists of two turns, where the aircraft accelerates and climbs to a higher altitude.

```
helperTrajectoryViewer(4, time, position, velocity, acceleration, trajTable);
```
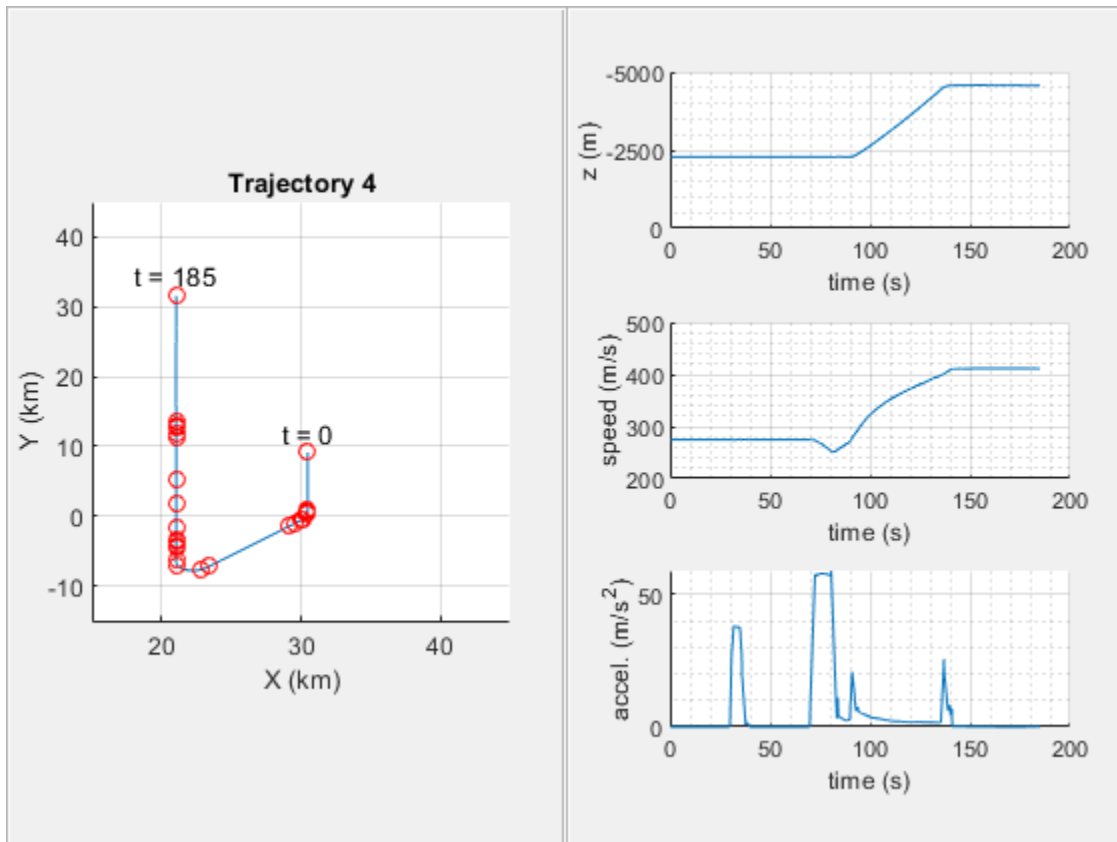
Trajectory 5

The fifth trajectory is representative of a maneuverable high-speed aircraft. It consists of three constant turns; however it accelerates considerably throughout the duration of the flight. After the third turn the aircraft ascends to a level flight.

```
helperTrajectoryViewer(5, time, position, velocity, acceleration, trajTable);
```
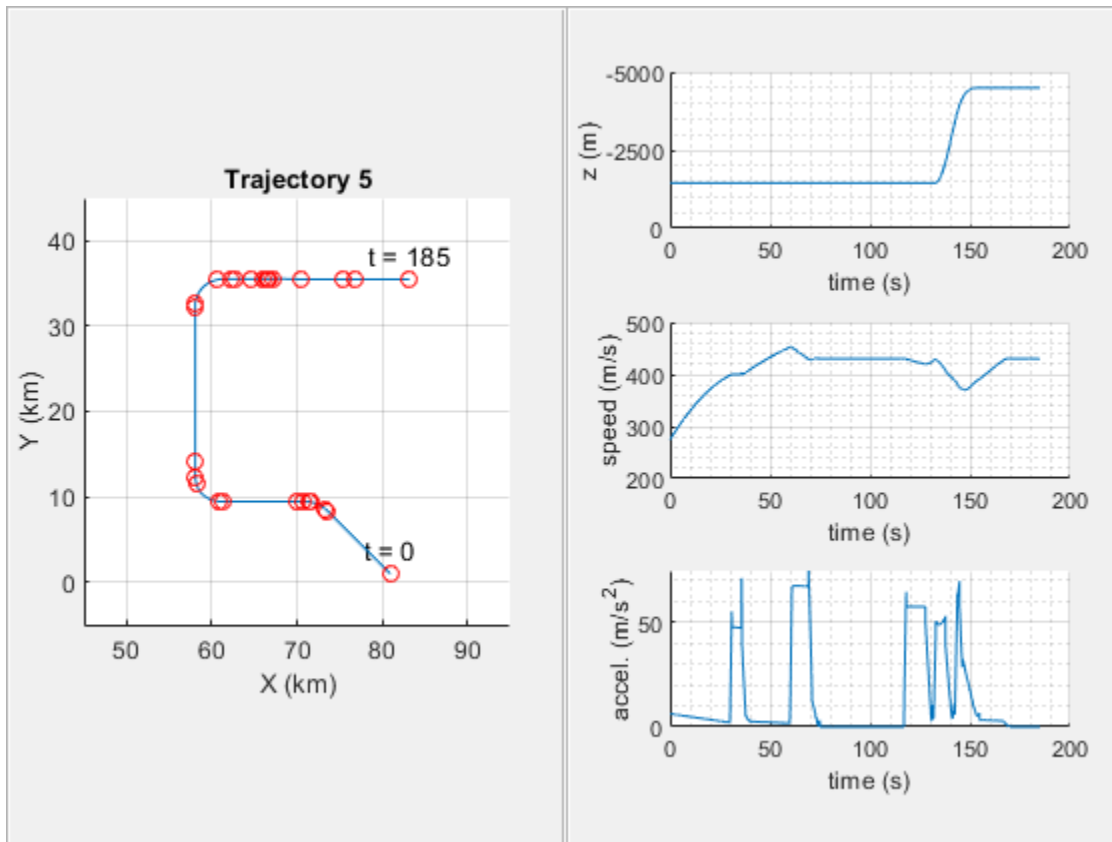
Trajectory 6

The sixth trajectory is also representative of a maneuverable high-speed aircraft. It consists of four turns. After the second turn the aircraft decreases altitude and speed and enters the third turn. After the third turn it accelerates rapidly and enters the fourth turn, continuing with straight and level flight.

```
helperTrajectoryViewer(6, time, position, velocity, acceleration, trajTable);
```
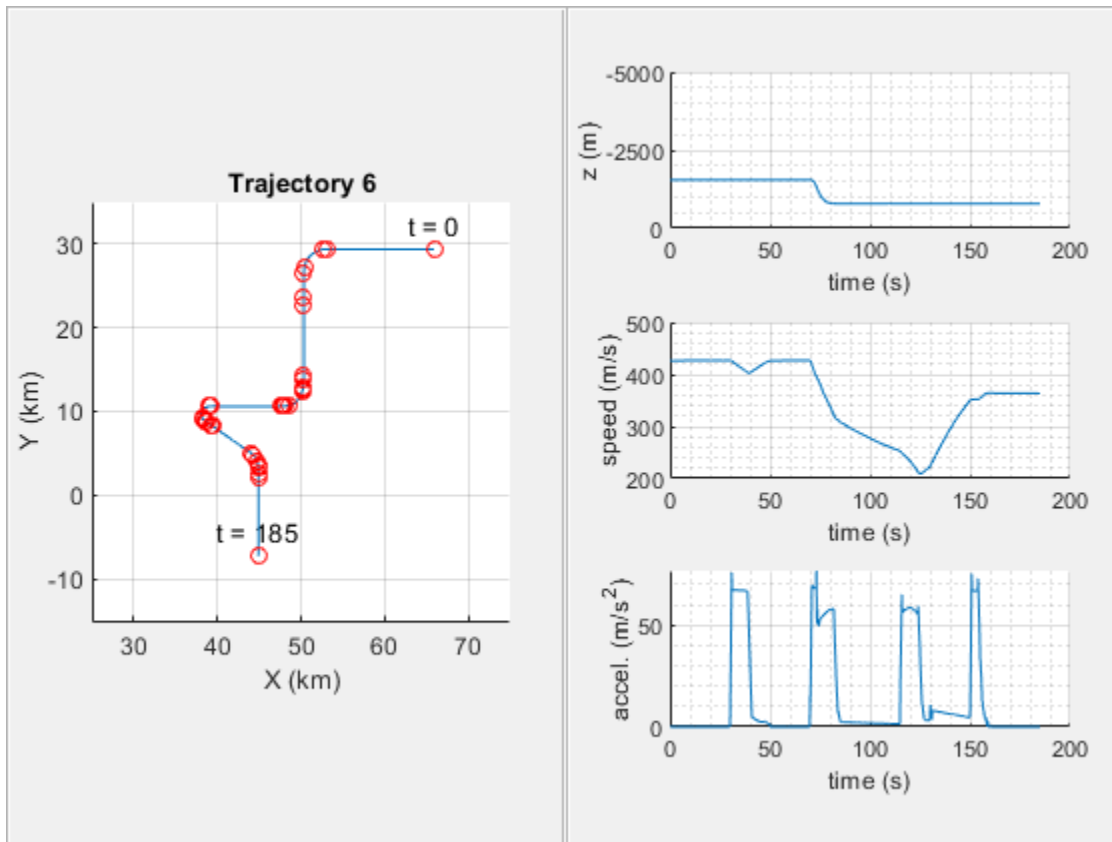
Summary

This example shows how to use `waypointTrajectory` and `trackingScenario` to create a multi-object tracking scenario. In this example you learned the concepts behind the interpolant used inside `waypointTrajectory` and were shown how a scenario could be reproduced with a small number of waypoints.

Reference

1   W.D. Blair, G. A. Watson, T. Kirubarajan, Y. Bar-Shalom, "Benchmark for Radar Allocation and Tracking in ECM." Aerospace and Electronic Systems IEEE Trans on, vol. 34. no. 4. 1998

Tracking with Range-Only Measurements

This example illustrates the use of particle filters and Gaussian-sum filters to track a single object using range-only measurements.

Introduction

Sensors that can only observe range information cannot provide a complete understanding of the object's state from a single detection. In addition, the uncertainty of a range-only measurement, when represented in a Cartesian coordinate frame, is non-Gaussian and creates a concave shape. For range-only sensors with a narrow field of view (FOV), the angular uncertainty is small and can be approximated by an ellipsoid, that is, a Gaussian distribution. However, for range-only sensors with a wide FOV, the uncertainty, when represented in a Cartesian frame, is described by a concave ring shape, which cannot be easily approximated by an ellipsoid. This is illustrated in the image shown below.
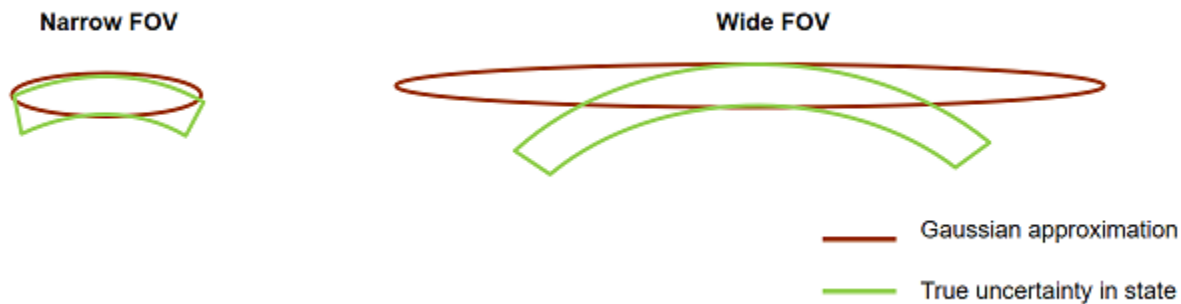
This phenomenon is also observed with long-range radars, which provide both azimuth and range information. When the FOV of each azimuth resolution cell spans a large region in space, the distribution of state becomes non-Gaussian. Techniques like long-range corrections are often employed to make use of Gaussian filters like extended Kalman filters (`trackingEKF`) and Unscented Kalman filters (`trackingUKF`) in those scenarios. This is demonstrated in detail in the "Multiplatform Radar Detection Fusion" on page 5-202 example.

In this example, you will learn how to use a particle filter and a Gaussian-sum filter to represent the non-Gaussian uncertainty in state caused by range measurements from large FOV sensors.

Define Scenario

The scenario models a single object traveling at a constant velocity in the X-Y plane. The object crosses through the coverage areas of three equally spaced sensors, with FOV of 60 degrees in azimuth. The sensors FOV overlap, which enhances observability when the object crosses through the overlapping regions. Each sensor reports range measurement with a measurement accuracy of 5 centimeters.

```
s = rng;
rng(2018);
exPath = fullfile(matlabroot,'examples','fusion','main');
addpath(exPath);
[scene,theaterDisplay] = helperCreateRangeOnlySensorScenario;
showScenario(theaterDisplay)
```

Range-only Tracking Scenario
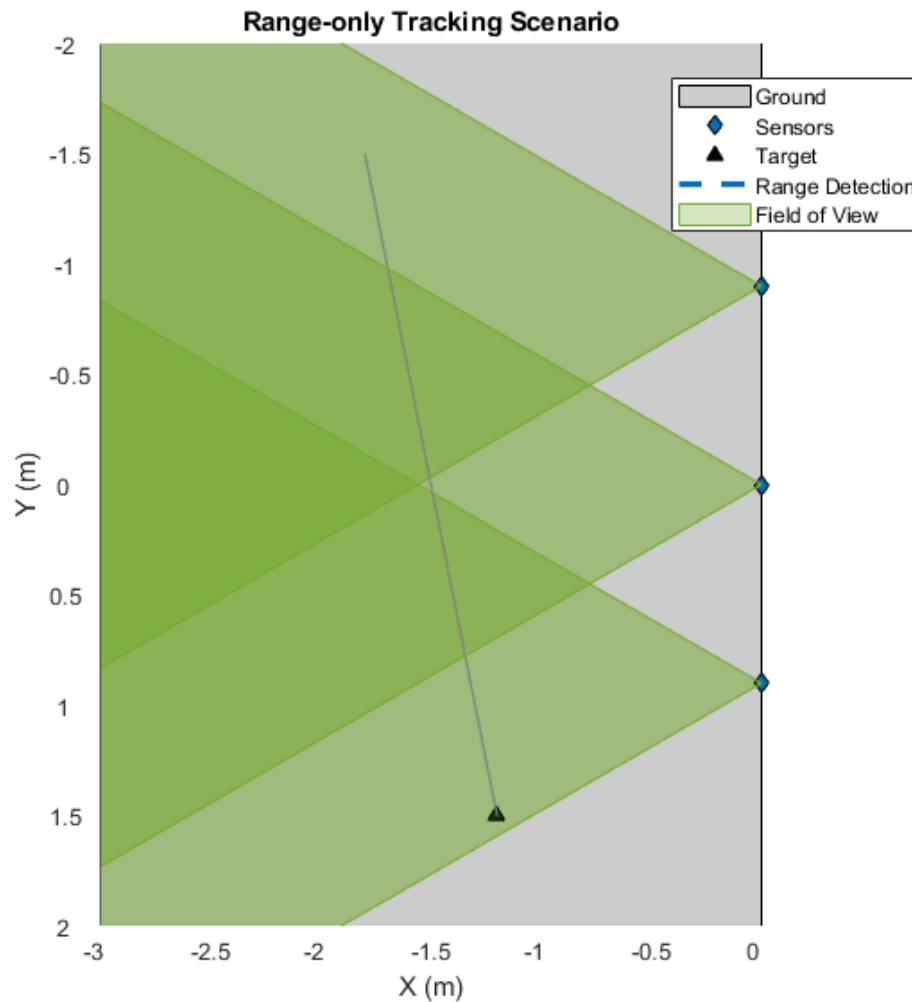
```
showGrabs(theaterDisplay,[]);
```

Track Using a Particle Filter

In this section, a particle filter, `trackingPF` is used to track the object. The tracking is performed by a single-hypothesis tracker using `trackerGNN`. The tracker is responsible for maintaining the track while reducing the number of false alarms.

The function `initRangeOnlyCVPF` initializes a constant velocity particle filter. The function is similar to the `initcvpf` function, but limits the angles from [-180 180] in azimuth and [-90 90] in elevation for `initcvpf` to the known sensor FOV in this problem.

```
% Tracker
tracker = trackerGNN('FilterInitializationFcn',@initRangeOnlyCVPF,'MaxNumTracks',5);

% Update display to plot particle data
theaterDisplay.FilterType = 'trackingPF';
```

```
% Advance scenario and track object
while advance(scene)
    % Current time
    time = scene.SimulationTime;

    % Generate detections
    [detections, configs] = generateRangeDetections(scene);

    % Pass detections to tracker
    if ~isempty(detections)
        [confTracks,~,allTracks] = tracker(detections,time);
    elseif isLocked(tracker)
        confTracks = predictTracksToTime(tracker,'confirmed',time);
    end

    % Update display
    theaterDisplay(detections,configs,confTracks,tracker);
end
```

Notice that the particles carry a non-Gaussian uncertainty in state along the arc of range-only measurement till the target gets detected by the next sensor. As the target moves through the boundaries of the sensor coverage areas, the likelihood of particles at the boundary increases as compared to other particles. This increase in likelihood triggers a resampling step in the particle filter and the particles collapse to the true target location.

```
showGrabs(theaterDisplay,[1 2]);
```
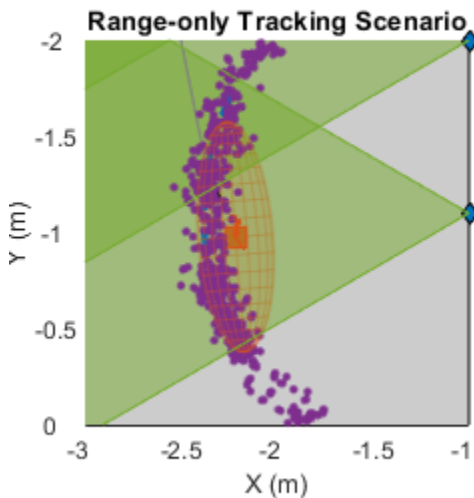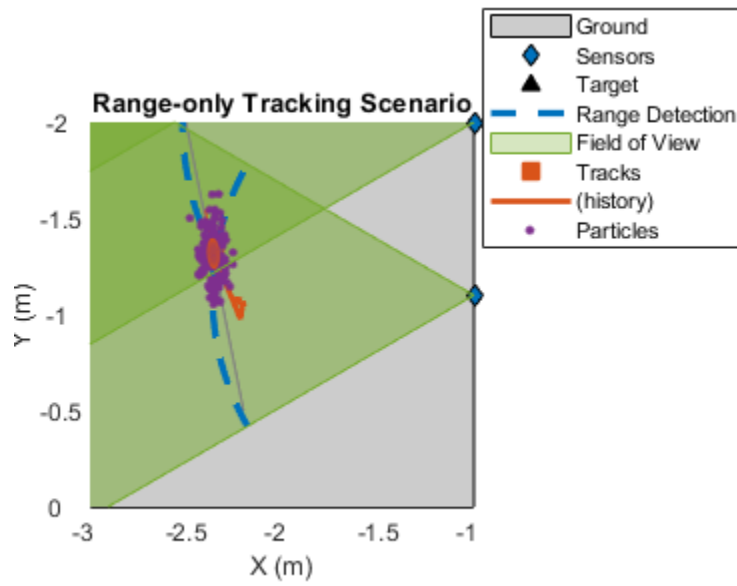
Using the particle filter, the tracker maintains the track for the duration of the scenario.

```
showGrabs(theaterDisplay,3);
```
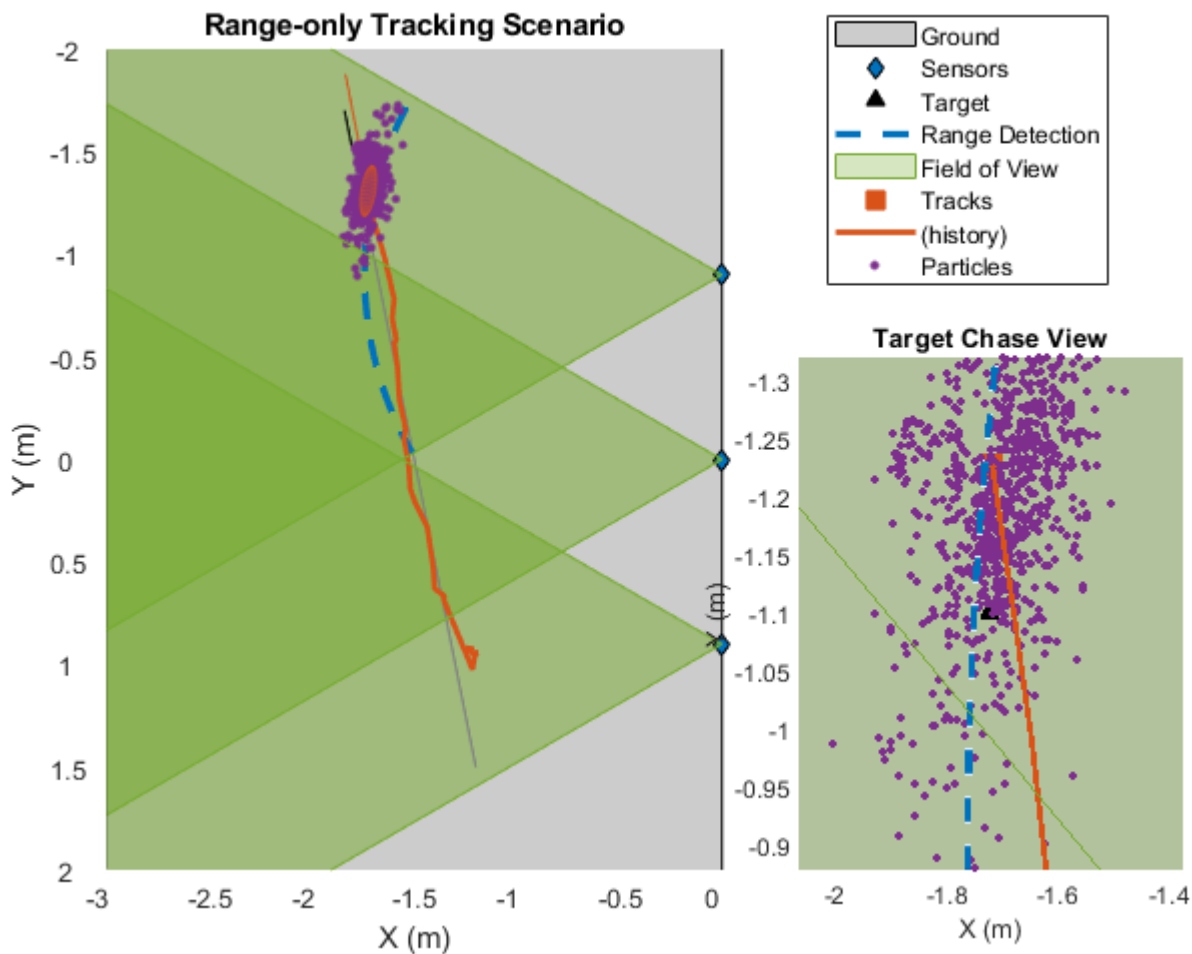
Track Using a Gaussian-Sum Filter

In this section, a Gaussian-sum filter, `trackingGSF`, is used to track the object. For range-only measurements, you can initialize a `trackingGSF` using `initapekf`. The `initapekf` function initializes an angle-parameterized extended Kalman filter. The function `initRangeOnlyGSF` defined in this example modifies the `initapekf` function to track slow moving objects.

```
% restart scene
restart(scene);
release(theaterDisplay);

% Tracker
tracker = trackerGNN('FilterInitializationFcn',@initRangeOnlyGSF);

% Update display to plot Gaussian-sum components
theaterDisplay.FilterType = 'trackingGSF';

while advance(scene)
    % Current time
    time = scene.SimulationTime;
```

```
    % Generate detections
    [detections, configs] = generateRangeDetections(scene);

    % Pass detections to tracker
    if ~isempty(detections)
        [confTracks,~,allTracks] = tracker(detections,time);
    elseif isLocked(tracker)
        confTracks = predictTracksToTime(tracker,'confirmed',time);
    end

    % Update display
    theaterDisplay(detections,configs,confTracks,tracker);
end
```

You can use the `TrackingFilters` property of the `trackingGSF` to see the state of each extended Kalman filter. Represented by "Individual Filters" in the next figure, notice how the filters are aligned along the arc generated by range-only measurement until the target reaches the overlapping region. Immediately after crossing the boundary, the likelihood of the filter at the boundary increases and the track converges to that individual filter. The weight, `ModelProbabilities`, of other individual filters drop as compared to the one closest to the boundary, and their contribution to the estimation of state reduces.

```
showGrabs(theaterDisplay,[4 5]);
```
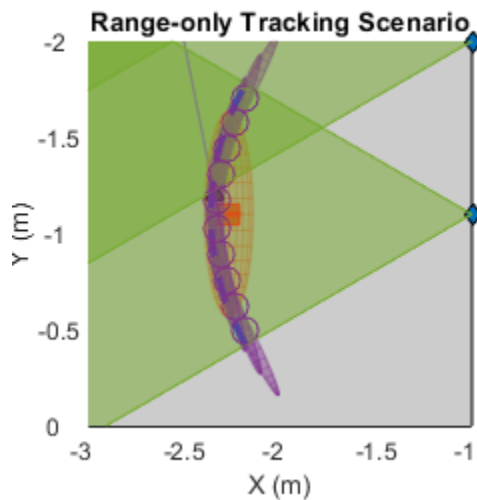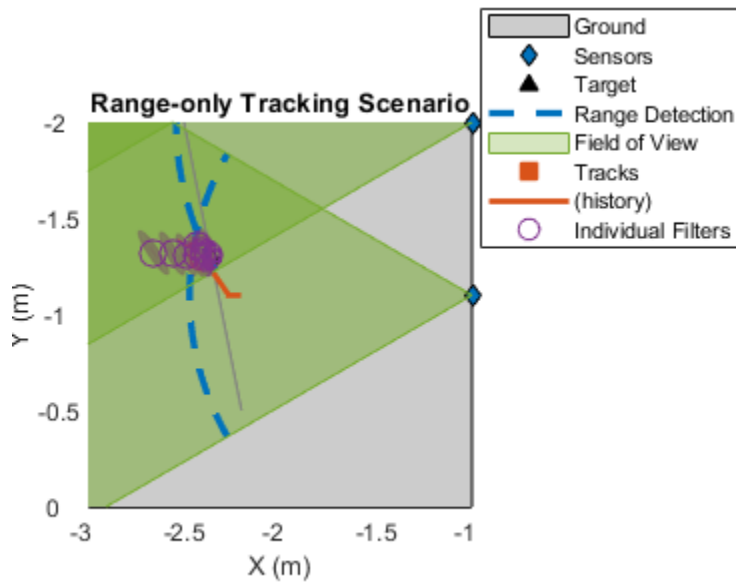
Using the Gaussian-sum filter, the tracker maintains the track during for the duration of the scenario.

```
showGrabs(theaterDisplay,6);
rng(s)
rmpath(exPath);
```
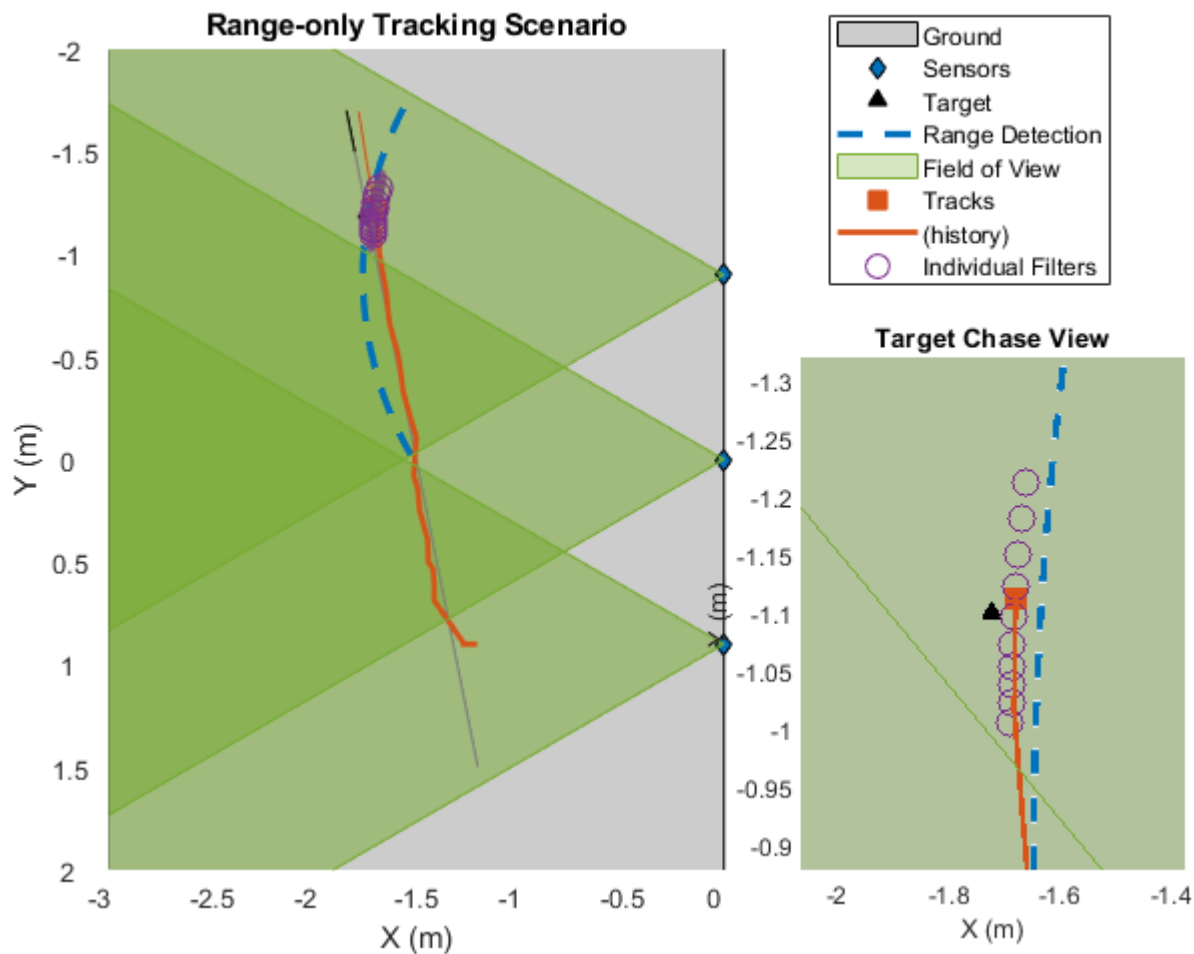
Summary

In this example, you learned how to use a particle filter and a Gaussian-sum filter to track an object using range-only measurements. Both particle filters and Gaussian-sum filters offer capabilities to track objects that follow a non-Gaussian state distribution. While the Gaussian-sum filter approximates the distribution by a weighted sum of Gaussian-components, a particle filter represents this distribution by a set of samples. The particle filter offers a more natural approach to represent any arbitrary distribution as long as samples can be generated from it. However, to represent the distribution perfectly, a large number of particles may be required, which increases computational requirements of using the filter. As state is described by samples, a particle filter architecture allows using any process noise distribution as well as measurement noise. In contrast, the Gaussian-sum filter uses a Gaussian process and measurement noise for each component. One of the major shortcoming of particles is the problem of "sample impoverishment". After resampling, the particles may collapse to regions of high likelihood, not allowing the filter to recover from an "incorrect" association. As no resampling is performed in Gaussian-sum filter and each state is Gaussian, the Gaussian-sum filters offers some capability to recover from such an association. However, this recovery is often dependent on the weights and covariances of each Gaussian-component.

Supporting Functions

generateRangeDetections This function generates range-only detection

```
function [detections,config] = generateRangeDetections(scene)
detections = [];
config = [];
time = scene.SimulationTime;
for i = 1:numel(scene.Platforms)
    [thisDet, thisConfig] = detectOnlyRange(scene.Platforms{i},time);
    config = [config;thisConfig]; %#ok<AGROW>
    detections = [detections;thisDet]; %#ok<AGROW>
end
end
```

detectOnlyRange This function remove az and el from a spherical detection

```
function [rangeDetections,config] = detectOnlyRange(observer,time)
    [fullDetections,numDetections,config] = detect(observer,time);
    % Remove all except range information
    rangeDetections = fullDetections;
    for i = 1:numDetections
        rangeDetections{i}.Measurement = rangeDetections{i}.Measurement(end);
        rangeDetections{i}.MeasurementNoise = rangeDetections{i}.MeasurementNoise(end);
        rangeDetections{i}.MeasurementParameters(1).HasAzimuth = false;
        rangeDetections{i}.MeasurementParameters(1).HasElevation = false;
    end
end
```

initRangeOnlyGSF This function initializes an angle-parameterized Gaussian-sum filter. It uses angle limits of [-30 30] to specify FOV and sets the number of filters as 10. It also modifies the state covariance and process noise for slow moving object in 2-D.

```
function filter = initRangeOnlyGSF(detection)
filter = initapekf(detection,10,[-30 30]);
for i = 1:numel(filter.TrackingFilters)
    filterK = filter.TrackingFilters{i};
    filterK.ProcessNoise = 0.01*eye(3);
    filterK.ProcessNoise(3,3) = 0;
    % Small covariance for slow moving targets
    filterK.StateCovariance(2,2) = 0.1;
    filterK.StateCovariance(4,4) = 0.1;
    % Low standard deviation in z as v = 0;
    filterK.StateCovariance(6,6) = 0.01;
end
end
```

initRangeOnlyCVPF This function initializes a constant velocity particle filter with particles limited to the FOV of [-30 30].

```
function filter = initRangeOnlyCVPF(detection)
filter = initcvpf(detection);

% Uniform az in FOV
az = -pi/6 + pi/3*rand(1,filter.NumParticles);
% no elevation;
el = zeros(1,filter.NumParticles);
% Samples of range from Gaussian range measurement
```

```
    r = detection.Measurement + sqrt(detection.MeasurementNoise)*randn(1,filter.NumParticles);

    % x,y,z in sensor frame
    [x,y,z] = sph2cart(az,el,r);

    % Rotate from sensor to scenario frame.
    senToPlat = detection.MeasurementParameters(1).Orientation';
    senPosition = detection.MeasurementParameters(1).OriginPosition;
    platToScenario = detection.MeasurementParameters(2).Orientation';
    platPosition = detection.MeasurementParameters(2).OriginPosition;
    posPlat = senToPlat*[x;y;z] + senPosition;
    posScen = platToScenario*posPlat + platPosition;

    % Position particles
    filter.Particles(1,:) = posScen(1,:);
    filter.Particles(3,:) = posScen(2,:);
    filter.Particles(5,:) = posScen(3,:);

    % Velocity particles
    % uniform distribution
    filter.Particles([2 4],:) = -1 + 2*rand(2,filter.NumParticles);

    % Process Noise is set to a low number for slow moving targets. Larger than
    % GSF to add more noise to particles preventing them from collapsing.
    filter.ProcessNoise = 0.1*eye(3);

    % Zero z-velocity
    filter.Particles(6,:) = 0;
    filter.ProcessNoise(3,3) = 0;
end
```

Search and Track Scheduling for Multifunction Phased Array Radar

This example shows how to simulate a multifunction phased array radar system. A multifunction radar can perform jobs that usually require multiple traditional radars. Examples of traditional radars are scanning radars, which are responsible for searching targets, and tracking radars, which are responsible for tracking targets. In this example, the multifunction phased array radar performs both scanning (searching) and tracking tasks. Based on the detections and tracks obtained from the current echo, the radar decides what to do next to ensure that targets of interest are tracked and the desired airspace is searched. The multifunction phased array radar works as a closed loop, including features such as task scheduling, waveform selection, detection generation, and target tracking.

This example requires Phased Array System Toolbox™.

Radar Configuration

Assume the multifunction radar operates at S band and must detect targets between 2 km and 100 km, with a minimum target radar cross section (RCS) of 1 square meters.

```
fc     = 2e9;              % Radar carrier frequency (Hz)
c      = 3e8;              % Propagation speed (m/s)
lambda = c/fc;             % Radar wavelength (m)

maxrng = 100e3;            % Maximum range (m)
minrng = 2e3;              % Minimum range (m)
```

Waveform

To satisfy the range requirement, define and use a linear FM waveform with a 1 MHz bandwidth.

```
bw     = 1e6;
fs     = 1.5*bw;
prf    = 1/range2time(maxrng,c);
dcycle = 0.1;

wav = phased.LinearFMWaveform('SampleRate', fs, ...
    'DurationSpecification', 'Duty cycle', 'DutyCycle', dcycle, ...
    'PRF', prf, 'SweepBandwidth', bw);
```

Calculate the range resolution achievable by the waveform.

```
rngres = bw2range(bw,c)
```

```
rngres =

   150
```

Radar Antenna

The multifunction radar is equipped with a phased array that can electronically scan the radar beams in space. Use a 50-by-50 rectangular array with elements separated by half wavelength to achieve a half power beam width of approximately 2 degrees.

```
arraysz    = 50;
ant        = phased.URA('Size',arraysz,'ElementSpacing',lambda/2);
```

```
ant.Element.BackBaffled = true;

arraystv  = phased.SteeringVector('SensorArray',ant,'PropagationSpeed',c);
radiator  = phased.Radiator('OperatingFrequency',fc, ...
    'PropagationSpeed', c, 'Sensor',ant, 'WeightsInputPort', true);
collector = phased.Collector('OperatingFrequency',fc, ...
    'PropagationSpeed', c, 'Sensor',ant);

beamw = rad2deg(lambda/(arraysz*lambda/2))


beamw =

    2.2918
```

Transmitter and Receiver

Use the detection requirements to derive the appropriate transmit power. Assume the noise figure on the receiving preamplifier is 7 dB.

```
pd      = 0.9;                       % Probability of detection
pfa     = 1e-6;                      % Probability of false alarm
snr_min = albersheim(pd, pfa, 1);
ampgain = 20;
tgtrcs  = 1;
ant_snrgain = pow2db(arraysz^2);

ppower  = radareqpow(lambda,maxrng,snr_min,wav.PulseWidth,...
    'RCS',tgtrcs,'Gain',ampgain+ant_snrgain);

tx = phased.Transmitter('PeakPower',ppower,'Gain',ampgain,'InUseOutputPort',true);
rx = phased.ReceiverPreamp('Gain',ampgain,'NoiseFigure',7,'EnableInputPort',true);
```

Signal Processing

The multifunction radar applies a sequence of operations, including matched filtering, time varying gain, monopulse, and detection, to the received signal to generate range and angle measurements of the detected targets.

```
% matched filter
mfcoeff = getMatchedFilter(wav);
mf      = phased.MatchedFilter('Coefficients',mfcoeff,'GainOutputPort', true);

% time varying gain
tgrid   = unigrid(0,1/fs,1/prf,'[)');
rgates  = c*tgrid/2;
rngloss = 2*fspl(rgates,lambda);
refloss = 2*fspl(maxrng,lambda);
tvg     = phased.TimeVaryingGain('RangeLoss',rngloss,'ReferenceLoss',refloss);

% monopulse
monfeed = phased.MonopulseFeed('SensorArray',ant,'PropagationSpeed',c,...
    'OperatingFrequency',fc,'SquintAngle',1);
monest  = getMonopulseEstimator(monfeed);
```

Data Processing

The detections are fed into a tracker, which performs several operations. The tracker maintains a list of tracks, that is, estimates of target states in the area of interest. If a detection cannot be assigned to any track already maintained by the tracker, the tracker initiates a new track. In most cases, whether the new track represents a true target or false target is unclear. At first, a track is created with a tentative status. If enough detections are obtained, the track becomes confirmed. Similarly, if no detections are assigned to a track, the track is coasted (predicted without correction). If the track has a few missed updates, the tracker deletes the track.

The multifunction radar uses a tracker that associates the detections to the tracks using a global nearest neighbor (GNN) algorithm.

```
tracker = trackerGNN('FilterInitializationFcn',@initMPARGNN,...
    'ConfirmationThreshold',[2 3], 'DeletionThreshold',5,...
    'HasDetectableTrackIDsInput',true,'AssignmentThreshold',100,...
    'MaxNumTracks',2,'MaxNumSensors',1);
```

Group all radar components together in a structure for easier reference in the simulation loop.

```
mfradar.Tx       = tx;
mfradar.Rx       = rx;
mfradar.TxAnt    = radiator;
mfradar.RxAnt    = collector;
mfradar.Wav      = wav;
mfradar.RxFeed   = monfeed;
mfradar.MF       = mf;
mfradar.TVG      = tvg;
mfradar.DOA      = monest;
mfradar.STV      = arraystv;
mfradar.Tracker = tracker;
mfradar.IsTrackerInitialized = false;
```

Target and Scene Definition

This example assumes the radar is stationary at the origin with two targets in its field of view. One target departs from the radar and is at a distance of around 50 km. The other target approaches the radar and is 30 km away. Both targets have an RCS of 1 square meters.

```
% Define the targets.
tgtpos = [29875 49637; 0 4225; 0 0];
tgtvel = [-100 120; 0 100; 0 0];

ntgt = size(tgtpos,2);
tgtmotion = phased.Platform('InitialPosition',tgtpos,'Velocity',tgtvel);
target = phased.RadarTarget('MeanRCS',tgtrcs*ones(1,ntgt),'OperatingFrequency',fc);
```

Assume the propagation environment is free space.

```
channel = phased.FreeSpace('SampleRate',fs,'TwoWayPropagation',true,'OperatingFrequency',fc);
```

Group targets and propagation channels together in a structure for easier reference in the simulation loop.

```
env.Target       = target;
env.TargetMotion = tgtmotion;
env.Channel      = channel;
```

Radar Resource Management

While using one multifunction radar to perform multiple tasks has its advantages, it also has a higher cost and more sophisticated logic. In general, a radar has finite resources to spend on its tasks. If resources are used for tracking tasks, then those resources are not available for searching tasks until the tracking tasks are finished. Because of this resource allocation, a critical component when using a multifunction radar is resource management.

Search Tasks

The search tasks can be considered as deterministic. In this example, a raster scan is used to cover the desired airspace. If no other tasks exist, the radar scans the space one angular cell at a time. The size of an angular cell is determined by the beam width of the antenna array.

Assume the radar scans a space from -30 to 30 degrees azimuth and 0 to 20 degrees elevation. Calculate the angular search grid using the beam width.

```
scanregion   = [-30, 30, 0, 20];
azscanspan   = diff(scanregion(1:2));
numazscan    = ceil(azscanspan/beamw);
azscanangles = linspace(scanregion(1),scanregion(2),numazscan);
elscanspan   = diff(scanregion(3:4));
numelscan    = ceil(elscanspan/beamw);
elscanangles = linspace(scanregion(3),scanregion(4),numelscan);
[elscangrid,azscangrid] = meshgrid(elscanangles,azscanangles);
scanangles   = [azscangrid(:) elscangrid(:)].';
```

The beam position grid and target scene are shown below.

```
sceneplot = helperMPARTaskPlot('initialize',scanangles,azscanangles,maxrng,beamw,tgtpos);
```
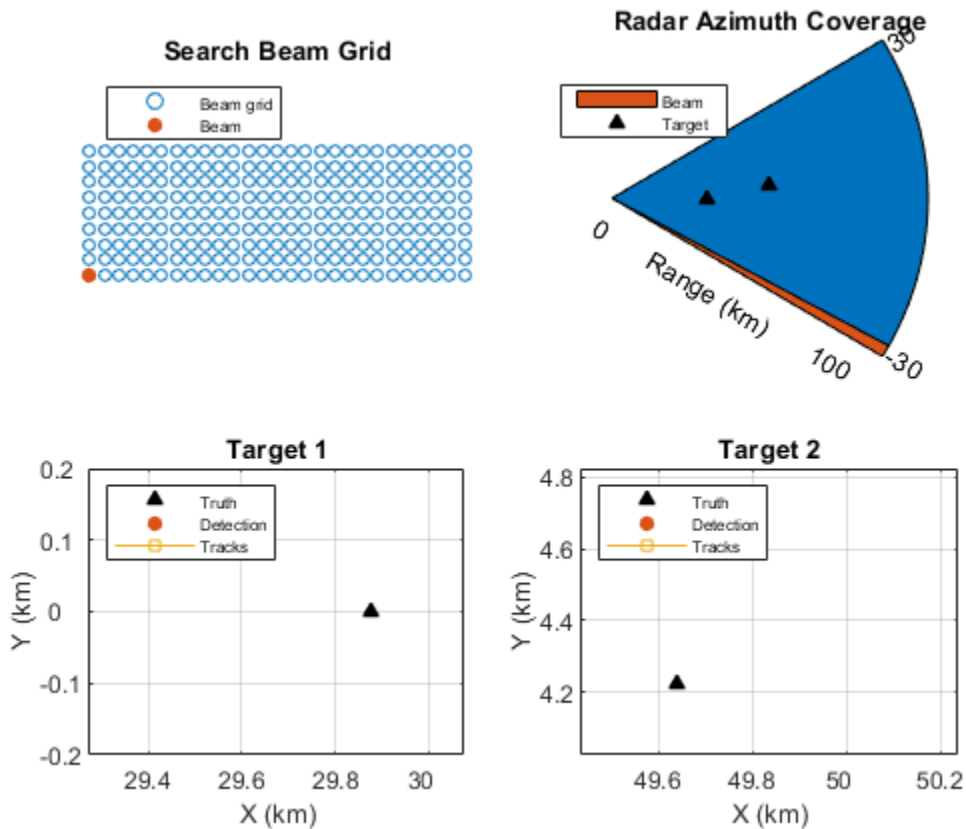
The search beams are transmitted one at a time sequentially until the entire search area is covered. Once the entire search area is covered, the radar repeats the search sequence. The searches are performed along the azimuthal direction, one elevation angle a time. The search tasks are often contained in a job queue.

```
searchq = struct('JobType','Search','BeamDirection',num2cell(scanangles,1),...
    'Priority',1000,'WaveformIndex',1);
current_search_idx = 1;
```

Each job in the queue specifies the job type as well as the pointing direction of the beam. It also contains a priority value for the job. This priority value is determined by the job type. This example uses a value of 1000 as the priority for search jobs.

```
disp(searchq(current_search_idx))
```

```
        JobType: 'Search'
  BeamDirection: [2x1 double]
       Priority: 1000
  WaveformIndex: 1
```

Track Tasks

Unlike search tasks, track tasks cannot be planned. Track tasks are created only when a target is detected by a search task or when the target has already been tracked. Track tasks are dynamic tasks that get created and executed based on the changing scenario. Similar to search tasks, track tasks are also managed in a job queue.

```
trackq(10) = struct('JobType',[],'BeamDirection',[],'Priority',3000,'WaveformIndex',[],...
    'Time',[],'Range',[],'TrackID',[]);
num_trackq_items = 0;
disp(trackq(1))
```

```
        JobType: []
  BeamDirection: []
       Priority: []
  WaveformIndex: []
           Time: []
          Range: []
        TrackID: []
```

Group search and track queues together in a structure for easier reference in the simulation loop.

```
jobq.SearchQueue  = searchq;
jobq.SearchIndex  = current_search_idx;
jobq.TrackQueue   = trackq;
jobq.NumTrackJobs = num_trackq_items;
```

Because a tracking job cannot be initialized before a target is detected, all tracking jobs start as empty jobs. Once a job is created, it contains the information such as its job type, the direction of the beam, and time to execute. The tracking task has a priority of 3000, which is higher than the priority of 1000 for a search job. This higher priority value means that when the time is in conflict, the system will execute the tracking job first.

The size limit for the queue in this example is set to 10.

Task Scheduling

In this example, for simplicity, the multifunction radar executes only one type of job within a small time period, often referred to as a dwell, but can switch tasks at the beginning of each dwell. For each dwell, the radar looks at all tasks that are due for execution and picks the one that has the highest priority. Consequently, jobs that get postponed will now have an increased priority and are more likely to be executed in the next dwell.

Simulation

This section of the example simulates a short run of the multifunction radar system. The entire structure of the multifunction radar simulation is represented by this diagram.
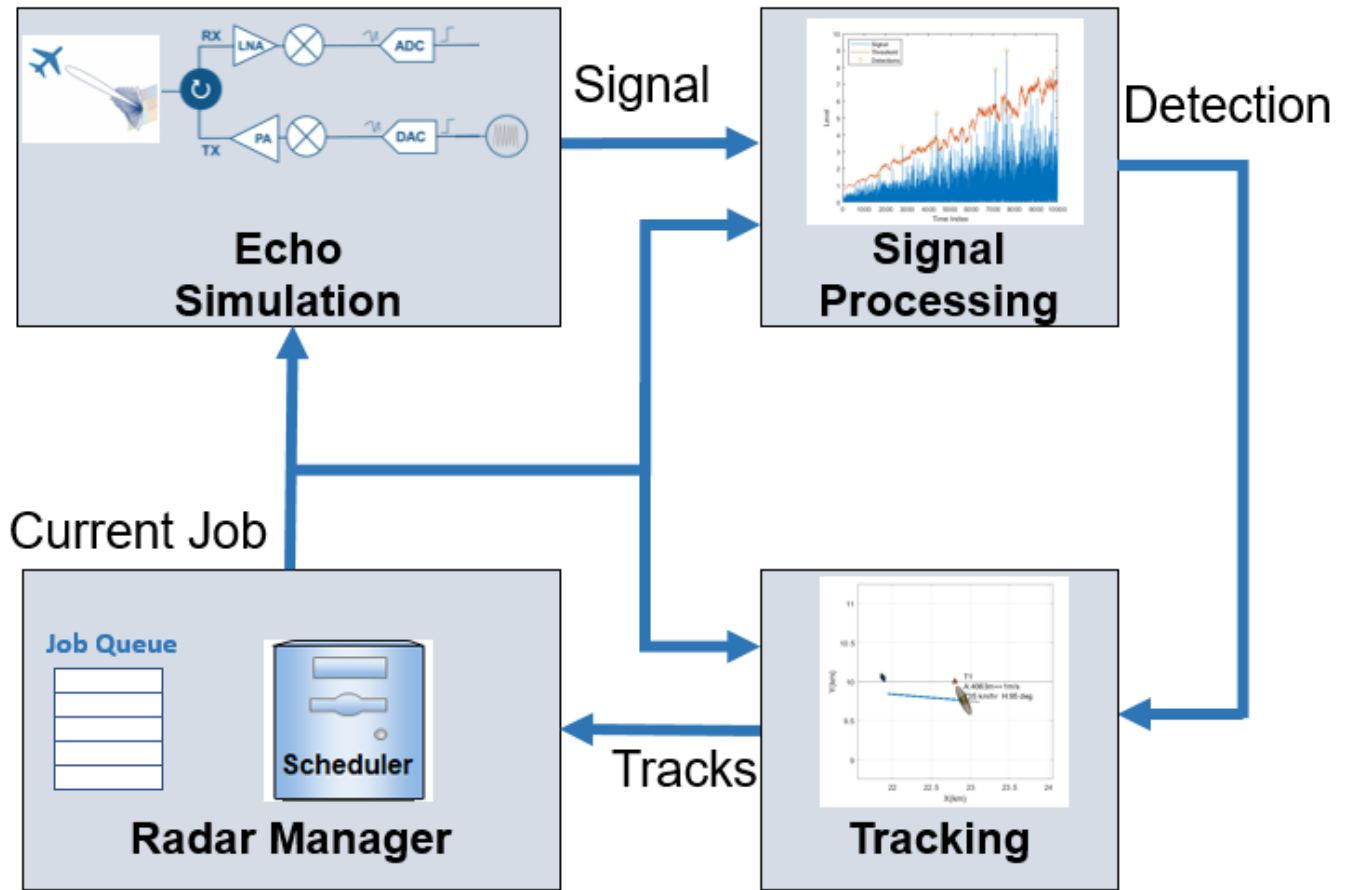
The simulation starts with the radar manager, which provides an initial job. Based on this job, the radar transmits the waveform, simulates the echo, and applies signal processing to generate the detection. The detection is processed by a tracker to create tracks for targets. The tracks then go back to the radar manager. Based on the tracks and the knowledge about the scene, the radar manager schedules new track jobs and picks the job for the next dwell.

The logic of the radar manager operation is shown in this flowchart and described in these steps.
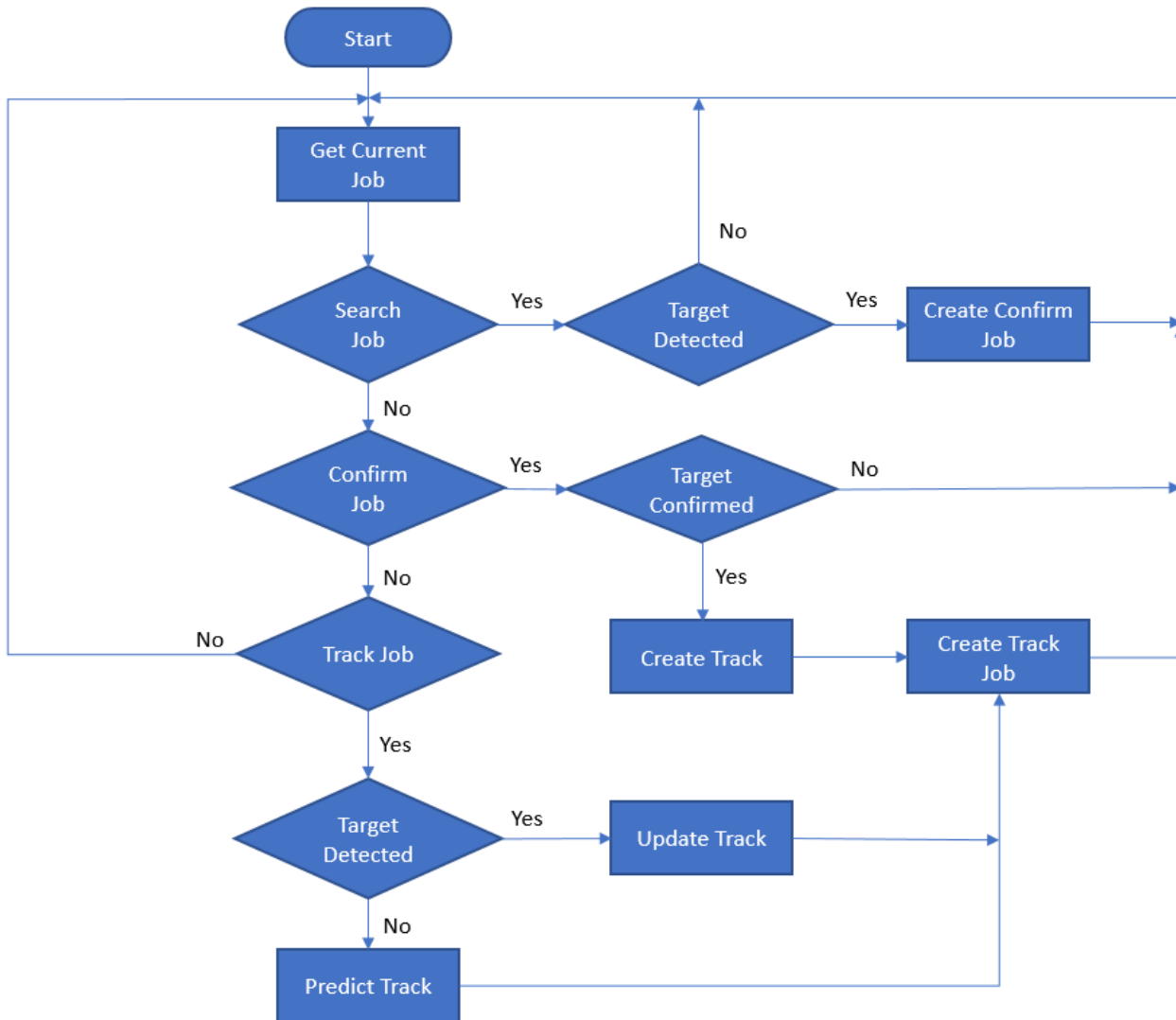
1 The radar starts with a search job.

2 If a target is present in the detection, the radar schedules a confirmation job in the same
  direction to ensure that the presence of this target is not a false alarm. The confirmation task has
  a higher priority than the search task but not as high as the track task. If the detection gets
  confirmed, a track is established, and a track job is created to be executed after a given revisit
  time. If the detection is not confirmed, then the original detection is considered as a false alarm,
  and no track is created.

3 If the current job is a track job, the radar performs the detection, updates the track, and creates
  a future track job.

4 Based on the priority and time for execution, the radar selects the next job.

Assume a dwell is 10 ms. At the beginning of the simulation, the radar is configured to search one
beam at a time.

```
rng(2018);
current_time = 0;
```

```
Npulses      = 10;
numdwells    = 200;
dwelltime    = 0.01;

jobload.num_search_job = zeros(1,numdwells);
jobload.num_track_job  = zeros(1,numdwells);
```

You can run the example in its entirety to see the plots being dynamically updated during execution. In the top two plots, the color of the beams indicate the types of the current job: red for search, yellow for confirm, and purple for track. The bottom two plots show the true locations (triangle), detections (circle), and tracks (square) of the two targets, respectively. System log also displays in the command line to explain the system behavior at the current moment. Next, the example shows more details about several critical moments of the simulation.

Simulate the system behavior until it detects the first target. The simulation loop follows the previous system diagram.

```
for dwell_idx = 1:14
    % Scheduler to provide current job
    [current_job,jobq]      = getCurrentJob(jobq,current_time);

    % Simulate the received I/Q signal
    [xsum,xdaz,xdel,mfradar] = generateEcho(mfradar,env,current_job);

    % Signal processor to extract detection
    [detection,mfradar]     = generateDetection(xsum,xdaz,xdel,mfradar,current_job,current_time)

    % Radar manager to perform data processing and update track queue
    [jobq,allTracks,mfradar] = updateTrackAndJob(detection,jobq,mfradar,current_job,current_time

    % Visualization
    helperMPARTaskPlot('update',sceneplot,current_job,maxrng,beamw,tgtpos,allTracks,detection.Mea

    % Update time
    tgtpos = env.TargetMotion(dwelltime-Npulses/mfradar.Wav.PRF);
    current_time = current_time+dwelltime;

    % Record resource allocation
    if strcmp(current_job.JobType,'Search')
        jobload.num_search_job(dwell_idx) = 1;
    else
        jobload.num_track_job(dwell_idx)  = 1;
    end

end
```

```
0.000000 sec:    Search    [-30.000000 0.000000]
0.010000 sec:    Search    [-27.692308 0.000000]
0.020000 sec:    Search    [-25.384615 0.000000]
0.030000 sec:    Search    [-23.076923 0.000000]
0.040000 sec:    Search    [-20.769231 0.000000]
0.050000 sec:    Search    [-18.461538 0.000000]
0.060000 sec:    Search    [-16.153846 0.000000]
0.070000 sec:    Search    [-13.846154 0.000000]
0.080000 sec:    Search    [-11.538462 0.000000]
0.090000 sec:    Search    [-9.230769 0.000000]
0.100000 sec:    Search    [-6.923077 0.000000]
```

```
0.110000 sec:    Search    [-4.615385 0.000000]
0.120000 sec:    Search    [-2.307692 0.000000]
0.130000 sec:    Search    [0.000000 0.000000]    Target detected at 29900.000000 m
```

Search Beam Grid

Radar Azimuth Coverage
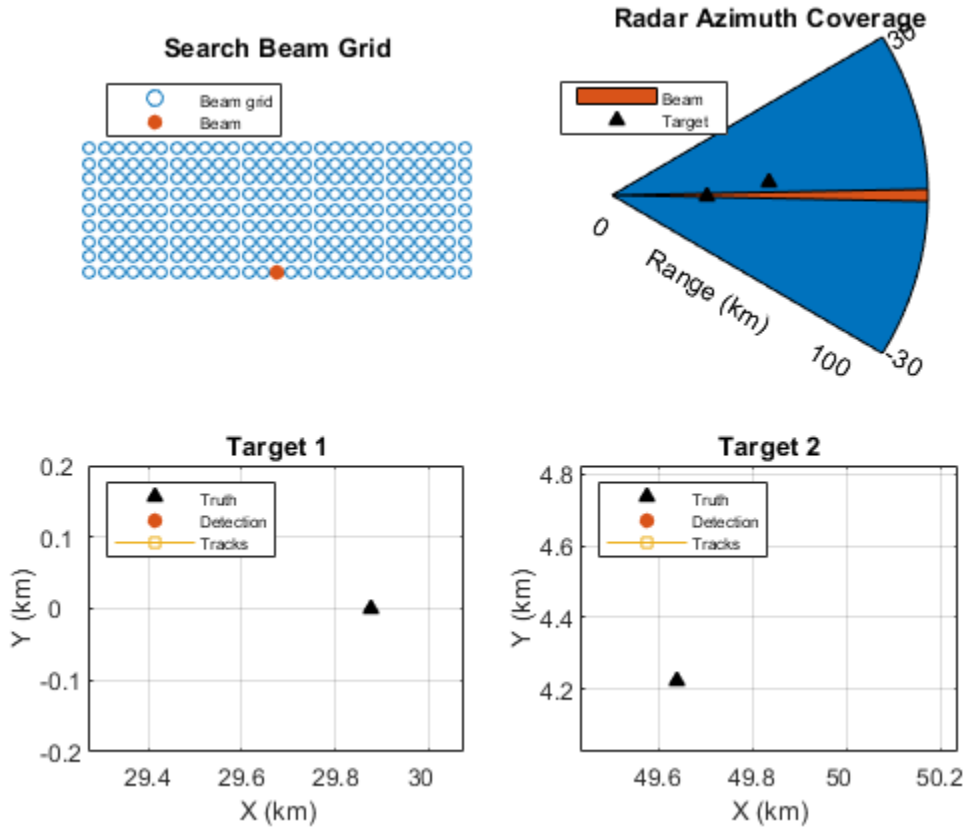
Target 1

Target 2

As expected, the radar gets a detection when the radar beam illuminates the target, as shown in the figure. When this happens, the radar sends a confirmation beam immediately to make sure it is not a false detection.

Next, show the results for the confirmation job. The rest of this example shows simplified code that combines the simulation loop into a system simulation function.

```
[mfradar,env,jobq,jobload,current_time,tgtpos] = MPARSimRun(...
    mfradar,env,jobq,jobload,current_time,dwelltime,sceneplot,maxrng,beamw,tgtpos,15,15);
```

```
0.140000 sec:    Confirm    [-0.000586 -0.000034]    Created track 1 at 29900.000000 m
```

Search Beam Grid

Radar Azimuth Coverage
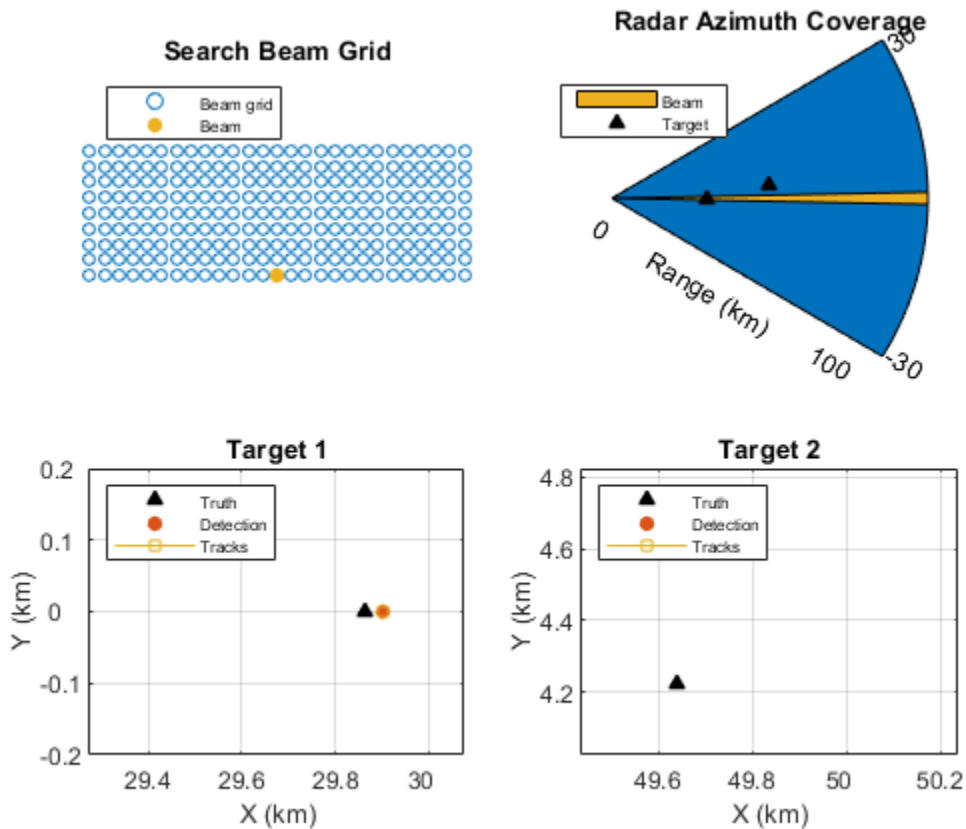
Target 1

Target 2

The figure now shows the confirmation beam. Once the detection is confirmed, a track is established for the target, and a track job is scheduled to execute after a short interval.

This process repeats for every detected target until the revisit time, at which point the multifunction radar stops the search sequence and performs the track task again.

```
[mfradar,env,jobq,jobload,current_time,tgtpos] = MPARSimRun(...
    mfradar,env,jobq,jobload,current_time,dwelltime,sceneplot,maxrng,beamw,tgtpos,16,25);
```

```
0.150000 sec:   Search    [2.307692 0.000000]
0.160000 sec:   Search    [4.615385 0.000000]    Target detected at 49900.000000 m
0.170000 sec:   Confirm    [4.881676 0.000739]    Created track 2 at 49900.000000 m
0.180000 sec:   Search    [6.923077 0.000000]
0.190000 sec:   Search    [9.230769 0.000000]
0.200000 sec:   Search    [11.538462 0.000000]
0.210000 sec:   Search    [13.846154 0.000000]
0.220000 sec:   Search    [16.153846 0.000000]
0.230000 sec:   Search    [18.461538 0.000000]
0.240000 sec:   Track     [-0.000399 0.000162]    Track 1 at 29900.000000 m
```
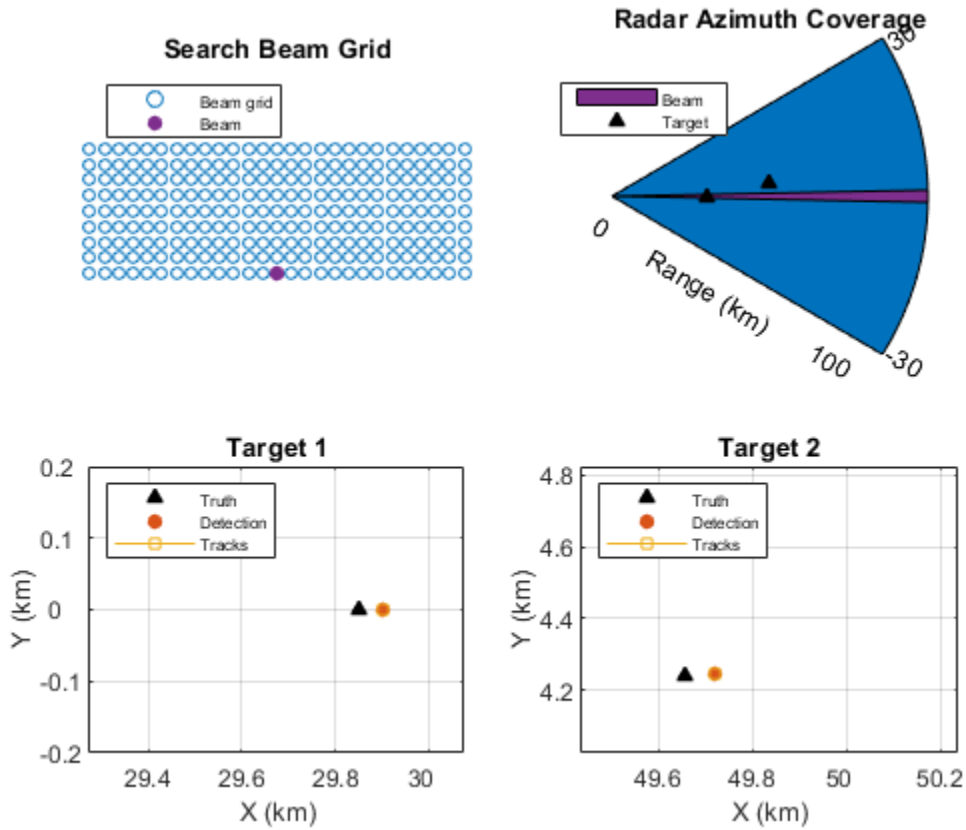
The results show that the simulation stops at a track beam. The zoomed-in figures around the two targets show how the tracks are updated based on the detection and measurements. A new track job for the next revisit is also added to the job queue during the execution of a track job.

This process repeats for each dwell. This simulation runs the radar system for a 2-second period. After a while, the second target is detected beyond 50 km. Based on this information, the radar manager reduces how often the system needs to track the second target. This reduction frees up resources for other, more urgent needs.

```
[mfradar,env,jobq,jobload,current_time,tgtpos] = MPARSimRun(...
    mfradar,env,jobq,jobload,current_time,dwelltime,sceneplot,maxrng,beamw,tgtpos,26,numdwells);
```

```
0.250000 sec:    Search    [20.769231 0.000000]
0.260000 sec:    Search    [23.076923 0.000000]
0.270000 sec:    Track     [4.882892 -0.000030]    Track 2 at 49900.000000 m
0.280000 sec:    Search    [25.384615 0.000000]
0.290000 sec:    Search    [27.692308 0.000000]
0.340000 sec:    Track     [0.001390 0.000795]     Track 1 at 29900.000000 m
0.370000 sec:    Track     [4.895153 0.000529]     Track 2 at 49900.000000 m
0.440000 sec:    Track     [0.000284 0.000446]     Track 1 at 29900.000000 m
0.470000 sec:    Track     [4.909764 -0.000394]    Track 2 at 49900.000000 m
0.540000 sec:    Track     [0.000455 -0.000130]    Track 1 at 29800.000000 m
0.570000 sec:    Track     [4.921876 -0.000210]    Track 2 at 49900.000000 m
0.640000 sec:    Track     [0.000181 -0.000020]    Track 1 at 29800.000000 m
0.670000 sec:    Track     [4.932942 -0.000988]    Track 2 at 49900.000000 m
0.740000 sec:    Track     [0.000348 0.000212]     Track 1 at 29800.000000 m
```

```
0.770000 sec:    Track    [4.944255 -0.001073]    Track 2 at 49900.000000 m
0.840000 sec:    Track    [0.000171 -0.000125]    Track 1 at 29800.000000 m
0.870000 sec:    Track    [4.954431 -0.000943]    Track 2 at 50000.000000 m
0.940000 sec:    Track    [0.000296 -0.000288]    Track 1 at 29800.000000 m
1.040000 sec:    Track    [0.000108 -0.000147]    Track 1 at 29800.000000 m
1.140000 sec:    Track    [-0.000096 -0.000179]    Track 1 at 29800.000000 m
1.240000 sec:    Track    [-0.000110 -0.000315]    Track 1 at 29800.000000 m
1.340000 sec:    Track    [-0.000291 -0.000515]    Track 1 at 29800.000000 m
1.370000 sec:    Track    [5.005679 -0.000877]    Track 2 at 50000.000000 m
1.440000 sec:    Track    [-0.000191 -0.000592]    Track 1 at 29800.000000 m
1.540000 sec:    Track    [-0.000140 -0.000787]    Track 1 at 29700.000000 m
1.640000 sec:    Track    [0.000069 -0.000600]    Track 1 at 29700.000000 m
1.740000 sec:    Track    [-0.000001 -0.000714]    Track 1 at 29700.000000 m
1.840000 sec:    Track    [0.000030 -0.000686]    Track 1 at 29700.000000 m
1.870000 sec:    Track    [5.057762 0.000107]    Track 2 at 50100.000000 m
1.940000 sec:    Track    [0.000067 -0.000511]    Track 1 at 29700.000000 m
```

Search Beam Grid

Radar Azimuth Coverage

Target 1

Target 2
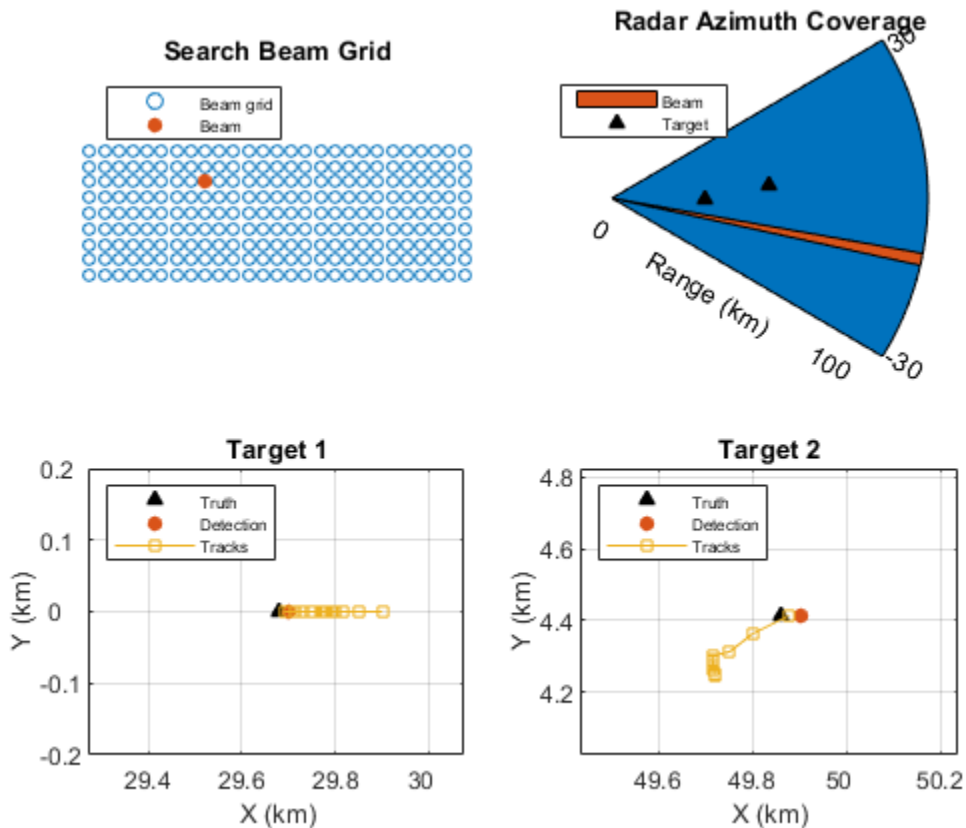
Resource Distribution Analysis

This section of the example shows how the radar resource is distributed among different tasks. This figure shows how the multifunction radar system in this example distributes its resources between search and track.

```
L = 10;
searchpercent = sum(buffer(jobload.num_search_job,L,L-1,'nodelay'))/L;
trackpercent  = sum(buffer(jobload.num_track_job,L,L-1,'nodelay'))/L;
figure;
plot((1:numel(searchpercent))*L*dwelltime,[searchpercent(:) trackpercent(:)]);
```

```matlab
xlabel('Time (s)');
ylabel('Job Percentage');
title('Resource Distribution between Search and Track');
legend('Search','Track','Location','best');
grid on;
```
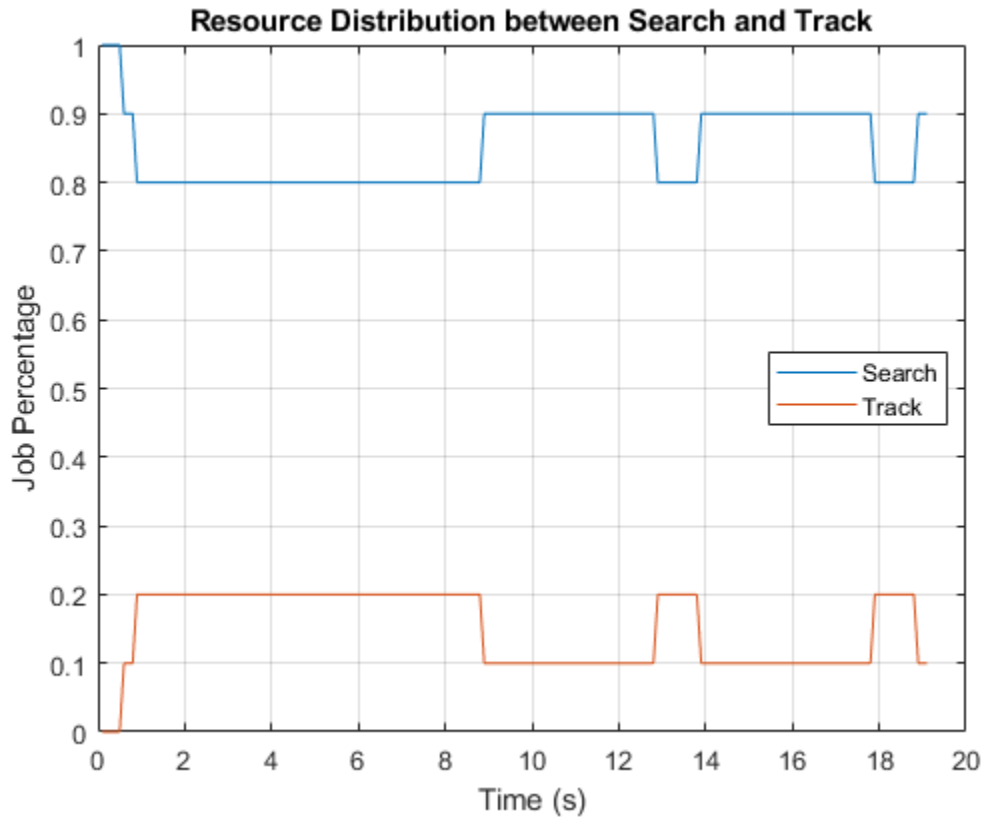

The figure suggests that, at the beginning of the simulation, all resources are spent on search. Once the targets are detected, the radar resources get split into 80% and 20% between search and track, respectively. However, once the second target gets farther away, more resources are freed up for search. The track load increases briefly when the time arrives to track the second target again.

Summary

This example introduces the concept of resource management and task scheduling in a multifunctional phased array radar system. It shows that, with the resource management component, the radar acts as a closed loop system. Although the multifunction radar in this example deals only with search and track tasks, the concept can be extended to more realistic situations where other functions, such as self-check and communication, are also involved.

References

[1] Walter Weinstock, "Computer Control of a Multifunction Radar", *Practical Phased Array Antenna Systems*, Lex Book, 1997

Appendices

These helper functions models the radar resource management workflow.

getCurrentJob

The function `getCurrentJob` compares the jobs in the search queue and the track queue and selects the job with the highest priority to execute.

```
function [currentjob,jobq] = getCurrentJob(jobq,current_time)

searchq   = jobq.SearchQueue;
trackq    = jobq.TrackQueue;
searchidx = jobq.SearchIndex;
num_trackq_items = jobq.NumTrackJobs;

% Update search queue index
searchqidx = mod(searchidx-1,numel(searchq))+1;

% Find the track job that is due and has the highest priority
readyidx = find([trackq(1:num_trackq_items).Time]<=current_time);
[~,maxpidx] = max([trackq(readyidx).Priority]);
taskqidx = readyidx(maxpidx);

% If the track job found has a higher priority, use that as the current job
% and increase the next search job priority since it gets postponed.
% Otherwise, the next search job due is the current job.
if ~isempty(taskqidx) && trackq(taskqidx).Priority >= searchq(searchqidx).Priority
    currentjob = trackq(taskqidx);
    for m = taskqidx+1:num_trackq_items
        trackq(m-1) = trackq(m);
    end
    num_trackq_items = num_trackq_items-1;
    searchq(searchqidx).Priority = searchq(searchqidx).Priority+100;
else
    currentjob = searchq(searchqidx);
    searchidx = searchqidx+1;

end

jobq.SearchQueue  = searchq;
jobq.SearchIndex  = searchidx;
jobq.TrackQueue   = trackq;
jobq.NumTrackJobs = num_trackq_items;
```

generateEcho

The function `generateEcho` simulates the complex (I/Q) baseband representation of the target echo received at the radar.

```
function [xrsint,xrdazint,xrdelint,mfradar] = generateEcho(mfradar,env,current_job)

% Radar position
radarpos = [0;0;0];
radarvel = [0;0;0];

% Number of pulses and operating frequency
Npulses = 10;
fc = mfradar.TxAnt.OperatingFrequency;
```

```
for m = 1:Npulses
    % Waveform
    x = mfradar.Wav();

    % Update target motion
    [tgtpos,tgtvel] = env.TargetMotion(1/mfradar.Wav.PRF);
    [~,tgtang]      = rangeangle(tgtpos);

    % Transmit
    [xt,inuseflag]  = mfradar.Tx(x);
    w  = mfradar.STV(fc,current_job.BeamDirection);
    xt = mfradar.TxAnt(xt,tgtang,conj(w));

    % Propagation
    xp = env.Channel(xt,radarpos,tgtpos,radarvel,tgtvel);
    xp = env.Target(xp);

    % Receive and monopulse
    xr = mfradar.RxAnt(xp,tgtang);
    [xrs,xrdaz,xrdel] = mfradar.RxFeed(xr,current_job.BeamDirection);

    % Pulse integration
    if m == 1
        xrsint   = mfradar.Rx(xrs,~(inuseflag>0));
        xrdazint = mfradar.Rx(xrdaz,~(inuseflag>0));
        xrdelint = mfradar.Rx(xrdel,~(inuseflag>0));
    else
        xrsint   = xrsint+mfradar.Rx(xrs,~(inuseflag>0));
        xrdazint = xrdazint+mfradar.Rx(xrdaz,~(inuseflag>0));
        xrdelint = xrdelint+mfradar.Rx(xrdel,~(inuseflag>0));
    end
end
```

generateDetection

The function `generateDetection` applies signal processing techniques on the echo to generate target detection.

```
function [detection,mfradar] = generateDetection(xrsint,xrdazint,xrdelint,mfradar,current_job,cu

% Compute detection threshold
nbw         = mfradar.Rx.SampleRate/(mfradar.Wav.SampleRate/mfradar.Wav.SweepBandwidth);
npower      = noisepow(nbw,mfradar.Rx.NoiseFigure,mfradar.Rx.ReferenceTemperature);
pfa         = 1e-6;
threshold   = npower * db2pow(npwgnthresh(pfa,1,'noncoherent'));
arraysz     = mfradar.TxAnt.Sensor.Size(1);
ant_snrgain = pow2db(arraysz^2);
mfcoeff     = getMatchedFilter(mfradar.Wav);
mfgain      = pow2db(norm(mfcoeff)^2);
threshold   = threshold * db2pow(mfgain+2*ant_snrgain);
threshold   = sqrt(threshold);

tgrid   = unigrid(0,1/mfradar.Wav.SampleRate,1/mfradar.Wav.PRF,'[)');
rgates  = mfradar.TxAnt.PropagationSpeed*tgrid/2;
```

```
% Matched filtering and time varying gain
xrsmf = mfradar.TVG(mfradar.MF(xrsint));

% Detection in range and angle estimation via monopulse
if any(abs(xrsmf)>threshold)
    [~,tgtidx] = findpeaks(abs(xrsmf),'MinPeakHeight',threshold,...
        'Sortstr','Descend','NPeaks',1);
    rng_est = rgates(tgtidx-(numel(mfcoeff)-1));
    ang_est = mfradar.DOA(xrsint(tgtidx-1),xrdazint(tgtidx-1),xrdelint(tgtidx-1),current_job.Bea
    % Form the detection object.
    measNoise = diag([0.1, 0.1, 150].^2);              % Measurement noise matrix
    detection = objectDetection(current_time,...
        [ang_est(1);ang_est(2);rng_est], 'MeasurementNoise', measNoise,...
        'MeasurementParameters',struct('Frame','spherical', 'HasVelocity', false));
else
    detection = objectDetection.empty;
end

if current_time < 0.3 || strcmp(current_job.JobType,'Track')
    fprintf('\n%f sec:\t%s\t[%f %f]',current_time,current_job.JobType,current_job.BeamDirection(1
        current_job.BeamDirection(2));
end
```

updateTrackAndJob

The function `updateTrackAndJob` tracks the detection and then passes tracks to the radar manager
to update the track task queue.

```
function [jobq,allTracks,mfradar] = updateTrackAndJob(detection,jobq,mfradar,current_job,current_

trackq          = jobq.TrackQueue;
num_trackq_items = jobq.NumTrackJobs;

% Execute current job
switch current_job.JobType
    case 'Search'
        % For search job, if there is a detection, establish tentative
        % track and schedule a confirmation job
        if ~isempty(detection)
            ang_est = detection.Measurement(1:2);
            rng_est = detection.Measurement(3);
            if ~mfradar.IsTrackerInitialized
                [~,~,allTracks] = mfradar.Tracker(detection,current_time,uint32([]));
                mfradar.IsTrackerInitialized = true;
            else
                [~,~,allTracks] = mfradar.Tracker(detection,current_time,uint32([]));
            end
            num_trackq_items = num_trackq_items+1;
            trackq(num_trackq_items) = struct('JobType','Confirm','Priority',2000,...
                'BeamDirection',ang_est,'WaveformIndex',1,'Time',current_time+dwellinterval,...
                'Range',rng_est,'TrackID',allTracks(~[allTracks.IsConfirmed]).TrackID);
            if current_time < 0.3 || strcmp(current_job.JobType,'Track')
                fprintf('\tTarget detected at %f m',rng_est);
            end
        else
```

```matlab
                    allTracks = [];
            end

        case 'Confirm'
            % For confirm job, if the detection is confirmed, establish a track
            % and create a track job corresponding to the revisit time
            if ~isempty(detection)
                trackid = current_job.TrackID;
                [~,~,allTracks] = mfradar.Tracker(detection,current_time,trackid);

                rng_est = detection.Measurement(3);
                if rng_est >= 50e3
                    updateinterval = 0.5;
                else
                    updateinterval = 0.1;
                end
                revisit_time = current_time+updateinterval;
                predictedTrack = predictTracksToTime(mfradar.Tracker,trackid,revisit_time);
                xpred = predictedTrack.State([1 3 5]);
                [phipred,thetapred,rpred] = cart2sph(xpred(1),xpred(2),xpred(3));
                num_trackq_items = num_trackq_items+1;
                trackq(num_trackq_items) = struct('JobType','Track','Priority',3000,...
                    'BeamDirection',rad2deg([phipred;thetapred]),'WaveformIndex',1,'Time',revisit_tir
                    'Range',rpred,'TrackID',trackid);
                if current_time < 0.3 || strcmp(current_job.JobType,'Track')
                    fprintf('\tCreated track %d at %f m',trackid,rng_est);
                end
            else
                allTracks = [];
            end

        case 'Track'
            % For track job, if there is a detection, update the track and
            % schedule a track job corresponding to the revisit time. If there
            % is no detection, predict and schedule a track job sooner so the
            % target is not lost.
            if ~isempty(detection)
                trackid = current_job.TrackID;
                [~,~,allTracks] = mfradar.Tracker(detection,current_time,trackid);

                rng_est = detection.Measurement(3);
                if rng_est >= 50e3
                    updateinterval = 0.5;
                else
                    updateinterval = 0.1;
                end

                revisit_time = current_time+updateinterval;
                predictedTrack = predictTracksToTime(mfradar.Tracker,trackid,revisit_time);
                xpred = predictedTrack.State([1 3 5]);
                [phipred,thetapred,rpred] = cart2sph(xpred(1),xpred(2),xpred(3));
                num_trackq_items = num_trackq_items+1;
                trackq(num_trackq_items) = struct('JobType','Track','Priority',3000,...
                    'BeamDirection',rad2deg([phipred;thetapred]),'WaveformIndex',1,'Time',revisit_tir
                    'Range',rpred,'TrackID',trackid);

                if current_time < 0.3 || strcmp(current_job.JobType,'Track')
                    fprintf('\tTrack %d at %f m',trackid,rng_est);
```

```matlab
            end
        else
            trackid = current_job.TrackID;
            [~,~,allTracks] = mfradar.Tracker(detection,current_time,trackid);

            updateinterval = 0.1;  % revisit sooner
            revisit_time = current_time+updateinterval;
            predictedTrack = predictTracksToTime(mfradar.Tracker,trackid,revisit_time);
            xpred = predictedTrack.State([1 3 5]);

            [phipred,thetapred,rpred] = cart2sph(xpred(1),xpred(2),xpred(3));
            num_trackq_items = num_trackq_items+1;
            trackq(num_trackq_items) = struct('JobType','Track','Priority',3000,...
                'BeamDirection',rad2deg([phipred;thetapred]),'WaveformIndex',1,'Time',revisit_ti
                'Range',rpred,'TrackID',trackid);

            if current_time < 0.3 || strcmp(current_job.JobType,'Track')
                fprintf('\tNo detection, track %d predicted',current_job.TrackID);
            end
        end

    end

    jobq.TrackQueue   = trackq;
    jobq.NumTrackJobs = num_trackq_items;
```

How to Generate C Code for a Tracker

This example shows how to generate C code for a MATLAB function that processes detections and outputs tracks. The function contains a `trackerGNN`, but any tracker can be used instead.

Automatic generation of code from MATLAB code has two key benefits:

1 Prototypes can be developed and debugged in the MATLAB environment. Once the MATLAB work is done, automatic C code generation makes the algorithms deployable to various targets. Additionally, the C code can be further tested by running the compiled MEX file in a MATLAB environment using the same visualization and analysis tools that were available during the prototyping phase.

2 After generating C code, you can generate executable code, which in many cases runs faster than the MATLAB code. The improved run time can be used to develop and deploy real-time sensor fusion and tracking systems. It also provides a better way to batch test the tracking systems on a large number of data sets.

The example explains how to modify the MATLAB code in the "Air Traffic Control" on page 5-2 example to support code generation. This example requires a MATLAB Coder license for generating C code.

Modify and Run MATLAB Code

To generate C code, MATLAB Coder requires MATLAB code to be in the form of a function. Furthermore, the arguments of the function cannot be MATLAB classes.

In this example, the code for the air traffic control (ATC) example has been restructured such that the `trackerGNN` that performs sensor fusion and tracking resides in a separate file, called `tracker_kernel.m`. Review this file for important information about memory allocation for code generation.

To preserve the state of the `trackerGNN` between calls to `tracker_kernel.m`, the tracker is defined as a `persistent` variable.

This function takes a cell array of `objectDetection` objects, generated by the `monostaticRadarSensor` object, and time as input arguments.

Similarly, the outputs from a function that supports code generation cannot be objects. The outputs from `tracker_kernel.m` are:

1 Confirmed tracks - A `struct` array that contains a variable number of tracks.
2 Number of tracks - An integer scalar.
3 Information about the tracker processing at the current update.

By restructuring the code this way, you can reuse the same display tools used in the ATC example. These tools still run in MATLAB and do not require code generation.

```
% If a previous tracker is defined, clear it.
clear tracker_kernel

% Create the ATC scene with radar and platforms.
[scenario,tower,radar] = helperCreateATCScenario;

% Create a display to show the true, measured, and tracked positions of the
```

```
% airliners.
[theater,fig] = helperTrackerCGExample('Create Display',scenario);
helperTrackerCGExample('Update Display',theater,scenario,tower);
```
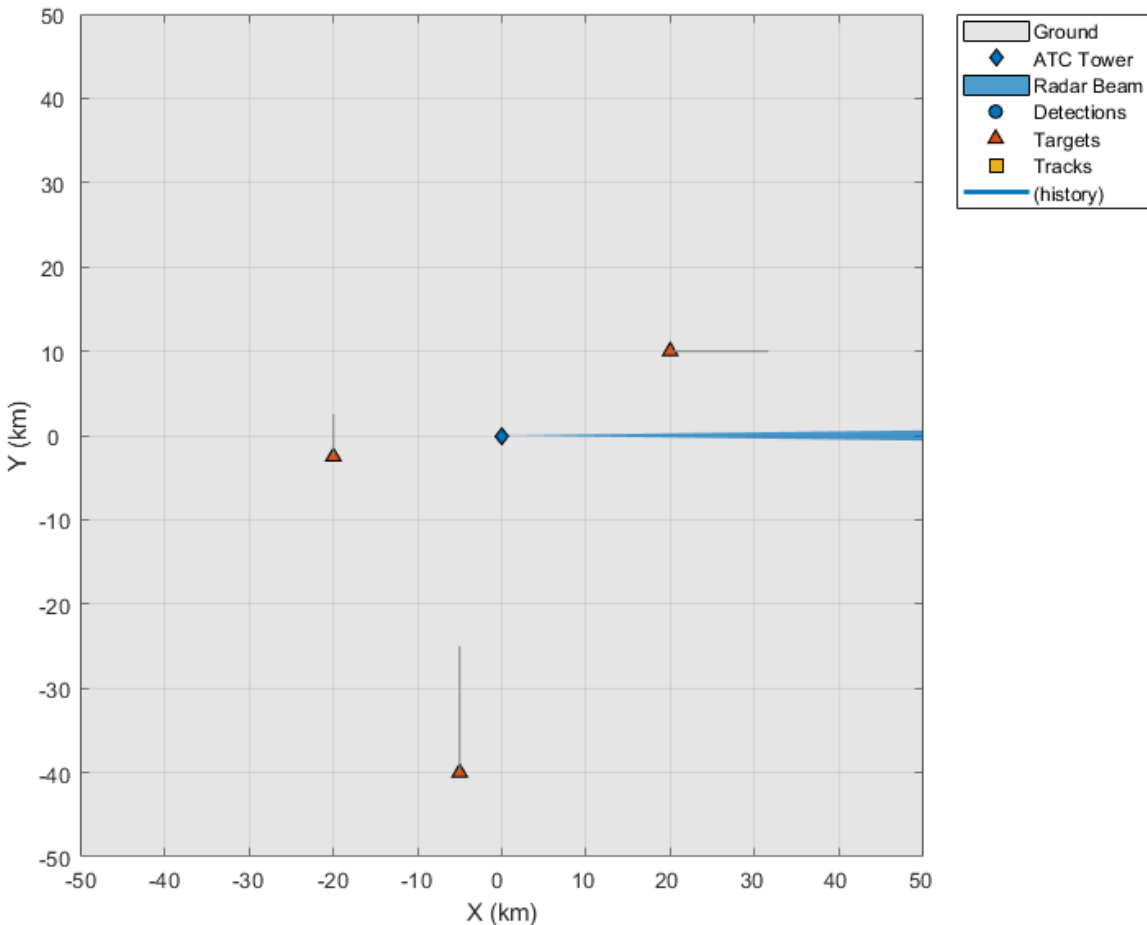

Now run the example by calling the `tracker_kernel` function in MATLAB. This initial run provides a baseline to compare the results and enables you to collect some metrics about the performance of the tracker when it runs in MATLAB or as a MEX file.

Simulate and Track Airliners

The following loop advances the platform positions until the end of the scenario. For each step forward in the scenario, the radar generates detections from targets in its field of view. The tracker is updated with these detections after the radar has completed a 360 degree scan in azimuth.

```
% Set simulation to advance at the update rate of the radar.
scenario.UpdateRate = radar.UpdateRate;

% Create a buffer to collect the detections from a full scan of the radar.
scanBuffer = {};

% Initialize the track array.
```

```matlab
tracks = [];

% Set random seed for repeatable results.
rng(2020)

% Allocate memory for number of tracks and time measurement in MATLAB.
numSteps  = 12;
numTracks = zeros(1, numSteps);
runTimes  = zeros(1, numSteps);
index = 0;
while advance(scenario) && ishghandle(fig)

    % Generate detections on targets in the radar's current field of view.
    [dets,config] = detect(scenario);

    scanBuffer = [scanBuffer;dets]; %#ok<AGROW> Allow the buffer to grow.

    % Update tracks when a 360 degree scan is complete.
    if config.IsScanDone
        % Update tracker
        index = index + 1;
        tic
        [tracks, numTracks(index), info] = tracker_kernel(scanBuffer,scenario.SimulationTime);
        runTimes(index) = toc; % Gather MATLAB run time data

        % Clear scan buffer for next scan.
        scanBuffer = {};
    end

    % Update display with current beam position, buffered detections, and
    % track positions.
    helperTrackerCGExample('Update Display',theater,scenario,tower,scanBuffer,tracks);
end
```
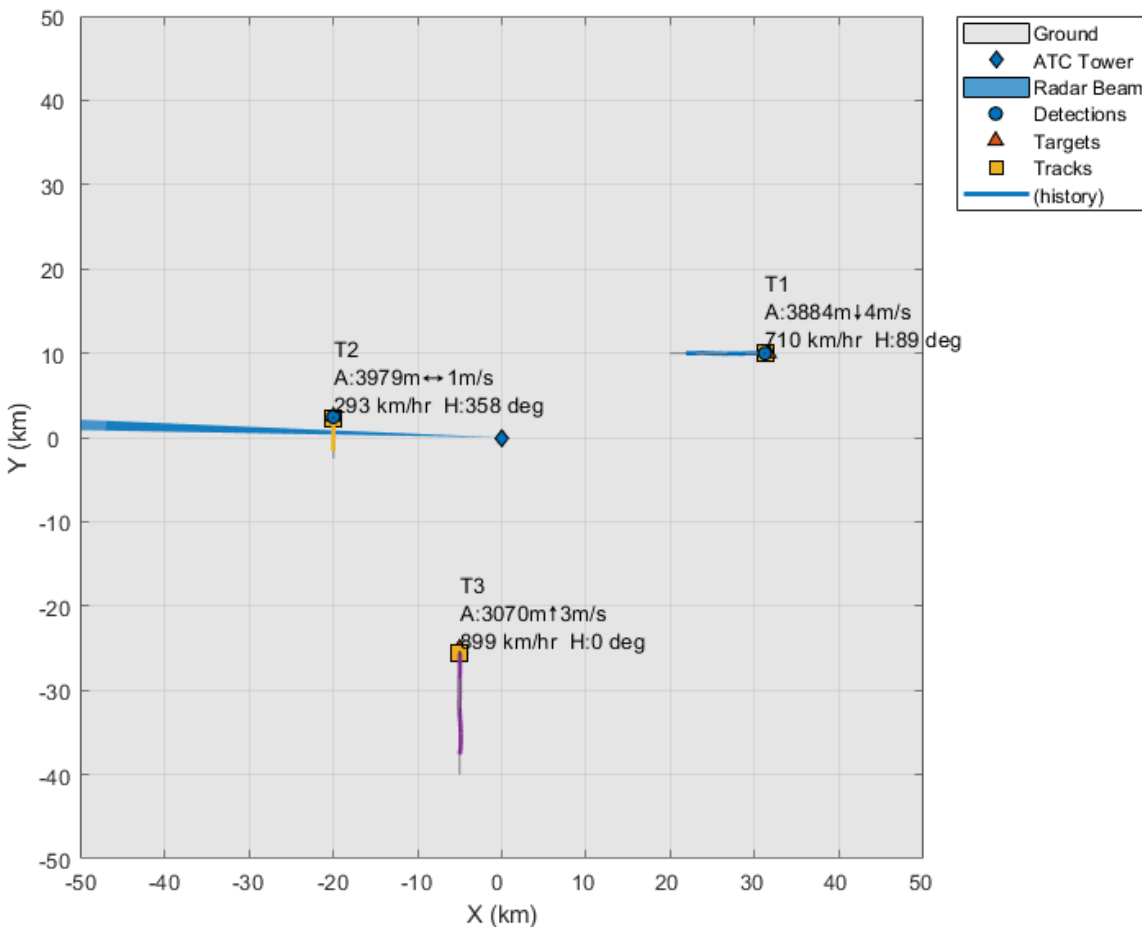
Compile the MATLAB Function into a MEX File

Use the `codegen` function to compile the `tracker_kernel` function into a MEX file. You can specify the `-report` option to generate a compilation report that shows the original MATLAB code and the associated files that were created during C code generation. Consider creating a temporary directory where MATLAB Coder can store generated files. Note that unless you use the `-o` option to specify the name of the executable, the generated MEX file has the same name as the original MATLAB file with `_mex` appended.

MATLAB Coder requires that you specify the properties of all the input arguments. The inputs are used by the tracker to create the correct data types and sizes for objects used in the tracking. The data types and sizes must not change between data frames. One easy way to do this is to define the input properties by example at the command line using the `-args` option. For more information, see "Input Specification" (MATLAB Coder).

```
% Define the properties of the input. First define the detections buffer as
% a variable-sized cell array that contains objectDetection objects. Then
% define the second argument as simTime, which is a scalar double.
dets = coder.typeof(scanBuffer(1), [inf 1], [1 0]);
compInputs  = {dets scenario.SimulationTime};
```

```
% Code generation may take some time.
h = msgbox({'Generating code. This may take a few minutes...';'This message box will close when
% Generate code.
try
    codegen tracker_kernel -args compInputs;
    close(h)
catch ME
    close(h)
    delete(videoDisplayHandle.Parent.Parent)
    throw(ME)
end
```

Run the Generated Code

Now that the code has been generated, run the exact same scenario with the generated MEX file
`tracker_kernel_mex`. Everything else remains the same.

```
% If a previous tracker is defined, clear it.
clear tracker_kernel_mex

% Allocate memory for number of tracks and time measurement
numTracksMex = zeros(1, numSteps);
runTimesMex  = zeros(1, numSteps);

% Reset the scenario, data counter, plotters, scanBuffer, tracks, and rng.
index = 0;
restart(scenario)
scanBuffer = {};
clearPlotterData(theater);
tracks = [];
rng(2020)
while advance(scenario) && ishghandle(fig)

    % Generate detections on targets in the radar's current field of view.
    [dets,config] = detect(scenario);

    scanBuffer = [scanBuffer;dets]; %#ok<AGROW> Allow the buffer to grow.

    % Update tracks when a 360 degree scan is complete.
    if config.IsScanDone
        % Update tracker.
        index = index + 1;
        tic
        [tracks, numTracksMex(index), info] = tracker_kernel_mex(scanBuffer,scenario.SimulationT
        runTimesMex(index) = toc; % Gather MEX run time data

        % Clear scan buffer for next scan.
        scanBuffer = {};
    end

    % Update display with current beam position, buffered detections, and
    % track positions.
    helperTrackerCGExample('Update Display',theater,scenario,tower,scanBuffer,tracks);
end
```
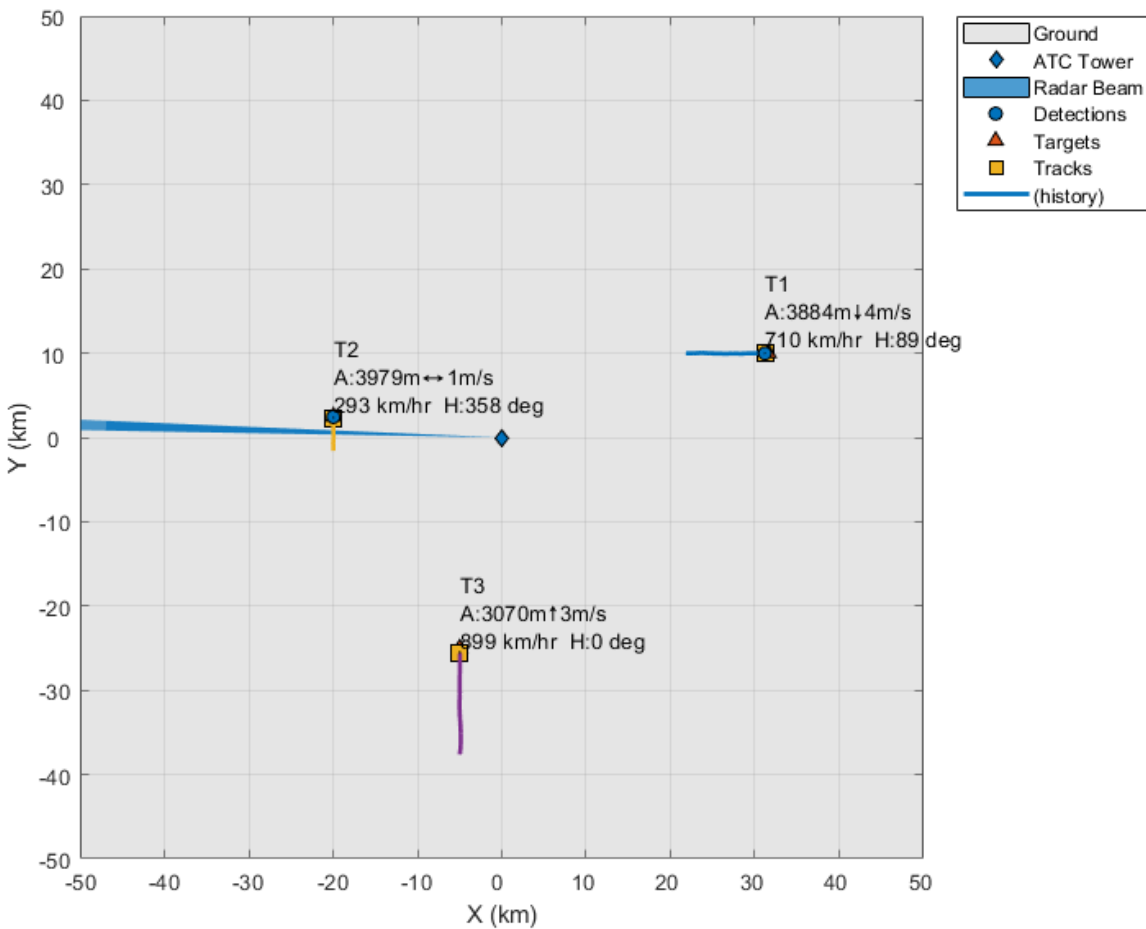
Compare the Results of the Two Runs

Compare the results and the performance of the generated code vs. the MATLAB code. The following plots compare the number of tracks maintained by the trackers at each time step. They also show the amount of time it took to process each call to the function.

```
figure(2)
subplot(2,1,1)
plot(2:numSteps, numTracks(2:numSteps), 's:', 2:numSteps, numTracksMex(2:numSteps), 'x-.')
title('Number of Tracks at Each Step');
legend('MATLAB', 'MEX')
grid
subplot(2,1,2)
plot(2:numSteps, runTimesMex(2:numSteps)*1e3);
title('MEX Processing Time at Each Step')
grid
xlabel('Time Step')
ylabel('MEX Processing Time [ms]')
```
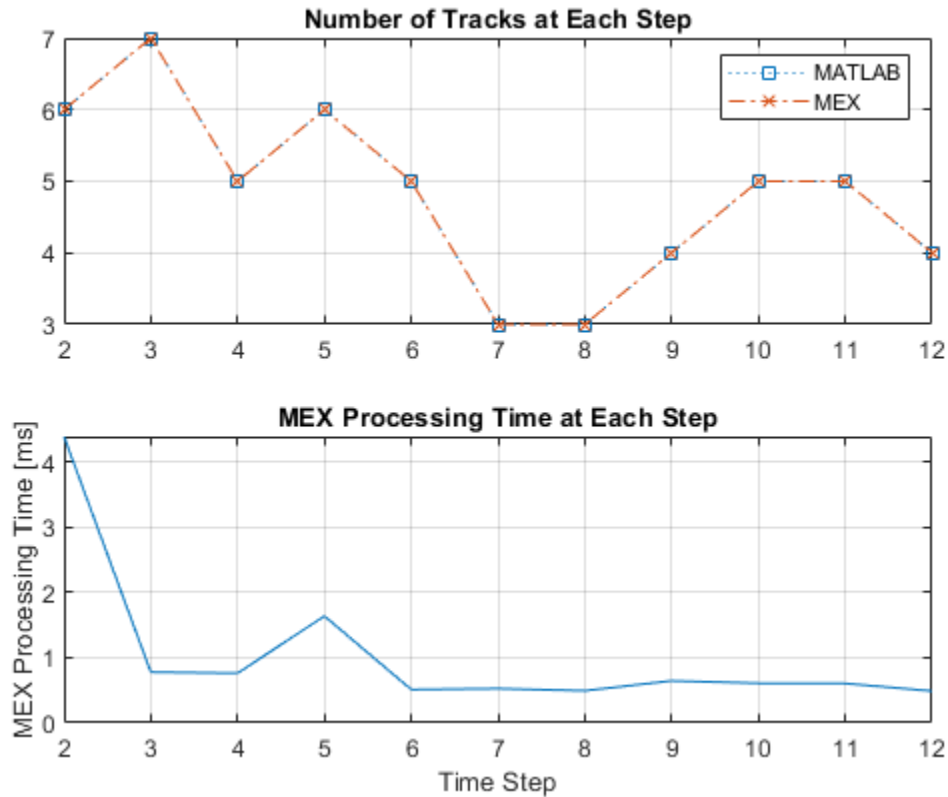
The top plot shows that the number of tracks that were maintained by each tracker are the same. It measures the size of the tracking problem in terms of number of tracks. Even though there were 3 confirmed tracks throughout the tracking example, the total number of all tracks maintained by the tracker varies based on the number of tentative tracks, that were created by false detections.

The bottom plot shows the time required for the generated code function to process each step. The first step was excluded from the plot, because it takes a disproportionately longer time to instantiate all the tracks on the first step.

The results show the number of milliseconds required by the MEX code to perform each update step on your computer. In this example, the time required for the MEX code to run an update step is measured in a few milliseconds.

Summary

This example showed how to generate C code from MATLAB code for sensor fusion and tracking.

The main benefits of automatic code generation are the ability to prototype in the MATLAB environment, and generate a MEX file that can run in the MATLAB environment. The generated C code can be deployed to a target. In most cases, the generated code is faster than the MATLAB code, and can be used for batch testing of algorithms and generating real-time tracking systems.

How to Efficiently Track Large Numbers of Objects

This example shows how to use the `trackerGNN` to track large numbers of targets. Similar techniques can be applied to the `trackerJPDA` and `trackerTOMHT` as well.

Introduction

In many applications, trackers are required to track hundreds or thousands of objects. Increasing the number of tracks maintained by a tracker is a challenge, caused by the computational complexity of the algorithm at the core of every tracker. In particular, two common stages in the tracker update step are not easily scalable: calculating the assignment cost and performing the assignment. Assignment cost calculation is common to `trackerGNN`, `trackerJPDA`, and `trackerTOMHT`, and the techniques shown in this example can be applied when using any of these trackers. The way each tracker performs the assignment is unique to each tracker, and may require tailored solutions to improve the tracker performance, which are beyond the scope of this example.

Scenario

For the purposes of this example, you define a scenario that contains 900 platforms, organized in a 15-by-15 grid, with each grid cell containing 4 platforms. The purpose of the grid cells is to demonstrate the benefit of coarse cost calculation, explained later in the example.

The following code arranges the 900 objects in the grid cell and creates the visualization. On the left, the entire scenario is shown. On the right, the visualization zooms in on 4 grid cells. Note each cell contains 4 platforms.

```
[platforms,tp,zoomedtp] = createPlatforms;
```
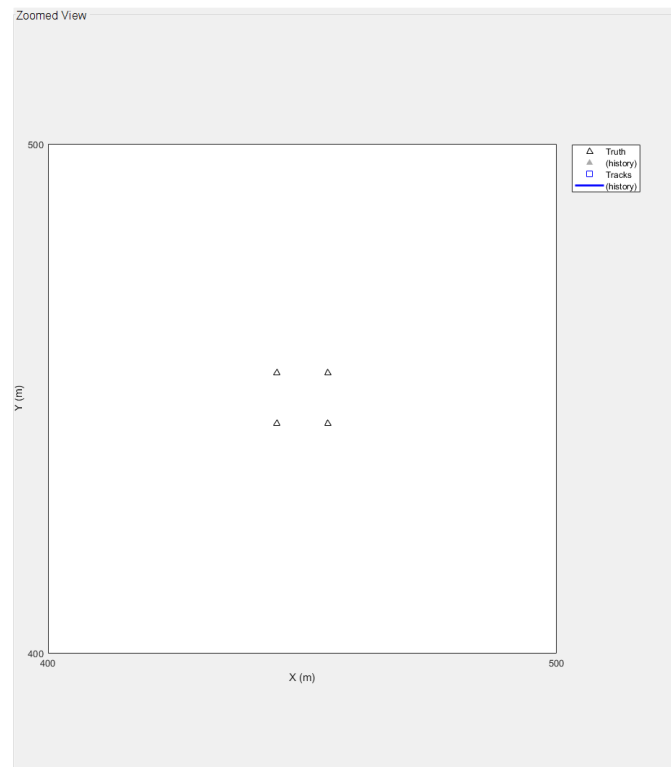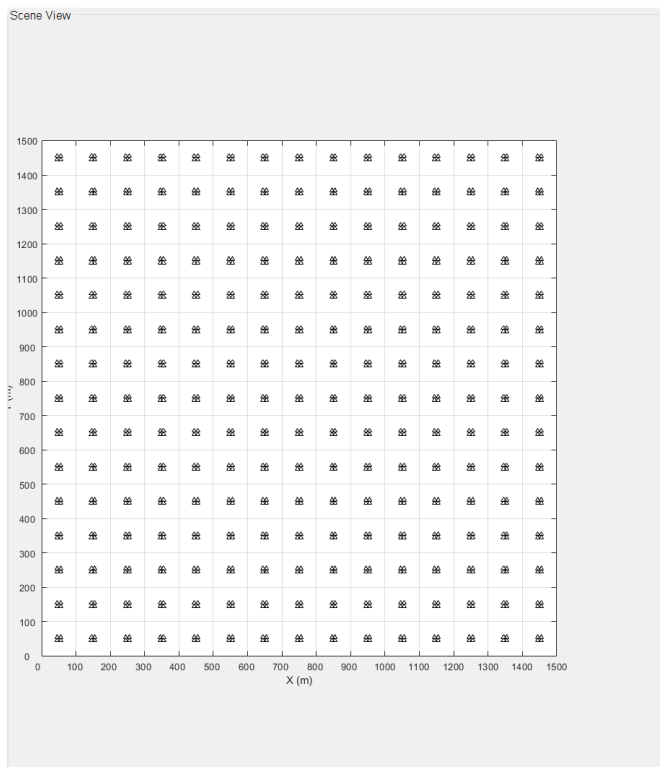
Use the Default Assignment Cost Calculation

This section shows the results of tracking the platforms defined above using a `trackerGNN` with default `AssignmentThreshold`. The `AssignmentThreshold` property contains two values: `[C1 C2]`, where `C1` is the threshold used for the assignment and `C2` is a threshold for coarse calculation explained in the next section.

When the tracker is updated with a new set of detections, it calculates the cost of assigning every detection to every track. The accurate cost calculation must take into account the measurement and uncertainty of each detection as well as the expected measurement and expected uncertainty from each track, as depicted below.

For each combination of track and detection:

$$y = z - z_e = z - h(x)$$
$$S = R + HPH', H = \frac{\partial h}{\partial x}$$
$$d^2 = y'S^{-1}y + \log(|S|)$$

For assignment: $d^2 \leq d^2_{max}$

z_{e_1}

Predicted Track in
Measurement Space +
Predicted Uncertainty

z

Measurement +
Measurement Noise

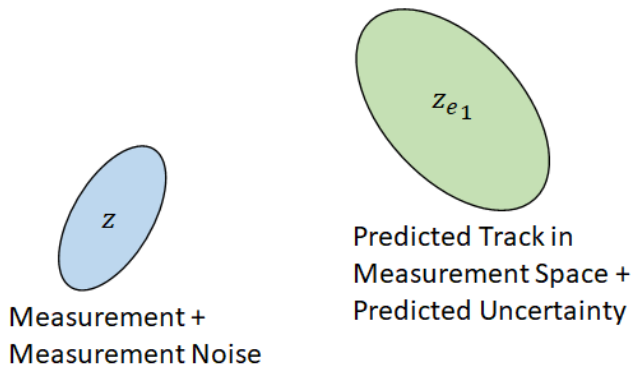By default, `C2` is set to Inf, which requires that the costs of all combinations of track and detection are calculated. This leads to a more accurate assignment, but is more computationally intensive. You should start with the default setting to make sure the tracker assigns detections to tracks in the best way, and then consider lowering the value of `C2` to reduce the time required for calculating the assignment cost.

During assignment cost calculation, elements of the cost matrix whose values are higher than `C1` are replaced with Inf. Doing so helps the assignment algorithm to ignore impossible assignments.

Define a tracker that can track up to 1000 tracks. The tracker uses the default constant velocity extended Kalman filter, and its state is defined as `[x;vx;y;vy;z;vz]`, which is used by the `positionSelector` below to get the position components.

```
tracker = trackerGNN('MaxNumTracks',1000, 'AssignmentThreshold', [30 Inf]);
positionSelector = [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 0 0];
```

On the first call to step, the tracker instantiates all tracks. To isolate the time required to instantiate the tracks from the processing time required for the first step, you can call `setup` and `reset` before stepping the tracker. See the supporting function `runTracker` at the end of this example for more details.

```
[trkSummary,truSummary,info] = runTracker(platforms,tracker,positionSelector,tp,zoomedtp);
```

```
Tracker set up time: 8.3108
Step 1 time: 3.7554
Step 2 time: 15.3029
Step 3 time: 14.1099
Step 4 time: 14.3506
Step 5 time: 14.3963
```
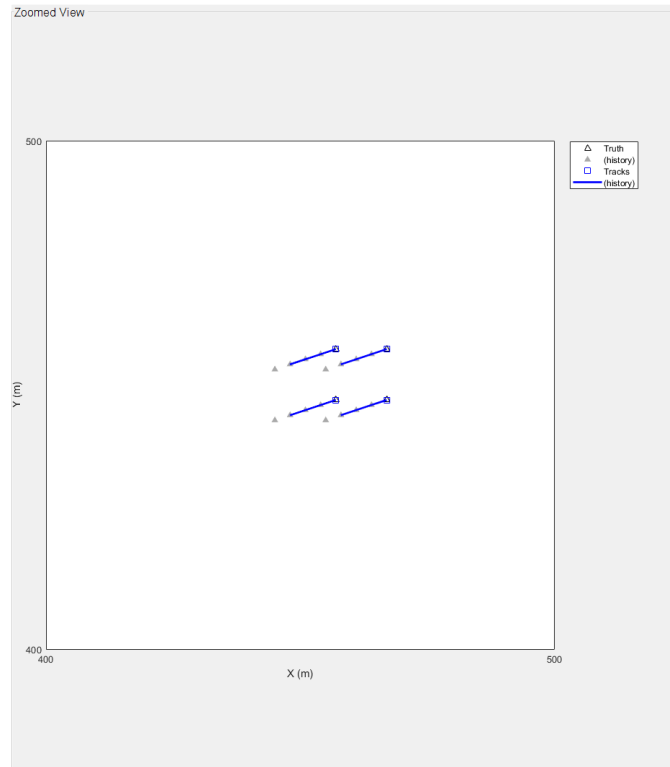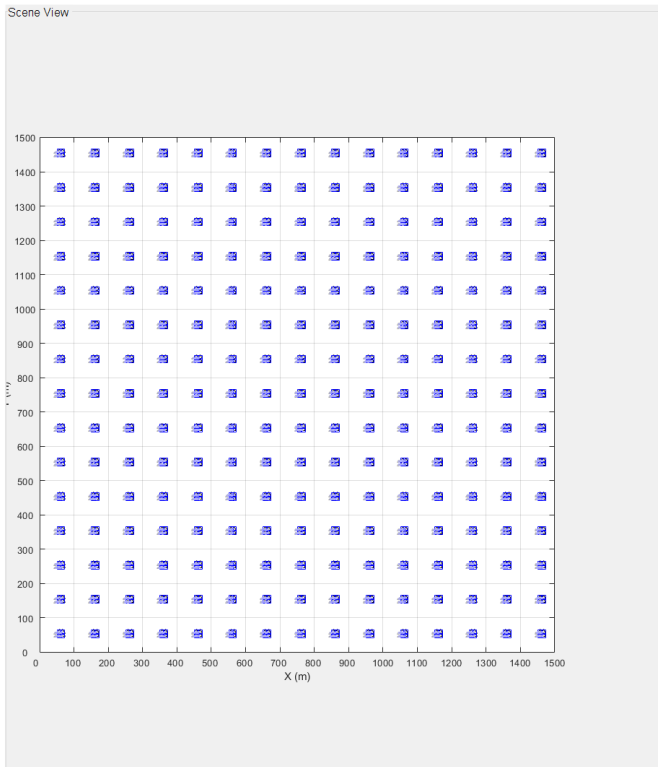
```
All steps from now are without detections.
Step 6 time: 0.53103
Step 7 time: 0.52582
Step 8 time: 0.50639
Step 9 time: 0.50909
Step 10 time: 0.16034
Scenario done. All tracks are now deleted.
```

You analyze the tracking results by examining the track assignment metrics values. For perfect tracking the total number of tracks should be equal to the number of platforms and there should be no false, swapped, or divergent tracks. Similarly, there should be no missing truth or breaks in the truth summary.

`assignmentMetricsSummary(trkSummary,truSummary)`

```
Track assignment metrics summary:
```

TotalNumTracks	NumFalseTracks	MaxSwapCount	MaxDivergenceCount	MaxDivergenceLength
900	0	0	0	0

```
Truth assignment metrics summary:
```

TotalNumTruths	NumMissingTruths	MaxEstablishmentLength	MaxBreakCount
900	0	1	0

Use Coarse Assignment Cost Calculation

In the previous section, you saw that the tracker is able to track all the platforms, but every update step takes a long time. Most of the time was spent on calculating the assignment cost matrix.

Examining the cost matrix, you can see that vast majority of its elements are, in fact, Inf.

```
cm = info.CostMatrix;
disp("Cost matrix has " + numel(cm) + " elements.");
disp("But the number of finite values is " + numel(cm(isfinite(cm))) + newline)
```

```
Cost matrix has 810000 elements.
But the number of finite values is 2700
```

The above result shows that the cost calculation spends too much time on calculating the assignment cost of all the combinations of track and detection. However, most of these combinations are too far to be assigned, as the actual measurement is too far from the track expected measurement based on the sensor characteristics. To avoid the waste in calculating all the costs, you can use coarse cost calculation.

Coarse calculation is done to verify which combinations of track and detection may require an accurate normalized distance calculation. Only combinations whose coarse assignment cost is lower than C2 are calculated accurately. The coarse cost calculation is depicted in the image below. A detection is represented by its measurement z and measurement noise R. Two tracks are predicted to the time of the detection and projected to the measurement space, depicted by the points z_{e_1} and z_{e_2}. Note that the track uncertainty is not projected to the measurement space, which allows us to vectorize the coarse calculation. This is a rough estimate, because only the uncertainty around the detection is taken into account. In the depicted example, the first track falls outside of the coarse calculation gate while the second track falls inside it. Thus, accurate cost calculation is only done for the combination of this detection and the second track.
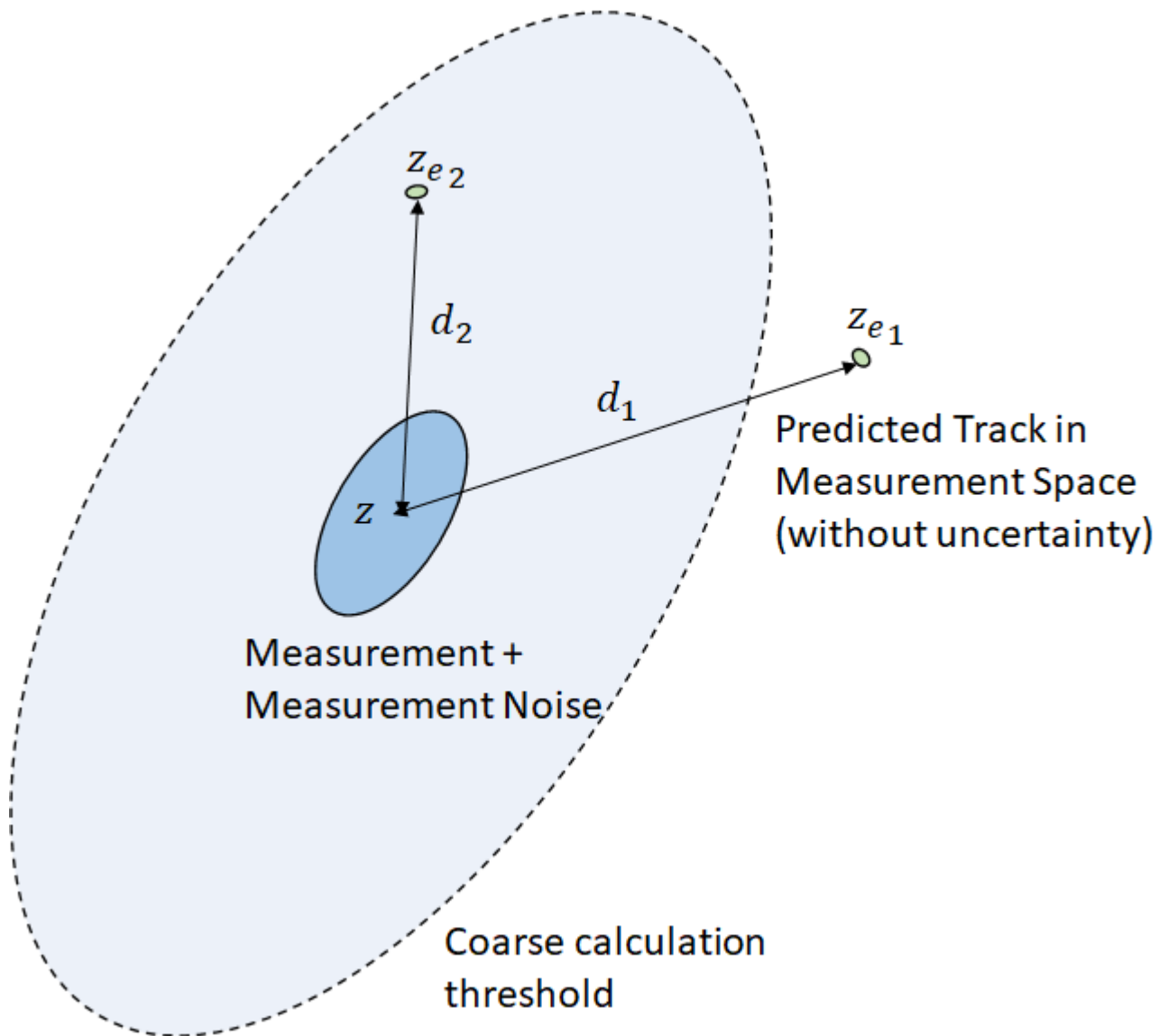
To use coarse cost calculation, release the tracker and modify its `AssignmentThreshold` to a value of [30 200]. Then, rerun the tracker.

```
release(tracker)
tracker.AssignmentThreshold = [30 200];
[trkSummary,truSummary] = runTracker(platforms,tracker,positionSelector,tp,zoomedtp);
```

```
Tracker set up time: 6.5846
Step 1 time: 3.5863
Step 2 time: 3.4095
Step 3 time: 2.9347
Step 4 time: 2.8555
Step 5 time: 2.9397
```
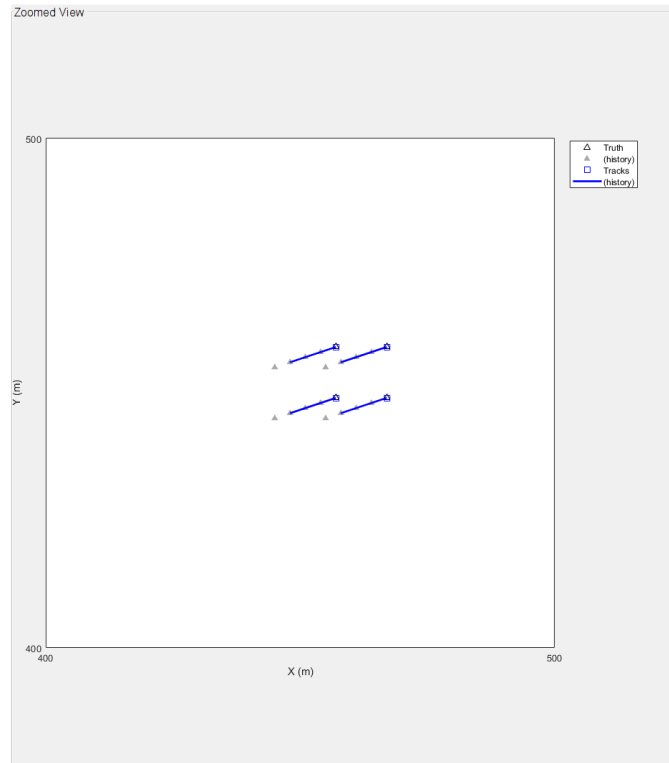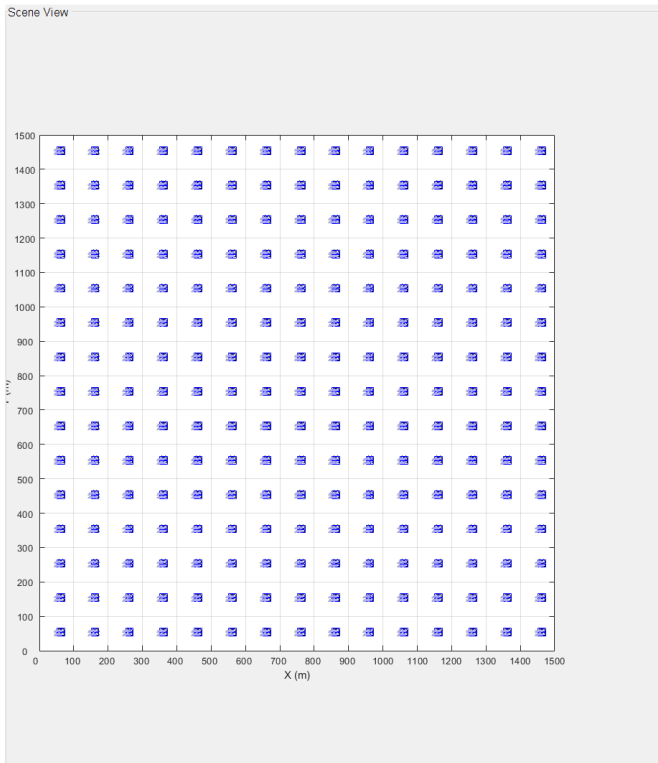
```
All steps from now are without detections.
Step 6 time: 0.51446
Step 7 time: 0.52277
Step 8 time: 0.54865
Step 9 time: 0.50941
Step 10 time: 0.19085
Scenario done. All tracks are now deleted.
```

You observe that the steps 3-5 now require significantly less time to complete. Step 2 is also faster than it used to be, but still slower than steps 3-5.

To understand why step 2 is slower, consider the track states after the first tracker update. The states contain position information, but the velocity is still zero. When the tracker calculates the assignment cost, it predicts the track states to the detection times, but since the tracks have zero velocity, they remain in the same position. This results in large distances between the detection measurements and the expected measurements from the predicted track states. These relatively large assignment costs make it harder for the assignment algorithm to find the best assignment, which causes step 2 to take more time than steps 3-5.

It's important to verify that the track assignment with coarse cost calculation remains the same as without it. If the track assignment metrics are not the same, you must increase the size of the coarse calculation gate. The following shows that the tracking is still perfect as it was in the previous section, but each processing step took less time.

```
assignmentMetricsSummary(trkSummary,truSummary)
```

```
Track assignment metrics summary:
    TotalNumTracks    NumFalseTracks    MaxSwapCount    MaxDivergenceCount    MaxDivergenceLengtl
```

_____	_____	_____	_____	_____
900	0	0	0	0

Truth assignment metrics summary:

TotalNumTruths	NumMissingTruths	MaxEstablishmentLength	MaxBreakCount
_____	_____	_____	_____
900	0	1	0

Use an External Cost Calculation

Another way to control the time it takes to calculate the cost assignment is by using your own assignment cost calculation instead of the default the tracker uses.

An external cost calculation can take into account attributes that are not part of the track state and expected measurement. It can also use different distance metrics, for example Euclidean norm instead of normalized distance. The choice of which cost calculation to apply depends on the specifics of the problem, the measurement space, and how you define the state and measurement.

To use an external cost calculation, you release the tracker and set its `HasCostMatrixInput` property to true. You must pass your own cost matrix as an additional input with each update to the tracker. See the supporting function `runTracker` for more details.

```
release(tracker);
tracker.HasCostMatrixInput = true;
[trkSummary,truSummary] = runTracker(platforms,tracker,positionSelector,tp,zoomedtp);
assignmentMetricsSummary(trkSummary,truSummary)
```

```
Tracker set up time: 6.559
Step 1 time: 3.4394
Step 2 time: 1.7852
Step 3 time: 1.474
Step 4 time: 1.5312
Step 5 time: 1.5152
```
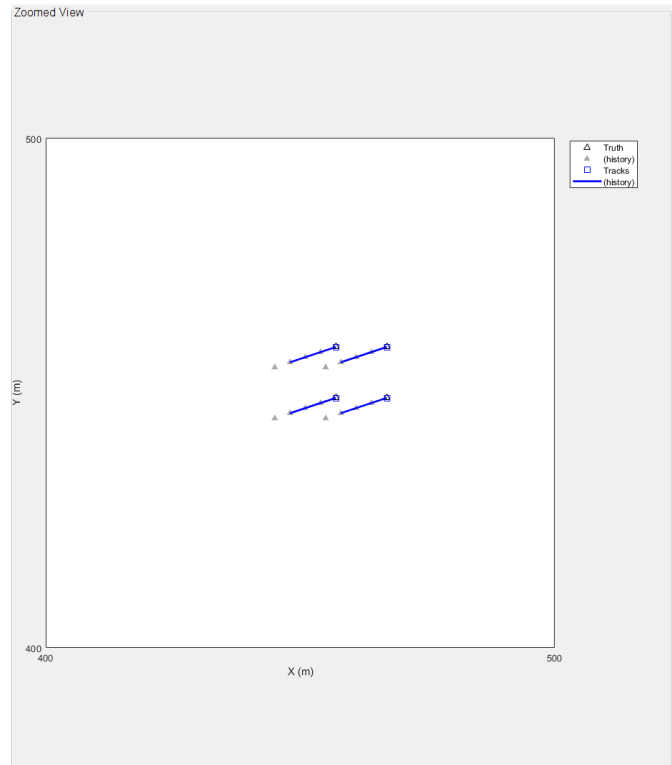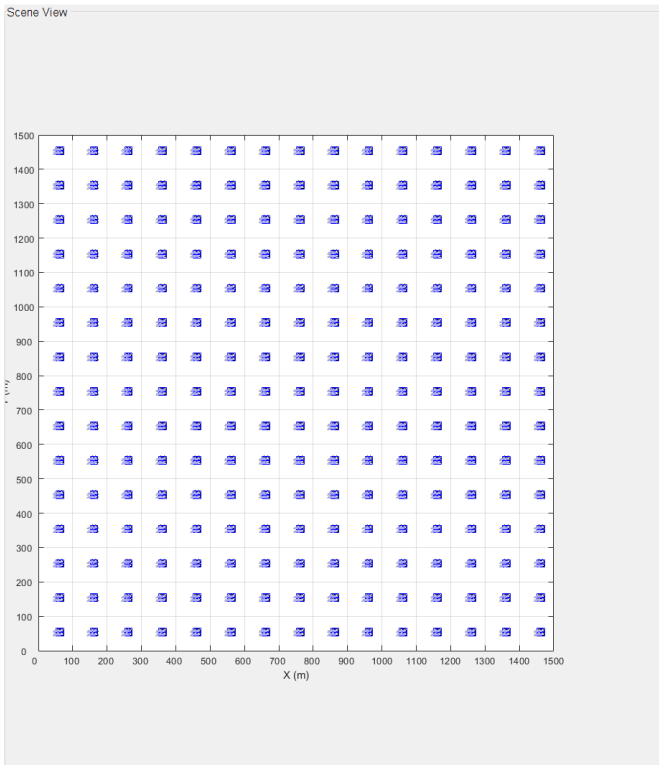
```
All steps from now are without detections.
Step 6 time: 0.60809
Step 7 time: 0.61374
Step 8 time: 0.616
Step 9 time: 0.63798
Step 10 time: 0.22762
Scenario done. All tracks are now deleted.

Track assignment metrics summary:
```

TotalNumTracks	NumFalseTracks	MaxSwapCount	MaxDivergenceCount	MaxDivergenceLength
900	0	0	0	0

```
Truth assignment metrics summary:
```

TotalNumTruths	NumMissingTruths	MaxEstablishmentLength	MaxBreakCount
900	0	1	0

As expected, the processing time is even lower when using an external cost calculation function.

Change the GNN Assignment Algorithm

Another option to try is using a different GNN assignment algorithm that may be more efficient in finding the assignment by modifying the Assignment property of the tracker.

```
release(tracker)
tracker.Assignment = 'Jonker-Volgenant';
```

```
tracker.HasCostMatrixInput = true;
runTracker(platforms,tracker,positionSelector,tp,zoomedtp);
```

```
Tracker set up time: 6.494
Step 1 time: 3.5346
Step 2 time: 1.894
Step 3 time: 3.1192
Step 4 time: 3.1212
Step 5 time: 3.1458
```
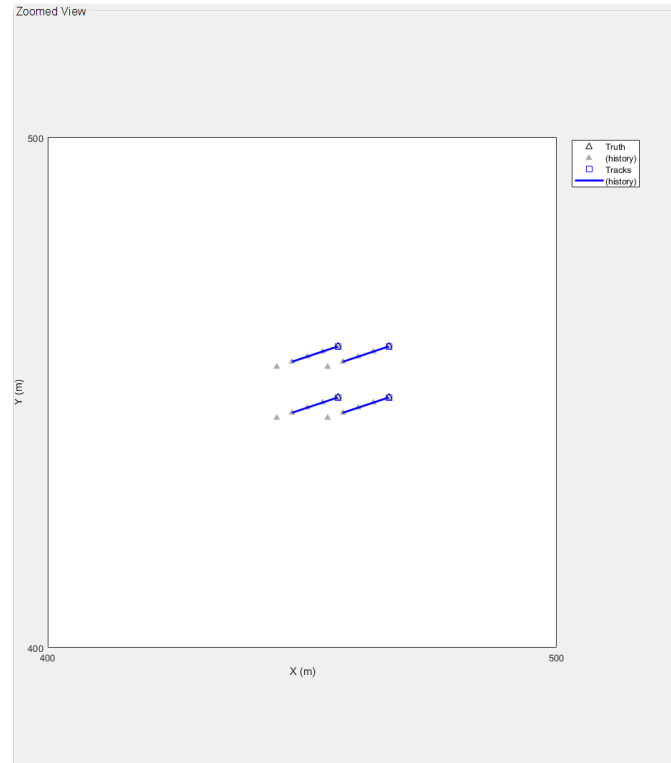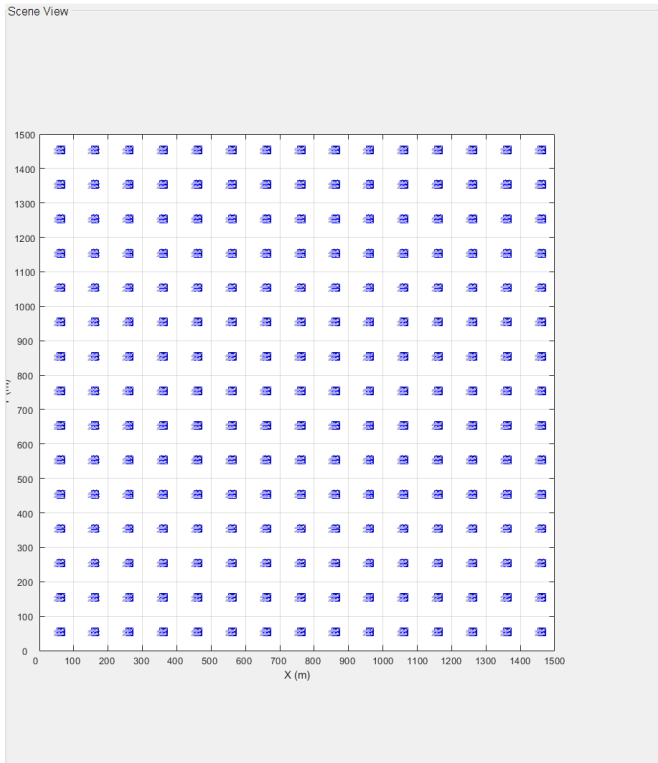


```
All steps from now are without detections.
Step 6 time: 0.61109
Step 7 time: 0.62456
Step 8 time: 0.61849
Step 9 time: 0.60604
Step 10 time: 0.22303
Scenario done. All tracks are now deleted.
```

The Jonker-Volgenant algorithm performs the assignment in the second step faster relative to the default Munkres algorithm.

Monte-Carlo Simulation

If you want to run multiple scenarios without modifying the tracker settings, there is no need to call the `release` method. Instead, just call the `reset` method to clear previous track information from the tracker. This way, you save the time required to instantiate all the tracks. Note the "Tracker set up time" below relative to previous runs.

```
reset(tracker)
runTracker(platforms,tracker,positionSelector,tp,zoomedtp);
```

```
Tracker set up time: 0.097531
Step 1 time: 3.4684
Step 2 time: 1.6592
Step 3 time: 3.1429
Step 4 time: 3.1274
Step 5 time: 3.0994
```
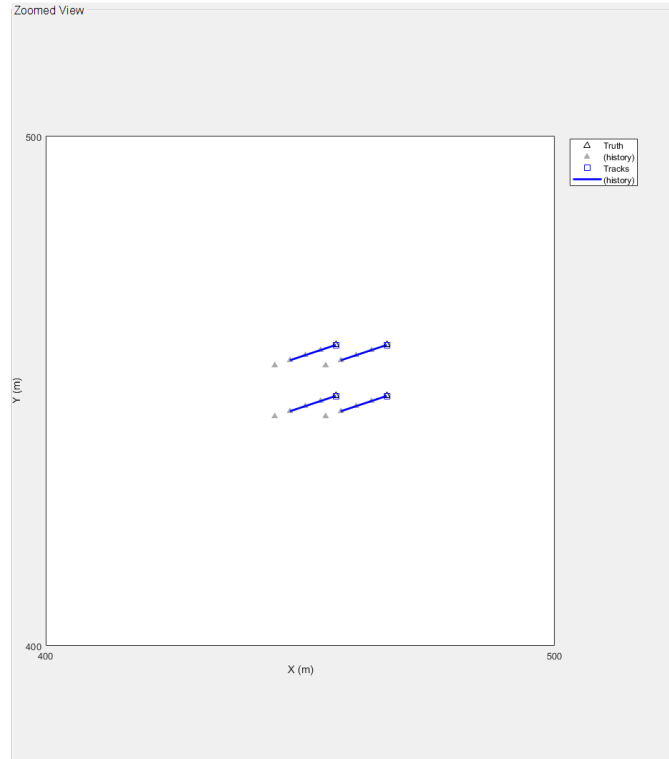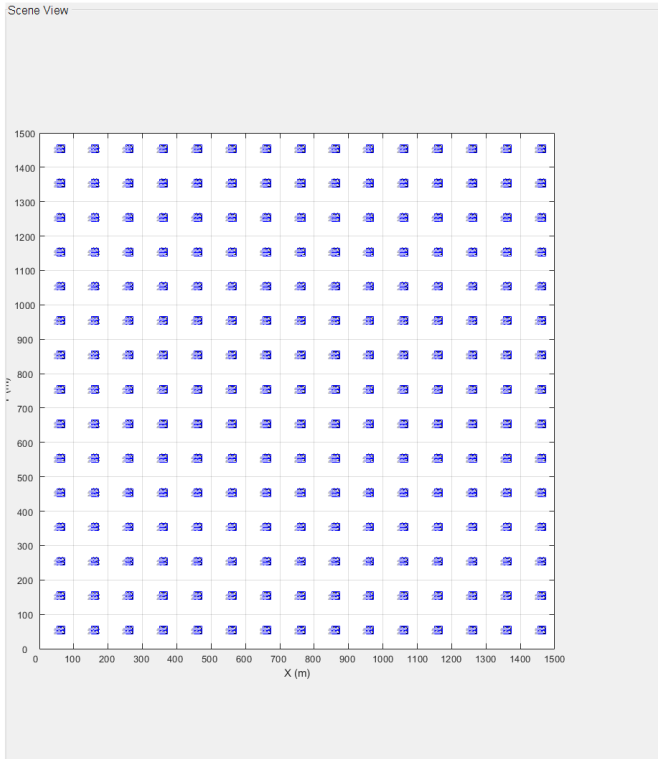


```
All steps from now are without detections.
Step 6 time: 0.63232
Step 7 time: 0.61857
Step 8 time: 0.61433
Step 9 time: 0.60698
Step 10 time: 0.25301
Scenario done. All tracks are now deleted.
```

Summary

This example showed how to track large numbers of objects. When tracking many objects, the tracker spends a large fraction of the processing time on computing the cost assignment for each combination of track and detection. You saw how to use the cost calculation threshold to improve the time spent on calculating the assignment cost. In addition, the example showed how to use an external cost calculation, which may be designed to be more computationally efficient for the particular tracking problem you have.

You can reduce the cost assignment threshold or use an external cost calculation to improve the speed of the `trackerJPDA` and the `trackerTOMHT` as well.

Supporting Functions

createPlatforms

This function creates the platforms in a 20x20 grid with 2x2 platforms per grid cell.

```matlab
function [platforms,tp,zoomedtp] = createPlatforms
% This is a helper function to run the tracker and display the results. It
% may be removed in the future.
nh  = 15; % Number of horizontal grid cells
nv  = 15; % Number of vertical grid cells
nsq = 2;  % 2x2 platforms in a grid cell
nPl = nh*nv*nsq^2; % Overall number of platforms
xgv = sort(-50 + repmat(100 * (1:nh), [1 nsq]));
ygv = sort(-50 + repmat(100 * (1:nv), [1 nsq]));
[X,Y] = meshgrid(xgv,ygv);

npts = nsq/2;
xshift = 10*((-npts+1):npts) -5;
yshift = xshift;

xadd = repmat(xshift, [1 nh]);
yadd = repmat(yshift, [1 nv]);

[Xx, Yy] = meshgrid(xadd,yadd);

X = X + Xx;
Y = Y + Yy;
pos = [X(:),Y(:),zeros(numel(X),1)];

% The following creates an array of struct for the platforms, which are
% used later for track assignment metrics.
vel = [3 1 0]; % Platform velocity
platforms = repmat(struct('PlatformID', 1, 'Position', [0 0 0], 'Velocity', vel),nPl,1);
for i = 1:nPl
    platforms(i).PlatformID  = i;
    platforms(i).Position(:) = pos(i,:);
end

% Visualization
f = figure('Position',[1 1 1425 700]);
movegui center;
h1 = uipanel(f,'FontSize',12,'Position',[.01 .01 .48 .98],"Title","Scene View");
a1 = axes(h1,'Position',[0.05 0.05 0.9 0.9]);
tp = theaterPlot('Parent', a1, 'XLimits',[0 nh*100], 'YLimits',[0 nv*100]);
set(a1,'XTick',0:100:nh*100)
set(a1,'YTick',0:100:nv*100)
grid on
pp = trackPlotter(tp,'Tag','Truth','Marker','^','MarkerEdgeColor','k','MarkerSize',4,'HistoryDept
plotTrack(pp,reshape([platforms.Position],3,[])');
trackPlotter(tp, 'Tag','Tracks','MarkerEdgeColor','b','MarkerSize',6,'HistoryDepth',10);
c = get(a1.Parent,'Children');
for i = 1:numel(c)
    if isa(c(i),'matlab.graphics.illustration.Legend')
        set(c(i),'Visible','off')
    end
end
```

```matlab
    h2 = uipanel(f,'FontSize',12,'Position',[.51 .01 .48 .98],'Title','Zoomed View');
    a2 = axes(h2,'Position',[0.05 0.05 0.9 0.9]);
    zoomedtp = theaterPlot('Parent', a2, 'XLimits',[400 500], 'YLimits',[400 500]);
    set(a2,'XTick',400:100:500)
    set(a2,'YTick',400:100:500)
    grid on
    zoomedpp = trackPlotter(zoomedtp,'DisplayName','Truth','Marker','^','MarkerEdgeColor','k','Marker
    plotTrack(zoomedpp,reshape([platforms.Position],3,[])');
    trackPlotter(zoomedtp, 'DisplayName','Tracks','MarkerEdgeColor','b','MarkerSize',8,'HistoryDepth
end
```

runTracker

This function runs the tracker, updates the plotters, and gathers track assignment metrics.

```matlab
function [trkSummary,truSummary,info] = runTracker(platforms,tracker,positionSelector,tp,zoomedt
% This is a helper function to run the tracker and display the results. It
% may be removed in the future.

pp = findPlotter(tp,'Tag','Truth');
trp = findPlotter(tp, 'Tag','Tracks');
zoomedpp = findPlotter(zoomedtp,'DisplayName','Truth');
zoomedtrp = findPlotter(zoomedtp, 'DisplayName','Tracks');

% To save time, pre-allocate all the detections and assign them on the fly.
nPl = numel(platforms);
det = objectDetection(0,[0;0;0]);
dets = repmat({det},[nPl,1]);

% Define a track assignment metrics object.
tam = trackAssignmentMetrics;

% Bring the visualization back.
set(tp.Parent.Parent.Parent,'Visible','on')

hasExternalCostFunction = tracker.HasCostMatrixInput;

% Measure the time it takes to set the tracker up.
tic
if ~isLocked(tracker)
    if hasExternalCostFunction
        setup(tracker,dets,0,0);
    else
        setup(tracker,dets,0);
    end
end
reset(tracker)
disp("Tracker set up time: " + toc);

% Run 5 steps with detections for all the platforms.
for t = 1:5
    for i = 1:nPl
        dets{i}.Time = t;
        dets{i}.Measurement = platforms(i).Position(:);
    end

    tic
    if hasExternalCostFunction
```

5-331

```matlab
        if isLocked(tracker)
            % Use predictTracksToTime to get all the predicted tracks.
            allTracks = predictTracksToTime(tracker,'all',t);
        else
            allTracks = [];
        end
        costMatrix = predictedEuclidean(allTracks,dets,positionSelector);
        [tracks,~,~,info] = tracker(dets,t,costMatrix);
    else
        [tracks,~,~,info] = tracker(dets,t);
    end
    trPos = getTrackPositions(tracks, positionSelector);
    trIDs = string([tracks.TrackID]');
    disp("Step " + t + " time: " + toc)

    % Update the plot.
    plotTrack(pp,reshape([platforms.Position],3,[])');
    plotTrack(trp,trPos);
    plotTrack(zoomedpp,reshape([platforms.Position],3,[])');
    plotTrack(zoomedtrp,trPos,trIDs);
    drawnow

    % Update the track assignment metrics object.
    if nargout
        [trkSummary, truSummary] = tam(tracks,platforms);
    end

    % Update the platform positions.
    for i = 1:nPl
        platforms(i).Position = platforms(i).Position + platforms(i).Velocity;
    end
end
snapnow

% Run steps with no detections until the tracker deletes all the tracks.
disp("All steps from now are without detections.")
while ~isempty(tracks)
    t = t+1;
    tic
    if hasExternalCostFunction
        allTracks = predictTracksToTime(tracker,'all',t);
        costMatrix = predictedEuclidean(allTracks,{},positionSelector);
        tracks = tracker({},t,costMatrix);
    else
        tracks = tracker({},t);
    end
    disp("Step " + t + " time: " + toc)

    % Update the position of the tracks to plot.
    trPos = getTrackPositions(tracks,positionSelector);
    trIDs = string([tracks.TrackID]');

    % Update the plot.
    plotTrack(pp,reshape([platforms.Position],3,[])');
    plotTrack(trp,trPos);
    plotTrack(zoomedpp,reshape([platforms.Position],3,[])');
    plotTrack(zoomedtrp,trPos,trIDs);
    drawnow
```

```matlab
    % Update the platform positions.
    for i = 1:nPl
        platforms(i).Position = platforms(i).Position + platforms(i).Velocity;
    end
end
disp("Scenario done. All tracks are now deleted." + newline)
clearData(pp)
clearData(trp)
clearData(zoomedpp)
clearData(zoomedtrp)
set(tp.Parent.Parent.Parent,'Visible','off') % Prevent excessive snapshots
drawnow
end
```

predictedEuclidean

The function calculates the Euclidean distance between measured positions from detections and predicted positions from tracks.

```matlab
function euclidDist = predictedEuclidean(tracks,detections,positionSelector)
% This is a helper function to run the tracker and display the results. It
% may be removed in the future.

if isempty(tracks) || isempty(detections)
    euclidDist = zeros(numel(tracks),numel(detections));
    return
end

predictedStates = [tracks.State];
predictedPositions = positionSelector * predictedStates;
dets = [detections{:}];
measuredPositions = [dets.Measurement];
euclidDist = zeros(numel(tracks),numel(detections));
for i = 1:numel(detections)
    diffs = bsxfun(@minus, predictedPositions',measuredPositions(:,i)');
    euclidDist(:,i) = sqrt(sum((diffs .* diffs),2));
end
end
```

assignmentMetricsSummary

The function displays the key assignment metrics in a table form.

```matlab
function assignmentMetricsSummary(trkSummary,truSummary)
trkSummary = rmfield(trkSummary, {'TotalSwapCount','TotalDivergenceCount',...
    'TotalDivergenceLength','MaxRedundancyCount','TotalRedundancyCount',...
    'MaxRedundancyLength','TotalRedundancyLength'});
truSummary = rmfield(truSummary, {'TotalEstablishmentLength','TotalBreakCount',...
    'MaxBreakLength','TotalBreakLength'});
trkTable = struct2table(trkSummary);
truTable = struct2table(truSummary);
disp("Track assignment metrics summary:")
disp(trkTable)
disp("Truth assignment metrics summary:")
disp(truTable)
end
```

Tracking a Flock of Birds

This example shows how to track a large number of objects. A large flock of birds is generated and a global nearest neighbor multi-object tracker, `trackerGNN`, is used to estimate the motion of every bird in the flock.

Scenario Definition

The flock motion is simulated using the behavioral model proposed by Reynolds [1]. In this example, the flock is comprised of 1000 simulated birds, called boids, whose initial position and velocity was previously saved. They follow the three rules of flocking: collision avoidance, velocity matching, and flock centering. Each rule is associated with a weight and the overall behavior of the flock emerges from the relative weighting of each rule. In this case, weights are chosen that cause the flock to fly around a certain point and create a dense center. Other weight settings can cause different behaviors to emerge.

Tracking such a large and dense flock presents two challenges:

1 How to efficiently track 1000 boids?

2 How to be able to track individual boids in such a dense environment?

The following code simulates the flock behavior for 100 steps of 0.1 second. The plot on the left shows the flock as a whole and the plot on the right is zoomed in on the densest part at the flock center.

```
s = rng;    % Keep the current state of the random number generator
rng(2019); % Set the random number generator for repeatable results
load("initialFlock.mat","x","v");
flock = helperFlock("NumBoids",size(x,1),"CollisionAviodanceWeight",0.5,...
    "VelocityMatchingWeight",0.1,"FlockCenteringWeight",0.5,"Velocity",v,...
    "Position",x,"BoidAcceleration",1);
truLabels = string(num2str((1:flock.NumBoids)'));
bound = 20;
flockCenter = mean(x,1);
[tp1,tp2] = helperCreateDisplay(x,bound);

% Simulate 100 steps of flocking
numSteps = 100;
allx = repmat(x,[1 1 numSteps]);
dt = 0.1;
for i = 1:numSteps
    [x,v] = move(flock,dt);
    allx(:,:,i) = x;
    plotTrack(tp1.Plotters(1),x)
    inView = findInView(x,-bound+flockCenter,bound+flockCenter);
    plotTrack(tp2.Plotters(1),x(inView,:),truLabels(inView))
    drawnow
end
```
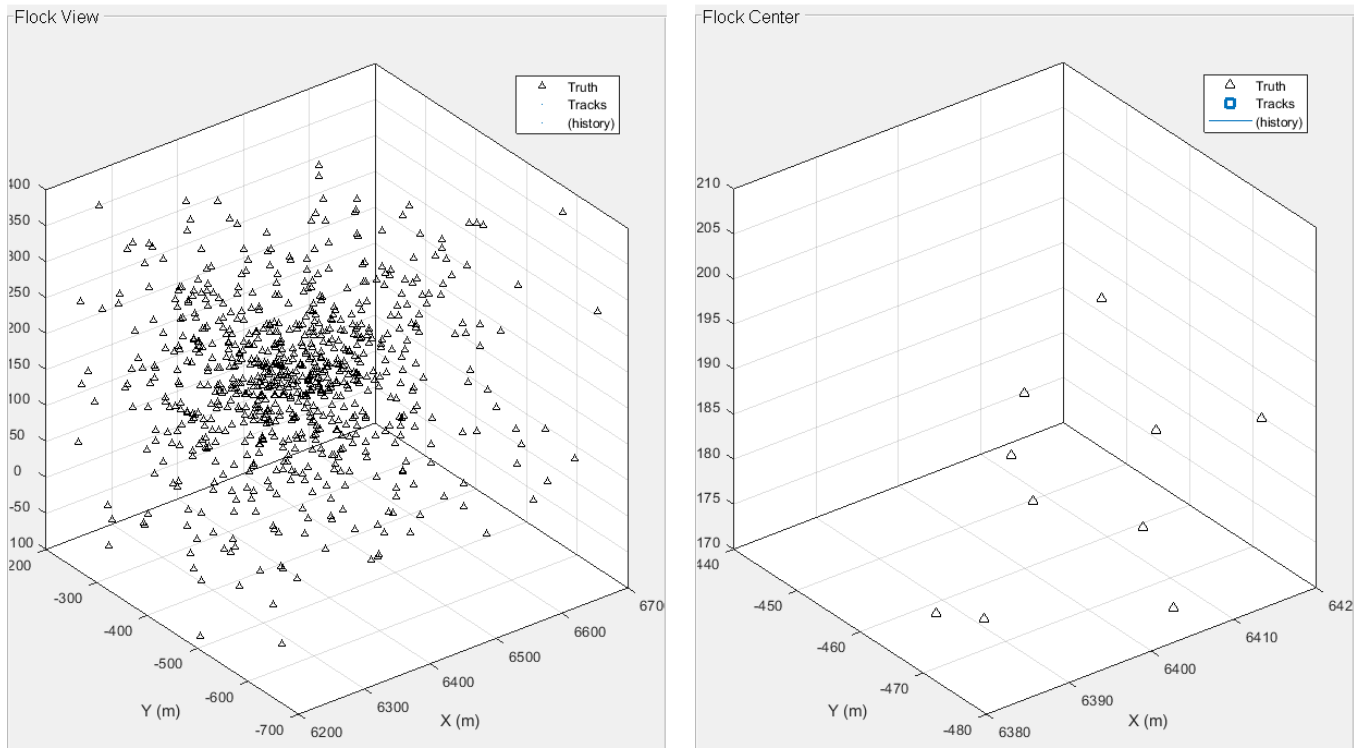
Tracker Definition

You define the tracker as shown in the example "How to Efficiently Track Large Numbers of Objects" on page 5-320.

You observe that the boids follow a curved path and choose a constant turn model defined by `initctekf`.

To limit the time required to calculate cost, you reduce the coarse cost calculation threshold in the `AssignmentThreshold` to a low value.

Further, you choose the more efficient `Jonker-Volgenant` as the assignment algorithm, instead of the default `Munkres` algorithm.

You want tracks to be quickly confirmed and deleted, and set the confirmation and deletion thresholds to [2 3] and [2 2], respectively.

Finally, you know that the sensor scans only a fraction of the flock at any given scan, and so you set the `HasDetectableTrackIDsInput` to `true` to be able to pass the detectable track IDs to the tracker.

The following line shows how the tracker is configured with the above properties. You can see how to generate code for a tracker in "How to Generate C Code for a Tracker" on page 5-313, and the tracker for this example is saved in the function flockTracker_kernel.m

```
% tracker = trackerGNN("FilterInitializationFcn",@initctekf,"MaxNumTracks",1500,...
%          "AssignmentThreshold",[50 800],"Assignment","Jonker-Volgenant",...
%          "ConfirmationThreshold",[2 3],"DeletionThreshold",[2 2],...
%          "HasDetectableTrackIDsInput",true);
```

Track the Flock

Next, you run the scenario and track the flock.

A simplified sensor model is simulated using the `detectFlock` supporting function. It simulates a sensor that scans the flock from left to right, and captures a fifth of the flock span in the x-axis in every scan. The sensor has a 0.98 probability of detection and the noise is simulated using a normal distribution with a standard deviation of 0.1 meters about each position component.

The sensor reports its `currentScan` bounds, which are used to provide the detectable track IDs to the tracker.

```
clear flockTracker_kernel
positionSelector = [1 0 0 0 0 0 0; 0 0 1 0 0 0 0; 0 0 0 0 0 1 0];
trackIDs = zeros(0,1,'uint32');
trax = zeros(0,3);
bounds = inf(3,2);
alltrax = zeros(size(allx));
allIDs = repmat({},1,numSteps);
trup2 = tp2.Plotters(1);
trap2 = tp2.Plotters(2);
trup2.HistoryDepth = 2*trap2.HistoryDepth;
clearPlotterData(tp1)
clearPlotterData(tp2)
for i = 1:numSteps
    t = i*dt;
    [detections, currentScan] = detectFlock(allx(:,:,i),t);
    bounds(1,:) = currentScan;
    tracksInScan = findInView(trax,bounds(:,1),bounds(:,2));
    [tracks,info] = flockTracker_kernel(detections,t,trackIDs(tracksInScan,1));
    trax = getTrackPositions(tracks,positionSelector);
    if ~isempty(tracks)
        trackIDs = uint32([tracks.TrackID]');
    else
        trackIDs = zeros(0,1,'uint32');
    end
    alltrax(1:size(trax,1),1:3,i) = trax;
    allIDs{i} = string(trackIDs);
    helperVisualizeDisplay(tp1,tp2,truLabels,allx,allIDs,alltrax,i)
end
rng(s); % Reset the random number generator to its previous state
```
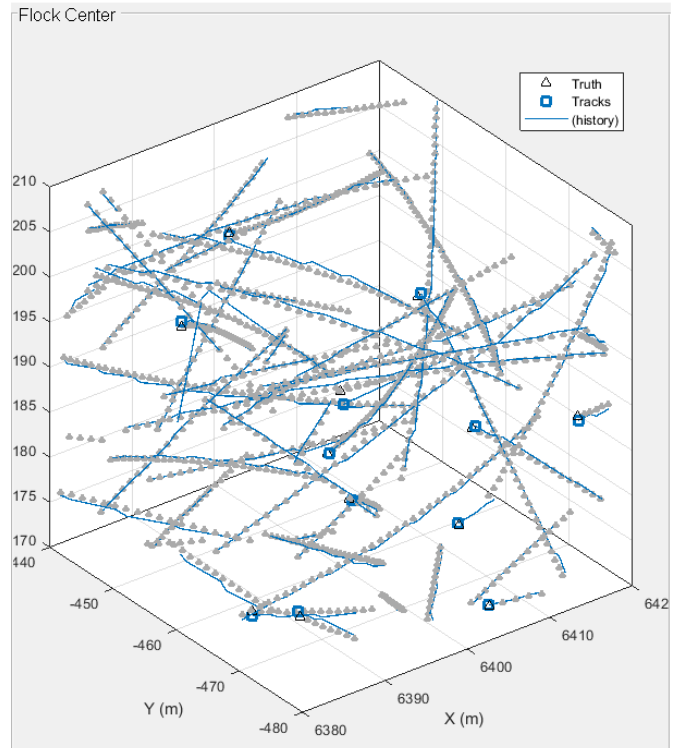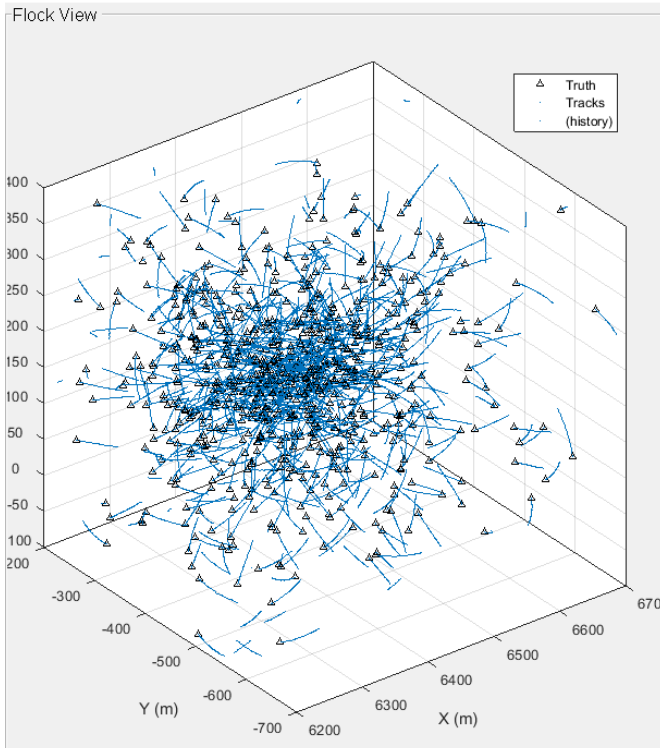
Result of the Tracker in Generated Code

The following GIF shows the performance of the tracker in a mex file.
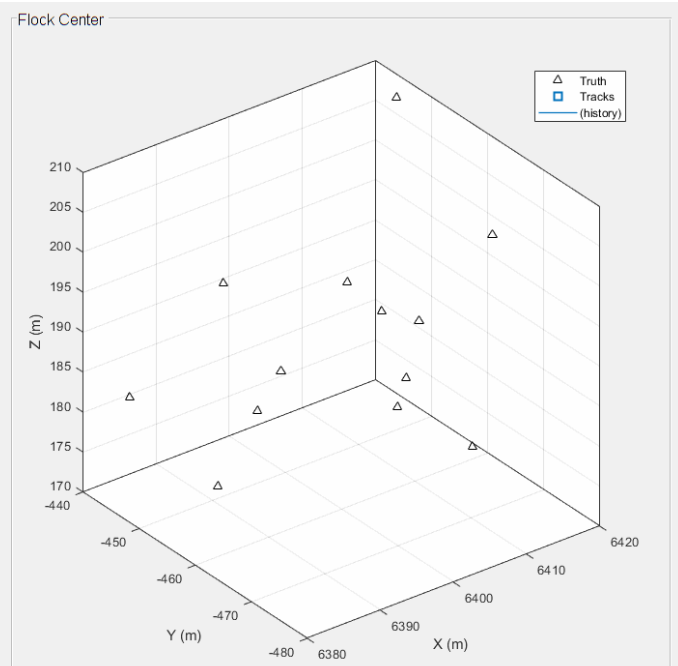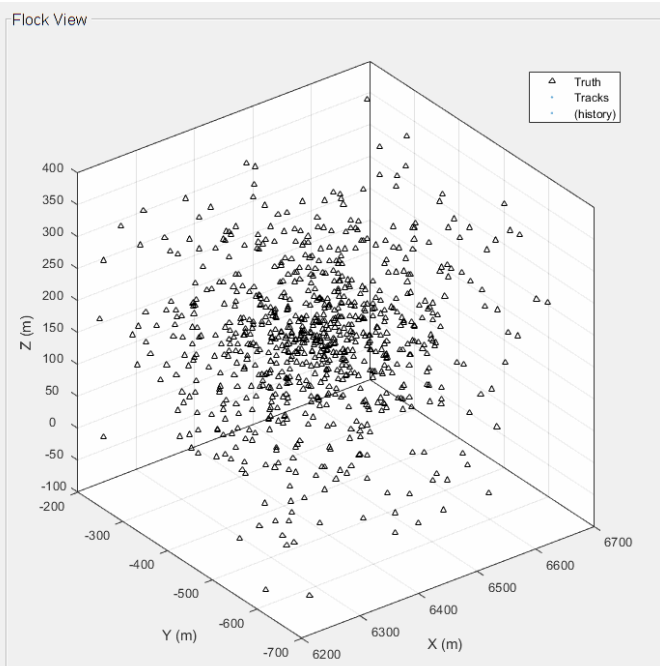
Summary

This example showed how to track a large number of objects in a realistic scenario, where a scanning sensor only reports a fraction of the objects in each scan. The example showed how to set the tracker up for large number of objects and how to use the detectable track IDs input to prevent tracks from being deleted.

References

[1] Craig W. Reynolds, "Flocks, Herds, and Schools: A Behavioral Model", Computer Graphics, Vol. 21, Number 4, July 1987.

Supporting Functions

helperCreateDisplay

The function creates the example display and returns a handle to both theater plots.

```
function [tp1,tp2] = helperCreateDisplay(x,bound)
f  = figure("Visible", "off");
set(f,"Position",[1 1 1425 700]);
movegui(f,"center")
h1 = uipanel(f,"FontSize",12,"Position",[.01 .01 .48 .98],"Title","Flock View");
h2 = uipanel(f,"FontSize",12,"Position",[.51 .01 .48 .98],"Title","Flock Center");
flockCenter = mean(x,1);

a1 = axes(h1,'Position',[0.05 0.05 0.9 0.9]);
grid(a1,'on')
tp1 = theaterPlot("Parent",a1);

% Flock View (Truncated)
halfspan = 250;
tp1.XLimits = 100*round([-halfspan+flockCenter(1) halfspan+flockCenter(1)]/100);
tp1.YLimits = 100*round([-halfspan+flockCenter(2) halfspan+flockCenter(2)]/100);
tp1.ZLimits = 100*round([-halfspan+flockCenter(3) halfspan+flockCenter(3)]/100);
trackPlotter(tp1,"DisplayName","Truth","HistoryDepth",0,"Marker","^","MarkerSize",4,"ConnectHist
set(findall(a1,"Type","line","Tag","tpTrackHistory_Truth"),"Color","k");
view(a1,3)
legend('Location','NorthEast')

% Flock center
a2 = axes(h2,'Position',[0.05 0.05 0.9 0.9]);
grid(a2,'on')
tp2 = theaterPlot("Parent",a2);
tp2.XLimits = 10*round([-bound+flockCenter(1) bound+flockCenter(1)]/10);
tp2.YLimits = 10*round([-bound+flockCenter(2) bound+flockCenter(2)]/10);
tp2.ZLimits = 10*round([-bound+flockCenter(3) bound+flockCenter(3)]/10);
trackPlotter(tp2,"DisplayName","Truth","HistoryDepth",0,...
    "Marker","^","MarkerSize",6,"ConnectHistory","off","FontSize",1);
set(findall(a2,"Type","line","Tag","tpTrackHistory_Truth"),"Color","k");

% Track plotters
TrackColor = [0 0.4470 0.7410]; % Blue
TrackLength = 50;
trackPlotter(tp1,"DisplayName","Tracks","HistoryDepth",TrackLength,"ConnectHistory","off",...
    "Marker",".","MarkerSize",3,"MarkerEdgeColor",TrackColor,"MarkerFaceColor",TrackColor);
set(findall(tp1.Parent,"Type","line","Tag","tpTrackHistory_Tracks"),...
    "Color",TrackColor,"MarkerSize",3,"MarkerEdgeColor",TrackColor);
```

```
trackPlotter(tp2,"DisplayName","Tracks","HistoryDepth",TrackLength,"ConnectHistory","on",...
    "Marker","s","MarkerSize",8,"MarkerEdgeColor",TrackColor,"MarkerFaceColor","none","FontSize"
set (findall(tp2.Parent,"Type","line","Tag","tpTrackPositions_Tracks"),"LineWidth",2);
set(findall(tp2.Parent,"Type","line","Tag","tpTrackHistory_Tracks"),"Color",TrackColor,"LineWidth

view(a2,3)
legend('Location','NorthEast')
set(f,'Visible','on')
end
```

detectFlock

The function simulates the sensor model. It returns an array of detections and the current sensor scan limits.

```
function [detections,scanLimits] = detectFlock(x,t)
persistent sigma allDetections currentScan numScans
numBoids = size(x,1);
pd = 0.98;
if isempty(sigma)
    sigma = 0.1;
    oneDet = objectDetection(0,[0;0;0],"MeasurementNoise",sigma,'ObjectAttributes',struct);
    allDetections = repmat(oneDet,numBoids,1);
    currentScan = 1;
    numScans = 5;
end

% Vectorized calculation of all the detections
x = x + sigma*randn(size(x));
[allDetections.Time] = deal(t);
y = mat2cell(x',3,ones(1,size(x,1)));
[allDetections.Measurement] = deal(y{:});

% Limit the coverage area based on the number of scans
flockXSpan = [min(x(:,1),[],1),max(x(:,1),[],1)];
spanPerScan = (flockXSpan(2)-flockXSpan(1))/numScans;
scanLimits = flockXSpan(1) + spanPerScan * [(currentScan-1) currentScan];
inds = and(x(:,1)>=scanLimits(1), x(:,1)<=scanLimits(2));

% Add Pd
draw = rand(size(inds));
inds = inds & (draw<pd);
dets = allDetections(inds);
detections = num2cell(dets);

% Promote the scan count
currentScan = currentScan+1;
if currentScan > numScans
    currentScan = 1;
end
end
```

findInView

The function returns a logical array for positions that fall within the limits of minBound and maxBound.

```
function inView = findInView(x,minBound,maxBound)
inView = false(size(x,1),1);
```

5-339

```
inView(:) = (x(:,1)>minBound(1) & x(:,1)<maxBound(1)) & ...
        (x(:,2)>minBound(2) & x(:,2)<maxBound(2)) & ...
        (x(:,3)>minBound(3) & x(:,3)<maxBound(3));
end
```

helperVisualizeDisplay

The function displays the flock and tracks after tracking.

```
function helperVisualizeDisplay(tp1,tp2,truLabels,allx,allIDs,alltrax,i)
trup1 = tp1.Plotters(1);
trap1 = tp1.Plotters(2);
trup2 = tp2.Plotters(1);
trap2 = tp2.Plotters(2);
plotTrack(trup1,allx(:,:,i))
n = numel(allIDs{i});
plotTrack(trap1,alltrax(1:n,:,i))

bounds = [tp2.XLimits;tp2.YLimits;tp2.ZLimits];

inView = findInView(allx(:,:,i),bounds(:,1),bounds(:,2));
plotTrack(trup2,allx(inView,:,i),truLabels(inView))
inView = findInView(alltrax(1:n,:,i),bounds(:,1),bounds(:,2));
plotTrack(trap2,alltrax(inView,:,i),allIDs{i}(inView))
drawnow
end
```

Tracking Using Bistatic Range Detections

This example shows you how to simulate bistatic range-only detections using four sensor-emitter pairs. In addition, this example demonstrates how to localize and track multiple targets using bistatic range-only measurements.

Introduction

A bistatic radar is a collection of a bistatic emitter or transmitter (T_X), and a bistatic receiver or sensor (R_X). The geometry of a bistatic system is depicted in the figure below. The sensor receives signals along the path forming the upper sides of the triangle $(R_T + R_R)$ with bistatic detections referenced to the emitter range. The relative bistatic range is given by:

$$R_{Bistatic} = R_T + R_R - L,$$

where R_T is the range from the emitter to the target, R_R is the range from the target to the sensor, and L, known also as the direct-path or baseline, is the range from the emitter to the sensor.
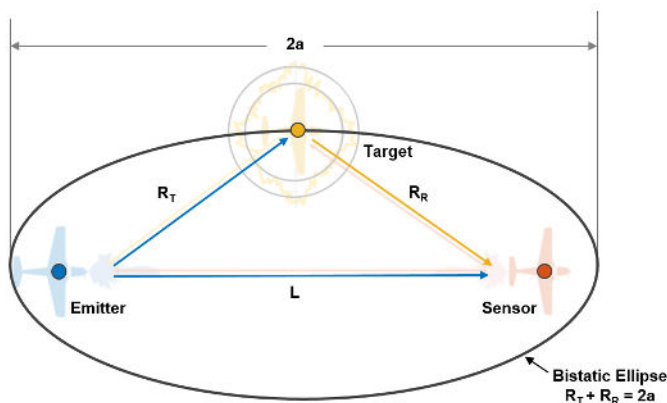
The emitter-to-target-to-sensor range obtained by the bistatic radar is equal to the sum:

$$R_T + R_R = 2a.$$

This sum defines an ellipsoid of constant range. The image shown above describes the ellipsoid when target, transmitter and sensor lie in the same plane. This results in a 2-D ellipse. The target lies somewhere on the surface of the constant-range ellipsoid with the foci being the emitter and sensor locations, which are separated by the baseline distance L and with a major axis equal to 2a.

Next, you will define a scenario to generate the bistatic detections and then use the detections to track the targets.

Define Scenario

You define a tracking scenario which simulates the 3-D area containing multiple platforms. You will define the bistatic system in this scenario.

```
% Set the random seed for repeatable results.
rng(2050);
```

```
% Create a tracking scenario to manage the movement of platforms.
scene = trackingScenario;

% Set the duration of the scenario to 30 seconds and the update rate of the
% scenario to 1 Hz.
scene.StopTime = 30;     % sec
scene.UpdateRate = 1;    % Hz
```

Set the number of each type of platform. In this example, the following types of platform will be created for sensing the targets:

- 1 radar emitter

- 4 bistatic radar sensors

```
% Set the minimum number of sensor-emitter pairs required by the spherical
% intersection algorithm for a better localization of the targets in
% 3-dimensional space.
numSensors = 4;
emitterIdx = 1; % Emitter is added as first platform in scene
```

Create and mount emitter

In a bistatic system, there are three types of emitters:

- Dedicated: This type of transmitter is intentionally designed and operated to support bistatic processing.

- Cooperative: This type of transmitter is designed to support other functions but is suitable for bistatic use. Information about the transmitter such as its transmitted waveform and position are known.

- Non-Cooperative: This type is a "transmitter of opportunity". Non-cooperative transmitters cannot be controlled but are deemed suitable for bistatic use.

In this example, the modeled emitter is considered to be a dedicated transmitter. Model an RF emission using `radarEmitter`. This emitter is an ideal, isotropic emitter with a 360 degrees field of view. The waveform type is a user-defined value used to enumerate the various waveform types that are present in the scenario. For this scenario, use the value 1 to indicate an LFM type waveform. Mount the emitter to a stationary platform at the origin.

```
% Define an emitter.
emitter = radarEmitter(emitterIdx,'No scanning', ...
    'FieldOfView',[360 180], ...    % [az el] deg
    'CenterFrequency', 300e6, ...   % Hz
    'Bandwidth', 30e6, ...          % Hz
    'WaveformType', 1);             % Use 1 for an LFM-like waveform

% Mount emitter on a platform.
thisPlat = platform(scene,'Trajectory',kinematicTrajectory('Position',[0 0 0],...
    'Velocity',[0 0 0]));
thisPlat.Emitters = {emitter};
```

Create and mount bistatic radar sensors

Model a bistatic radar sensor using `radarSensor`. The radar sensor is an ideal, isotropic receiver with a 360 degrees field of view. Set the `DetectionMode` to bistatic. Ensure that the sensor is configured so that its center frequency, bandwidth, and expected waveform types will match the

emitter configuration. Enable INS for tracking in scenario coordinates and mount the bistatic radar sensor to platforms.

```
% Define some random trajectories for the four bistatic radar sensors.
% Circularly distributed with some variance.
r = 2000;                                      % Range, m
theta = linspace(0,pi,numSensors);            % Theta, rad
xSen = r*cos(theta) + 100*randn(1,numSensors);   % m
ySen = r*sin(theta) + 100*randn(1,numSensors);   % m

% To observe the z of the targets, the sensors must have some elevation
% with respect to each other and emitter.
zSen = -1000*rand(1,numSensors);      % m

% Define a bistatic radar sensor.
sensor = radarSensor(1,'No scanning', ...
    'FieldOfView',[360 180], ...     % [az el] deg
    'DetectionMode','Bistatic', ... % Bistatic detection mode
    'CenterFrequency', 300e6, ...    % Hz
    'Bandwidth', 30e6, ...           % Hz
    'WaveformTypes', 1,...           % Use 1 for an LFM-like waveform
    'HasINS',true,...                % Has INS to enable tracking in scenario
    'AzimuthResolution',360,...      % Does not measure azimuth and has a single resolution cell
    'HasElevation',true,...          % Enable elevation to set elevation resolution
    'ElevationResolution',180);      % Single elevation resolution cell

% Mount bistatic radar sensors on platforms.
for iD = 1:numSensors
    % Create a platform with the trajectory. The sensing platforms are
    % considered stationary, but can be provided with a velocity.
    thisPlat = platform(scene,...
        'Trajectory',kinematicTrajectory('Position',[xSen(iD) ySen(iD) zSen(iD)],...
        'Velocity',[0 0 0]));
    % Clone the bistatic radar sensor and mount to platforms.
    thisPlat.Sensors = {clone(sensor)};
    % Provide the correct sensor index.
    thisPlat.Sensors{1}.SensorIndex = iD;
end
```

Simulating the bistatic scenario involves the following:

- Generating RF emissions
- Propagating the emissions and reflecting these emissions from platforms
- Receiving the emissions, calculating interference losses, and generating detections.

This process is wrapped in a supporting function, `detectBistaticTargetRange`, defined at the end of this example.

Target Localization

The figure in the Introduction section illustrated that with a bistatic measurement, the target lies somewhere on an ellipsoid defined by the emitter and sensor positions, as well as the measured bistatic range. Since the target can lie anywhere on the ellipsoid, a single bistatic measurement does not provide full observability of the target state. To localize the target (triangulate the target position) and achieve observability of target's state, multiple measurements from different sensors are needed. The target localization algorithm that is implemented in this example is based on the spherical

intersection method described in reference [1]. The non-linear nature of the localization problem results in two possible target locations from intersection of 3 or more sensor bistatic ranges. A decision about target location from two possible locations can be facilitated by using more than 3 sensors. In this example, you use four sensors to generate bistatic detections using one emitter.

Next, you will add a target to the scene to generate bistatic detections.

Add one target to scenario

Platforms without any attached emitters or sensors are referred to as *targets*. Create targets for tracking using `platform`.

```
% Add one target here using the platform method of scenario. Specify the
% trajectory using a kinematicTrajectory with random position and constant
% velocity.
platform(scene,'Trajectory',kinematicTrajectory(...
                            'Position', [2000*randn 100*randn -1000],...
                            'Velocity',[10*randn 10*randn 5*randn]));

% Initialize the display for visualization.
theaterDisplay = helperBistaticRangeFusionDisplay(scene,...
    'XLim',[-3 3],'YLim', [-3 3],'ZLim',[-2.5 0],...
    'GIF',''); % records new GIF if name specified
view(3);
```

In a scenario with a single target and no false alarms, multiple measurements can be triangulated to obtain the localized target position. This localized position can be used as an estimate of target position or can also be passed to a tracker to estimate the target state. Now, you will generate bistatic radar detections from a single target and visualize the geometry of bistatic ellipsoids. You will calculate the triangulated position using the supporting function `helperBistaticRangeFusion`, included with this example. The `helperBistaticRangeFusion` function calculates the triangulated position of the target, *given* the bistatic range detections generated by the target.

```
% Create a fused detection to represent the triangulated position and
% visualize the position as a 3-D fused position detection.
measParam = struct('Frame','Rectangular',...
    'HasAzimuth',true,'HasElevation',true,...
    'HasRange',true,'HasVelocity',false,...
    'OriginPosition',[0;0;0],...% Fused position is in scenario frame
    'OriginVelocity',[0;0;0],...% Scenario frame has zero velocity
    'Orientation',eye(3),...    % Scenario frame has no rotation.
    'IsParentToChild',false);   % Specify if rotation is specified in parent frame

% Represent the fused detection using objectDetection. It has a 3-D
% position and covariance.
fusedDetection = {objectDetection(0,zeros(3,1),'MeasurementNoise',eye(3),...
    'MeasurementParameters',measParam)};

% Change view
view(-125,9);

% Run scenario.
while advance(scene)
    % Get current simulation time.
    time = scene.SimulationTime;

    % Get bistatic range detections from 1 target.
```

```
        detections = detectBistaticTargetRange(scene,time,emitterIdx,true);

        % Triangulate detections to estimate target position.
        [position, covariance] = helperBistaticRangeFusion(detections);

        % Update the fused detection.
        fusedDetection{1}.Measurement = position;
        fusedDetection{1}.MeasurementNoise = covariance;

        % Update the display.
        theaterDisplay([detections;fusedDetection]);
    end

    % Write new GIF if requested
    writeGIF(theaterDisplay);
```
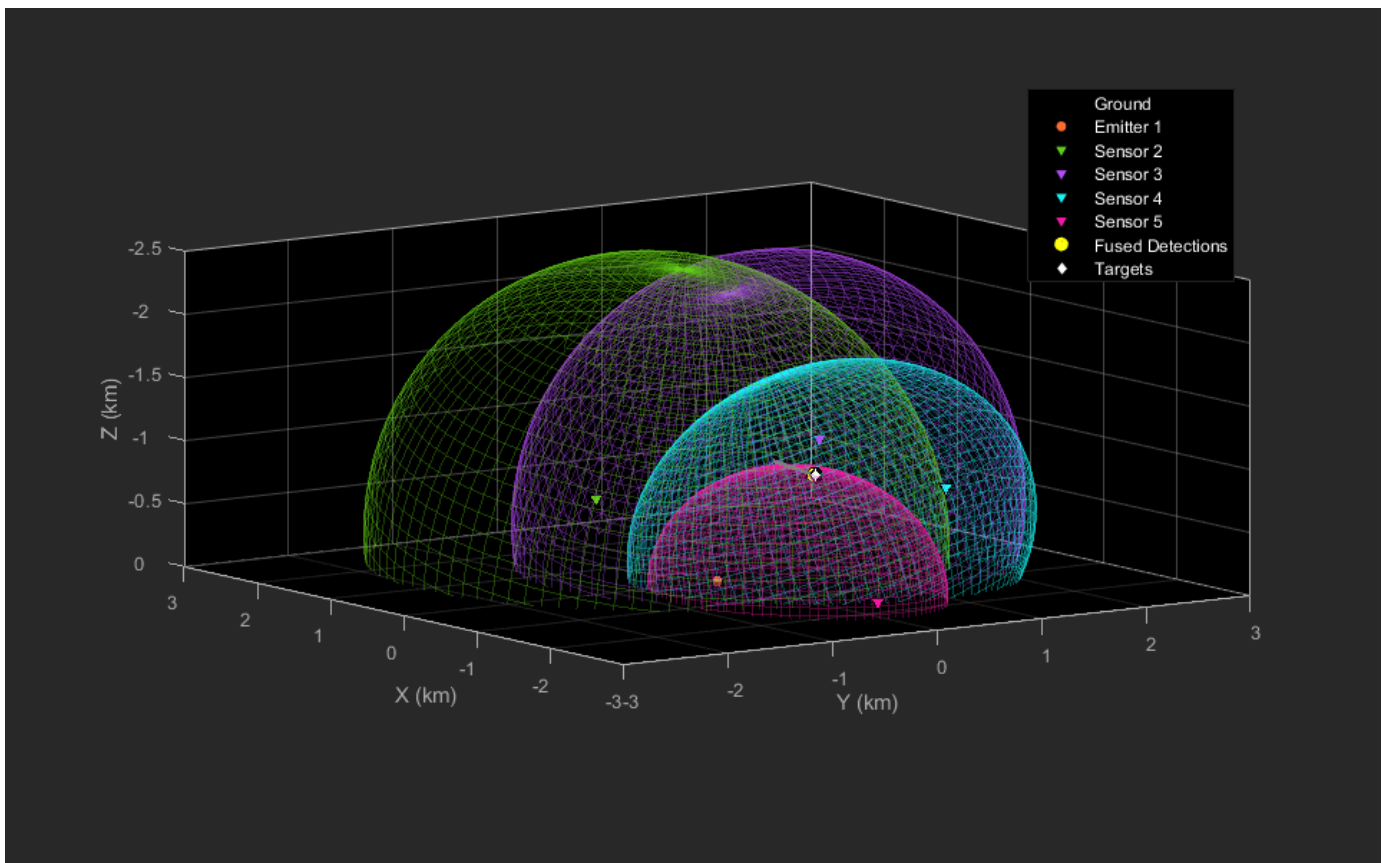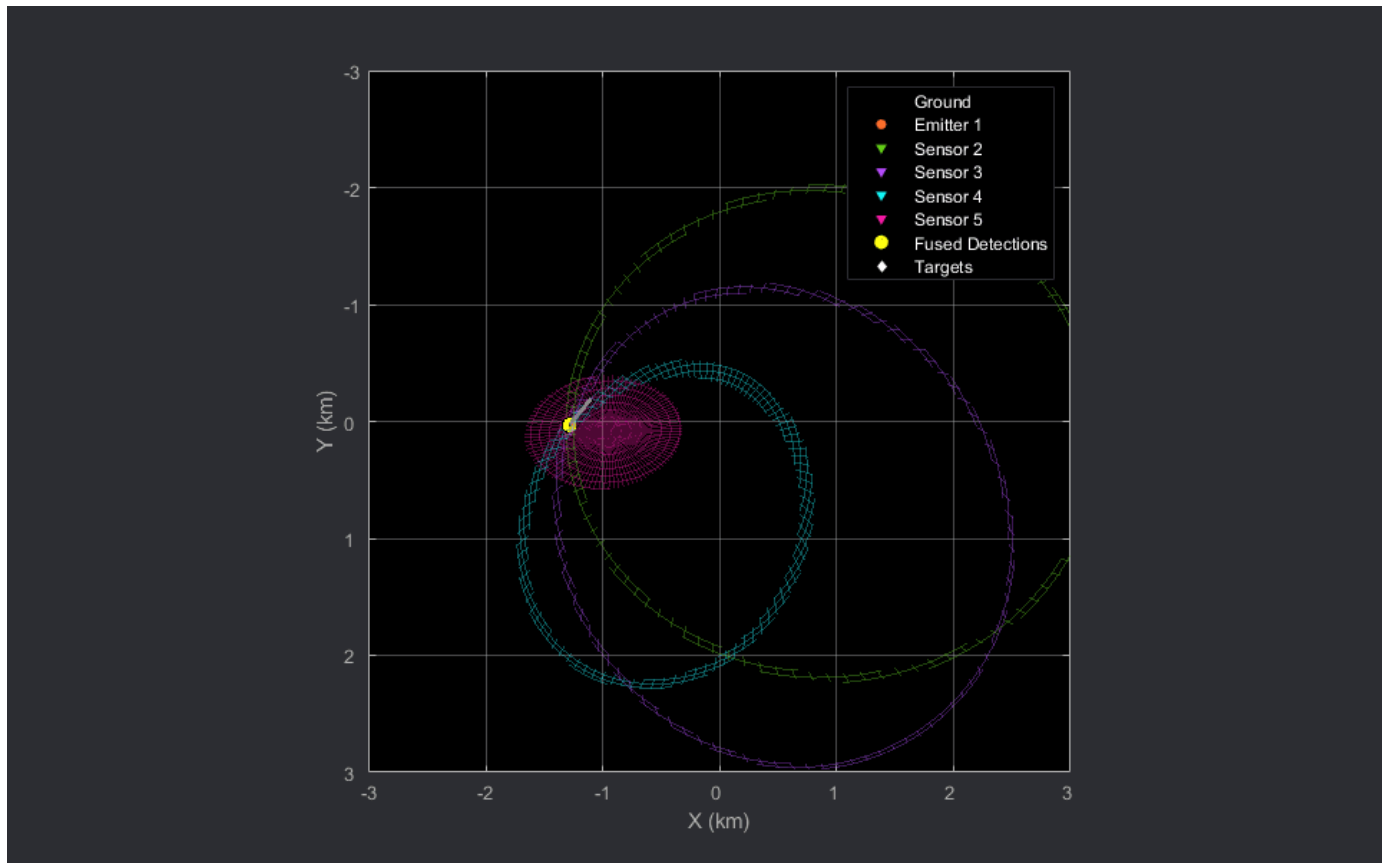
In the preceding animations, the bistatic radar sensors are depicted with the downward-pointing triangles. The stationary emitter is depicted with the purple circle marker at the origin. The target is denoted by the white diamond and the grey line shows the trajectory of the target. The 2-D animation is sliced at the target's Z location at each time stamp.

The fused detection, shown using the yellow circle marker, lies close to the intersection region generated the four ellipsoids and is close to the true position of the target during the scenario.

Multi-Target Scenario

The single-target scenario assumes that detections are known to be generated by the same target. Therefore, you can triangulate them to localize the target. However, in a multi-target scenario and in the presence of false alarms and missed detections, this information is usually not available. This results in the unknown data association between detections and the targets. To solve this problem, you use the `staticDetectionFuser`. The `staticDetectionFuser` estimates the unknown data association between detections and targets and finds the best solution using a multi-dimensional assignment formulation. The `staticDetectionFuser` outputs fused detections. The number of fused detections represent possible number of targets and each detection represents the Cartesian position of the target.

Next, you will add new targets to the scenario, and use the `staticDetectionFuser` to create fused detections from multiple targets in the presence of false alarms. These detections are further processed by a GNN tracker, `trackerGNN`, to track the targets.

```
% Restart the scenario and add remaining targets.
restart(scene);
```

```matlab
% Number of targets added here.
numTargets = 4;

% randomly distributed targets.
r = abs(2000*randn(1,numTargets));              % Random ranges (m)
theta = linspace(0,numTargets*pi/4,numTargets);  % Angular position (rad)
xTgt = r.*cos(theta) + 100*randn(1,numTargets);  % x position (m)
yTgt = -r.*sin(theta) + 100*randn(1,numTargets); % y position (m)

% Targets above ground.
zTgt = -1000*ones(1,numTargets); % z position (m)

for iD = 1:numTargets
    thisPlat = platform(scene,...
        'Trajectory',kinematicTrajectory('Position',[xTgt(iD) yTgt(iD) zTgt(iD)],...
        'Velocity',[10*randn 10*randn 5*randn]));
end

% Update platforms variable.
platforms = scene.Platforms;

% Reset the display.
release(theaterDisplay);

% Turn off plotting for bistatic ellipse for all targets.
theaterDisplay.PlotBistaticEllipse = false;

% No recording
theaterDisplay.GIF = '';

% Call once to plot new trajectories.
theaterDisplay();

% Scenario display with trajectories.
showScenario(theaterDisplay);
snapnow;
showGrabs(theaterDisplay,[]);
```
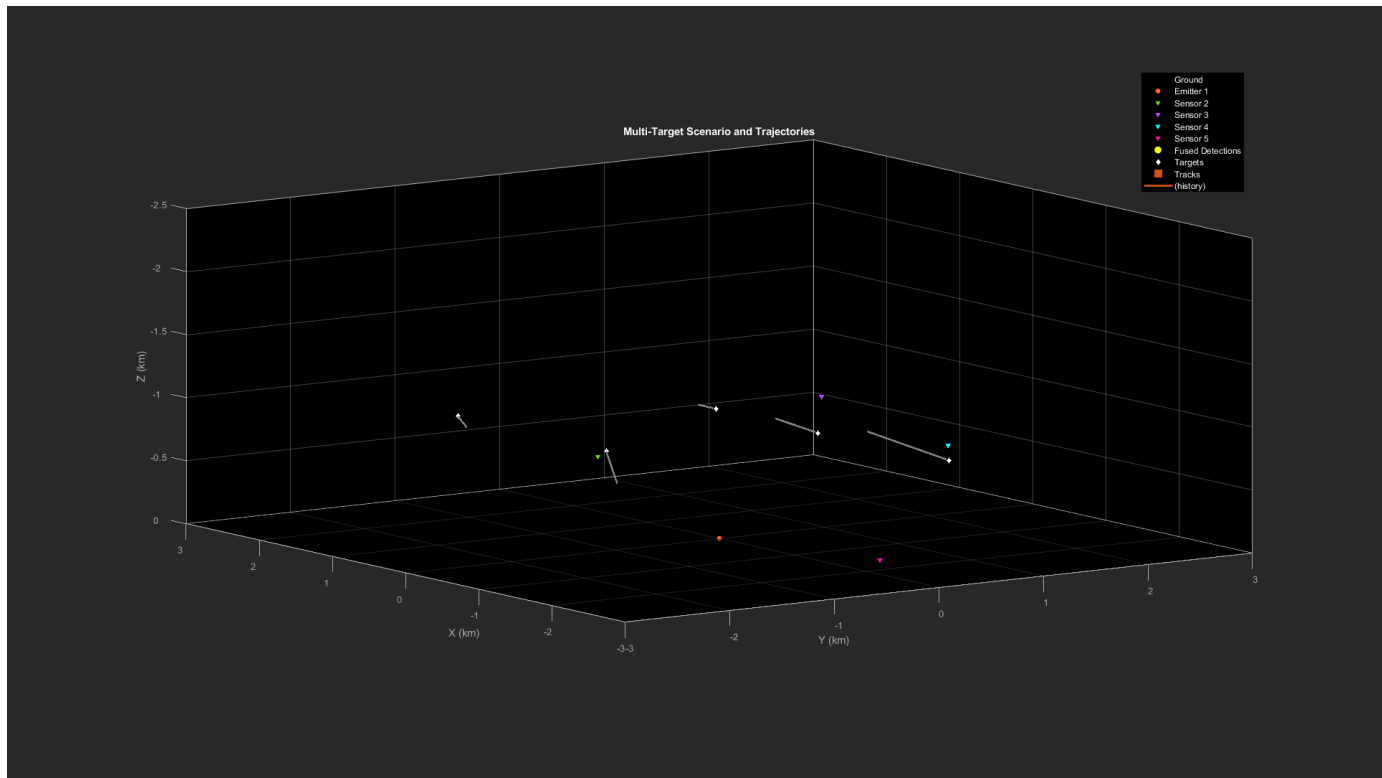
Setup Fuser and Tracker

This section creates a static detection fuser that uses the spherical intersection localization algorithm discussed earlier. Additionally, a Global Nearest Neighbor (GNN) tracker is defined to process the fused detections.

```
% Define a static detection fuser.
fuser = staticDetectionFuser( ...
    'MeasurementFusionFcn','helperBistaticRangeFusion', ...
    'UseParallel',true, ...                              % Do parallel processing
    'MaxNumSensors',numSensors, ...                      % Number of bistatic radar sensors
    'Volume',sensor.RangeResolution, ...                 % Volumes of the sensors' detection bin
    'MeasurementFormat','custom', ...                    % Define custom fused measurement as bistati
    'MeasurementFcn','helperBistaticMeas',...            % Set measurement function for reporting bis
    'DetectionProbability',0.99 ...                      % Probability of detecting the target
    );

% Define a GNN tracker.
tracker = trackerGNN('AssignmentThreshold',100);

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

Track Targets Using Static Fusion

The simulated bistatic detections are fused with the `staticDetectionFuser` using spherical intersection algorithm. The fused detections are then passed to the GNN tracker.

```
while advance(scene)
    % Get current simulation time.
```

```
time = scene.SimulationTime;

% Get bistatic range detections
detections = detectBistaticTargetRange(scene,time,emitterIdx);

% Fuse bistatic detections into one structure.
[superDets, info] = fuser(detections);

% Track fused bistatic detections using the GNN tracker.
confTracks = tracker(superDets,scene.SimulationTime);

% Update display with current platform positions and tracks.
theaterDisplay([superDets(:);detections(:)],confTracks);
end
```

Results

In the presence of multiple targets and possible false alarms, the ghost intersections may sometimes be more favorable than actual solution. As these ghost intersections appear randomly on the scenario, they are effectively treated as "false alarms" by the centralized tracker. You can notice in the figures below that the static fusion outputs detections at the incorrect positions. As ghost intersections compete with true intersections, two true targets are missed at this time.

In the "Current Estimated Tracks" plot, note that the tracker is able to maintain tracks on all 5 targets without creating any ghosts or false tracks.

```
showGrabs(theaterDisplay,1);
```
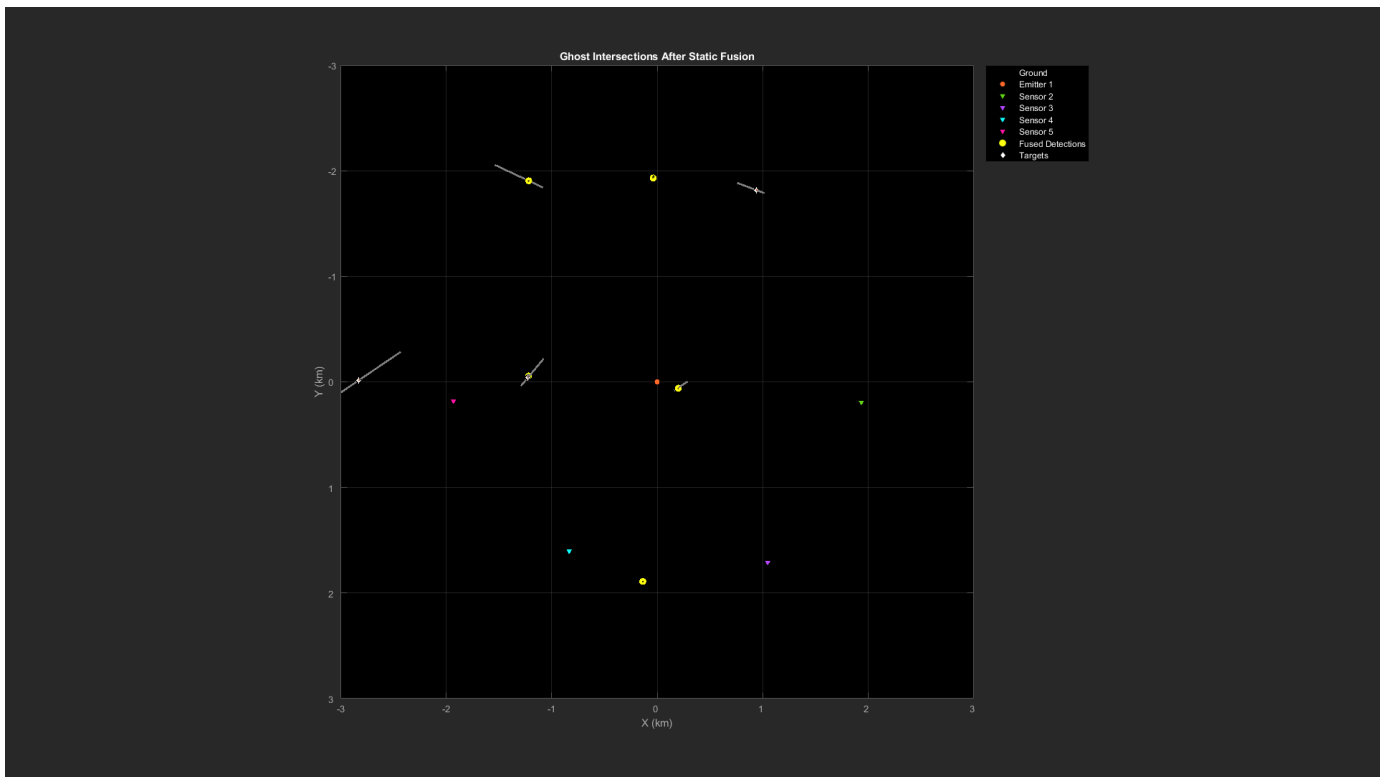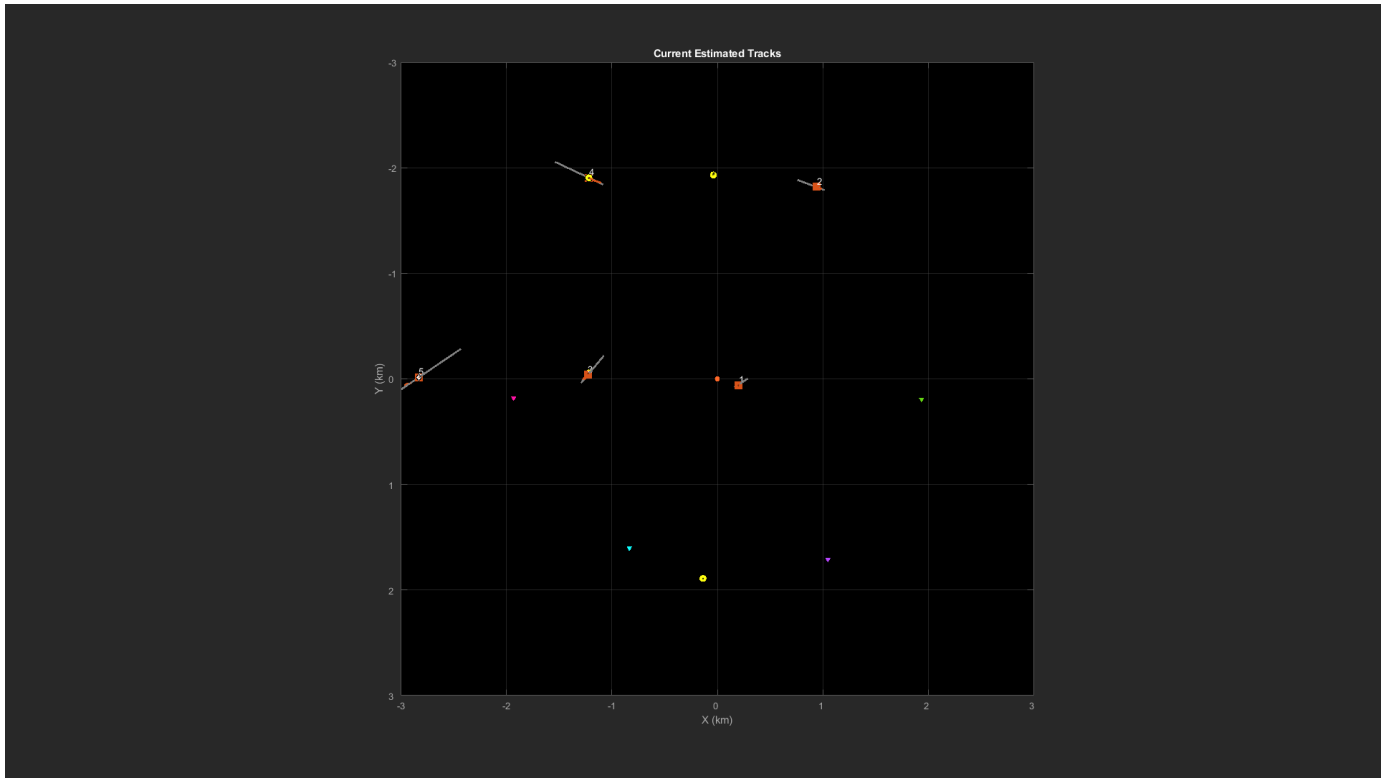
The plots below show the top view of tracks 1 and 2 with their histories at the end of the simulation. The history is represented by the orange line connecting the track. Notice that the track histories are close to the true trajectories of the targets.

```
showGrabs(theaterDisplay,2,false);
```
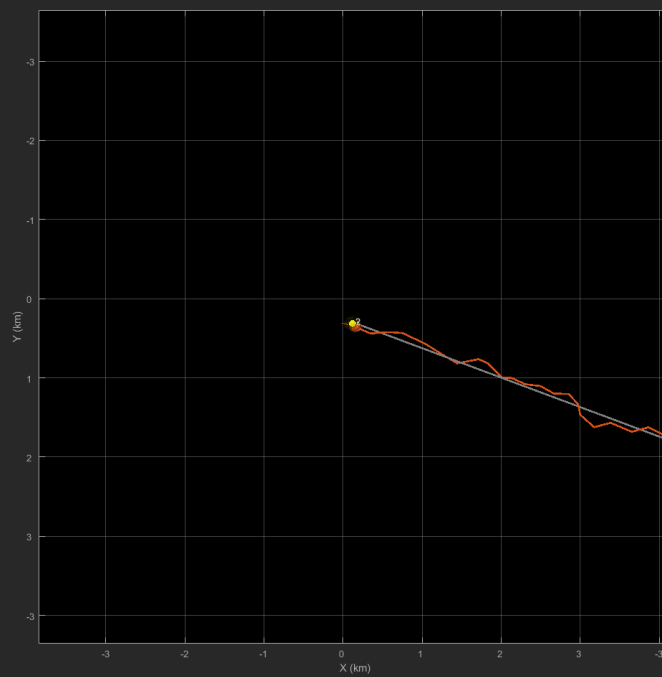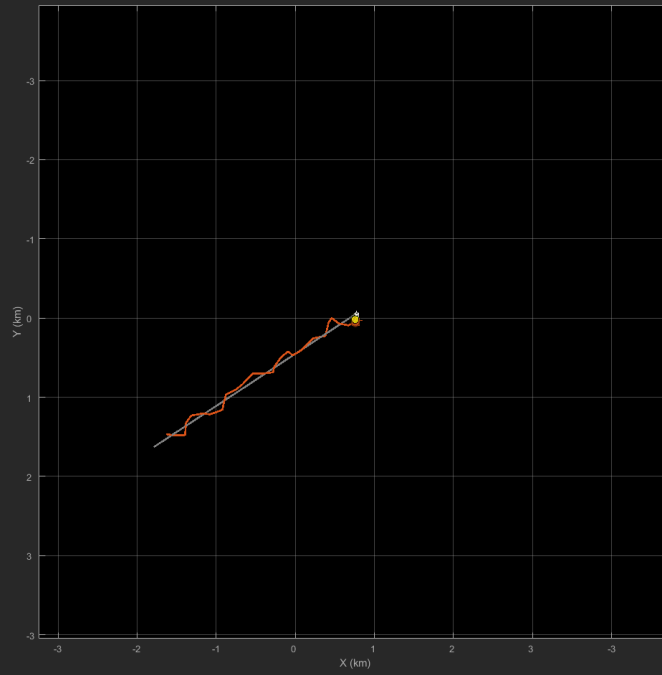
Summary

In this example, you learned how to simulate a scenario with bistatic sensors. You learned about the challenges associated with tracking targets using bistatic measurements. You used the `staticDetectionFuser` to fuse bistatic range detections from multiple targets and `trackerGNN` to track targets with the fused position measurements.

Supporting functions

helperBistaticRangeFusion Fuse range-only detections to triangulate target position

```
function [pos,cov] = helperBistaticRangeFusion(detections)
% This function is for example purposes only and may be removed in a future
% release.
% This function returns the estimated position and covariance of the target
% given the bistatic detections generated from it.

%   Copyright 2019 The MathWorks, Inc.

% Do a coarse gating, as a minimum of 3 measurements are required for
% finding a solution.
if numel(detections) < 3
    pos = 1e10*ones(3,1);
    cov = 2*eye(3);
else
    % Retrieve info from measurements
    ranges = zeros(numel(detections),1);
    receiverLocations = zeros(3,numel(detections));
    emitterLocation = detections{1}.MeasurementParameters.EmitterPosition;
    for i = 1:numel(detections)
        rLoc =  detections{i}.MeasurementParameters(2).OriginPosition;
        receiverLocations(:,i) = rLoc;

        % The spherical intersection method assumes that measurment is
        % Remit + Rrecv. Bistatic measurement is defined as Remit + Rrecv - Rb.
        % Add the Rb to the actual measurement
        L = norm(emitterLocation(:) - rLoc(:));
        ranges(i) = detections{i}.Measurement + L;
    end
    pos = helperSphericalIntersection(ranges,receiverLocations,emitterLocation);

    % Covariance is calculated only when required. This helps saving
    % computation during cost calculation for static fusion, where only
    % position is required.
    if nargout > 1
        cov = linearFusionFcn(pos,detections);
    end
end

end

%% linear fusion function for measurement noise
function measCov = linearFusionFcn(pos,thisDetections)
% Linear noise fusion function. It requires measJacobian to use linear
% transformation.
% Use a constant velocity state to calculate jacobians.
estState = zeros(6,1);
```

```
estState(1:2:end) = pos;
n = numel(thisDetections);
totalJacobian = zeros(n,3);
totalCovariance = zeros(n,n);
for i = 1:numel(thisDetections)
    H = cvmeasjac(estState,thisDetections{i}.MeasurementParameters);
    totalJacobian(i,:) = H(1,1:2:end);
    totalCovariance(i,i) = thisDetections{i}.MeasurementNoise;
end
toInvertJacobian = totalJacobian'/(totalCovariance)*totalJacobian;
I = eye(3);
% 2-D to 3-D conversion with 0 jacobian wrt z.
if toInvertJacobian(3,3) == 0
    toInvertJacobian(3,3) = 1;
end
measCov = I/toInvertJacobian;
% Return true positive definite.
measCov = (measCov + measCov')/2;
measCov(~isfinite(measCov)) = 1000; % Some big number for inf and nan
end
```

detectionBistaticTargetRange Generate bistatic range-only detections from targets in scenario.

```
function detections = detectBistaticTargetRange(scene,time,emitterIdx,removeFalseAlarms)
    % Get platforms from scenario.
    platforms = scene.Platforms;

    % A flag to indicate if false alarms should be removed from detections.
    if nargin == 3
        removeFalseAlarms = false;
    end

    % Distinguish between receivers and targets to remove detections from
    % the receiver. It is assumed that these detections can be removed from
    % the batch using prior information.
    isReceiver = cellfun(@(x)~isempty(x.Sensors),scene.Platforms);
    allIDs = cellfun(@(x)x.PlatformID,scene.Platforms);
    receiverIDs = allIDs(isReceiver);

    % Generate RF emissions
    emitPlatform = platforms{emitterIdx};
    txEmiss = emit(emitPlatform, time);

    % Propagate the emissions and reflect these emissions from platforms.
    reflSigs = radarChannel(txEmiss, platforms,'HasOcclusion',false);

    % Generate detections from the bistatic radar sensor.
    detections = {};
    numPlat = numel(platforms);
    for iPlat = 1:numPlat
        thisPlatform = platforms{iPlat};

        % Receive the emissions, calculate interference losses, and
        % generate bistatic detections.
        thisDet = detect(thisPlatform, reflSigs, time);

        % Remove the detections that are the bistatic receivers. Only the
```

```
        % detections from the target platforms will be fused and tracked.
        detectedTargetIDs = cellfun(@(x)x.ObjectAttributes{1}.TargetIndex,thisDet);
        toRemove = ismember(detectedTargetIDs, receiverIDs) | removeFalseAlarms*(detectedTargetID
        thisDet = thisDet(~toRemove);

        % Add this platform's detections to the detections array.
        detections = [detections; thisDet]; %#ok<AGROW>
    end

    % Determine emitter position and velocity for this simulation time.
    emitterPosition = emitPlatform.Trajectory.Position(:);

    % Update detections structure to indicate that only bistatic range measurements are retained
    for iD = 1:numel(detections)
        detections{iD}.Measurement = detections{iD}.Measurement(end);  %#ok<AGROW> % Range measu
        detections{iD}.MeasurementNoise = detections{iD}.MeasurementNoise(end,end); %#ok<AGROW> %
        detections{iD}.MeasurementParameters(1).HasAzimuth = false; %#ok<AGROW> % Update measurem
        detections{iD}.MeasurementParameters(1).HasElevation = false; %#ok<AGROW> % Update measu
        detections{iD}.MeasurementParameters(1).EmitterPosition = emitterPosition; %#ok<AGROW> %
    end
end
```

References

1 Malanowski, M. and K. Kulpa. "Two Methods for Target Localization in Multistatic Passive Radar." IEEE Transactions on Aerospace and Electronic Systems, Vol. 48, No. 1, Jan. 2012, pp. 572-578.

2 Willis, N. J. Bistatic Radar. Raleigh:SciTech Publishing, Inc., 2005.

Pose Estimation From Asynchronous Sensors

This example shows how you might fuse sensors at different rates to estimate pose. Accelerometer, gyroscope, magnetometer and GPS are used to determine orientation and position of a vehicle moving along a circular path. You can use controls on the figure window to vary sensor rates and experiment with sensor dropout while seeing the effect on the estimated pose.

Simulation Setup

Load prerecorded sensor data. The sensor data is based on a circular trajectory created using the `waypointTrajectory` class. The sensor values were created using the `gpsSensor` and `imuSensor` classes. The `CircularTrajectorySensorData.mat` file used here can be generated with the `generateCircularTrajSensorData` function.

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

Fs = ld.Fs; % maximum MARG rate
gpsFs = ld.gpsFs; % maximum GPS rate
ratio = Fs./gpsFs;
refloc = ld.refloc;

trajOrient = ld.trajData.Orientation;
trajVel = ld.trajData.Velocity;
trajPos = ld.trajData.Position;
trajAcc = ld.trajData.Acceleration;
trajAngVel = ld.trajData.AngularVelocity;

accel = ld.accel;
gyro = ld.gyro;
mag = ld.mag;
lla = ld.lla;
gpsvel = ld.gpsvel;
```

Fusion Filter

Create an `insfilterAsync` to fuse IMU + GPS measurements. This fusion filter uses a continuous-discrete extended Kalman filter (EKF) to track orientation (as a quaternion), angular velocity, position, velocity, acceleration, sensor biases, and the geomagnetic vector.

This `insfilterAsync` has several methods to process sensor data: `fuseaccel`, `fusegyro`, `fusemag` and `fusegps`. Because `insfilterAsync` uses a continuous-discrete EKF, the `predict` method can step the filter forward an arbitrary amount of time.

```
fusionfilt = insfilterAsync('ReferenceLocation', refloc);
```

Initialize the State Vector of the `insfilterAsync`

The `insfilterAsync` tracks the pose states in a 28-element vector. The states are:

```
    States                          Units    Index
  Orientation (quaternion parts)             1:4
  Angular Velocity (XYZ)          rad/s      5:7
  Position (NED)                  m          8:10
  Velocity (NED)                  m/s        11:13
  Acceleration (NED)              m/s^2      14:16
  Accelerometer Bias (XYZ)        m/s^2      17:19
  Gyroscope Bias (XYZ)            rad/s      20:22
```

```
Geomagnetic Field Vector (NED)     uT        23:25
Magnetometer Bias (XYZ)            uT        26:28
```

Ground truth is used to help initialize the filter states, so the filter converges to good answers quickly.

```
Nav = 100;
initstate = zeros(28,1);
initstate(1:4) = compact( meanrot(trajOrient(1:Nav)));
initstate(5:7) = mean( trajAngVel(10:Nav,:), 1);
initstate(8:10) = mean( trajPos(1:Nav,:), 1);
initstate(11:13) = mean( trajVel(1:Nav,:), 1);
initstate(14:16) = mean( trajAcc(1:Nav,:), 1);
initstate(23:25) = ld.magField;

% The gyroscope bias initial value estimate is low for the Z-axis. This is
% done to illustrate the effects of fusing the magnetometer in the
% simulation.
initstate(20:22) = deg2rad([3.125 3.125 3.125]);
fusionfilt.State = initstate;
```

Set the Process Noise Values of the `insfilterAsync`

The process noise variance describes the uncertainty of the motion model the filter uses.

```
fusionfilt.QuaternionNoise = 1e-2;
fusionfilt.AngularVelocityNoise = 100;
fusionfilt.AccelerationNoise = 100;
fusionfilt.MagnetometerBiasNoise = 1e-7;
fusionfilt.AccelerometerBiasNoise = 1e-7;
fusionfilt.GyroscopeBiasNoise = 1e-7;
```

Define the Measurement Noise Values Used to Fuse Sensor Data

Each sensor has some noise in the measurements. These values can typically be found on a sensor's datasheet.

```
Rmag = 0.4;
Rvel = 0.01;
Racc = 610;
Rgyro = 0.76e-5;
Rpos = 3.4;

fusionfilt.StateCovariance = diag(1e-3*ones(28,1));
```

Initialize Scopes

The `HelperScrollingPlotter` scope enables plotting of variables over time. It is used here to track errors in pose. The `PoseViewerWithSwitches` scope allows 3D visualization of the filter estimate and ground truth pose. The scopes can slow the simulation. To disable a scope, set the corresponding logical variable to false.

```
useErrScope = true; % Turn on the streaming error plot.
usePoseView = true; % Turn on the 3D pose viewer.
if usePoseView
    posescope = PoseViewerWithSwitches(...
        'XPositionLimits', [-30 30], ...
        'YPositionLimits', [-30, 30], ...
        'ZPositionLimits', [-10 10]);
```

```
    end
f = gcf;

if useErrScope
    errscope = HelperScrollingPlotter(...
        'NumInputs', 4, ...
        'TimeSpan', 10, ...
        'SampleRate', Fs, ...
        'YLabel', {'degrees', ...
        'meters', ...
        'meters', ...
        'meters'}, ...
        'Title', {'Quaternion Distance', ...
        'Position X Error', ...
        'Position Y Error', ...
        'Position Z Error'}, ...
        'YLimits', ...
        [ -1, 30
        -2, 2
        -2 2
        -2 2]);
end
```

Simulation Loop

The simulation of the fusion algorithm allows you to inspect the effects of varying sensor sample rates. Further, fusion of individual sensors can be prevented by unchecking the corresponding checkbox. This can be used to simulate sensor dropout.

Some configurations produce dramatic results. For example, turning off the GPS sensor causes the position estimate to drift quickly. Turning off the magnetometer sensor will cause the orientation estimate to slowly deviate from the ground truth as the estimate rotates too fast. Conversely, if the gyroscope is turned off and the magnetometer is turned on, the estimated orientation shows a wobble and lacks the smoothness present if both sensors are used.

Turning all sensors on but setting them to run at the lowest rate produces an estimate that visibly deviates from the ground truth and then snaps back to a more correct result when sensors are fused. This is most easily seen in the HelperScrollingPlotter of the running estimate errors.

The main simulation runs at 100 Hz. Each iteration inspects the checkboxes on the figure window and, if the sensor is enabled, fuses the data for that sensor at the appropriate rate.

```
for ii=1:size(accel,1)
    fusionfilt.predict(1./Fs);

    % Fuse Accelerometer
    if (f.UserData.Accelerometer) && ...
        mod(ii, fix(Fs/f.UserData.AccelerometerSampleRate)) == 0

        fusionfilt.fuseaccel(accel(ii,:), Racc);
    end

    % Fuse Gyroscope
    if (f.UserData.Gyroscope) && ...
        mod(ii, fix(Fs/f.UserData.GyroscopeSampleRate)) == 0

        fusionfilt.fusegyro(gyro(ii,:), Rgyro);
```

```
        end

        % Fuse Magnetometer
        if (f.UserData.Magnetometer) && ...
            mod(ii, fix(Fs/f.UserData.MagnetometerSampleRate)) == 0

            fusionfilt.fusemag(mag(ii,:), Rmag);
        end

        % Fuse GPS
        if (f.UserData.GPS) && mod(ii, fix(Fs/f.UserData.GPSSampleRate)) == 0
            fusionfilt.fusegps(lla(ii,:), Rpos, gpsvel(ii,:), Rvel);
        end

        % Plot the pose error
        [p,q] = pose(fusionfilt);
        posescope(p, q, trajPos(ii,:), trajOrient(ii));

        orientErr = rad2deg(dist(q, trajOrient(ii) ));
        posErr = p - trajPos(ii,:);
        errscope(orientErr, posErr(1), posErr(2), posErr(3));
    end
```
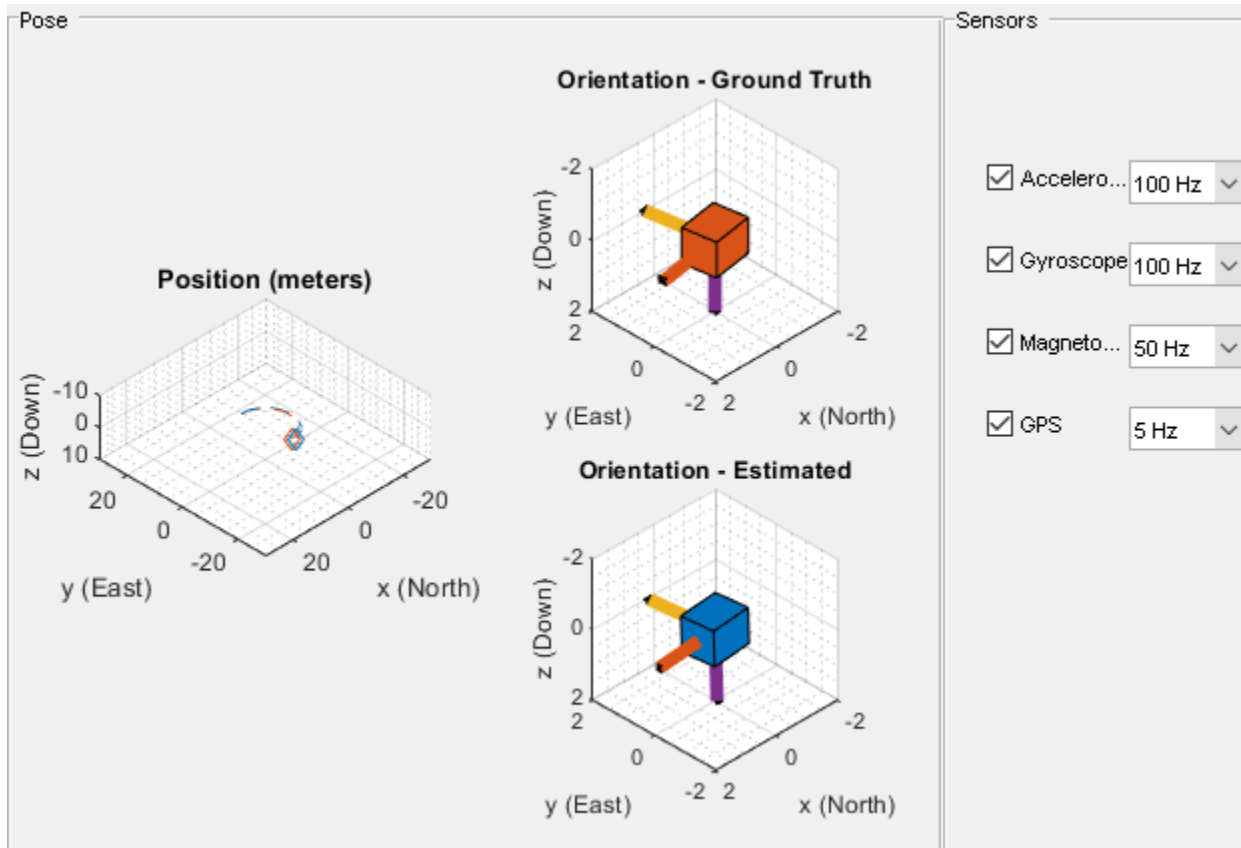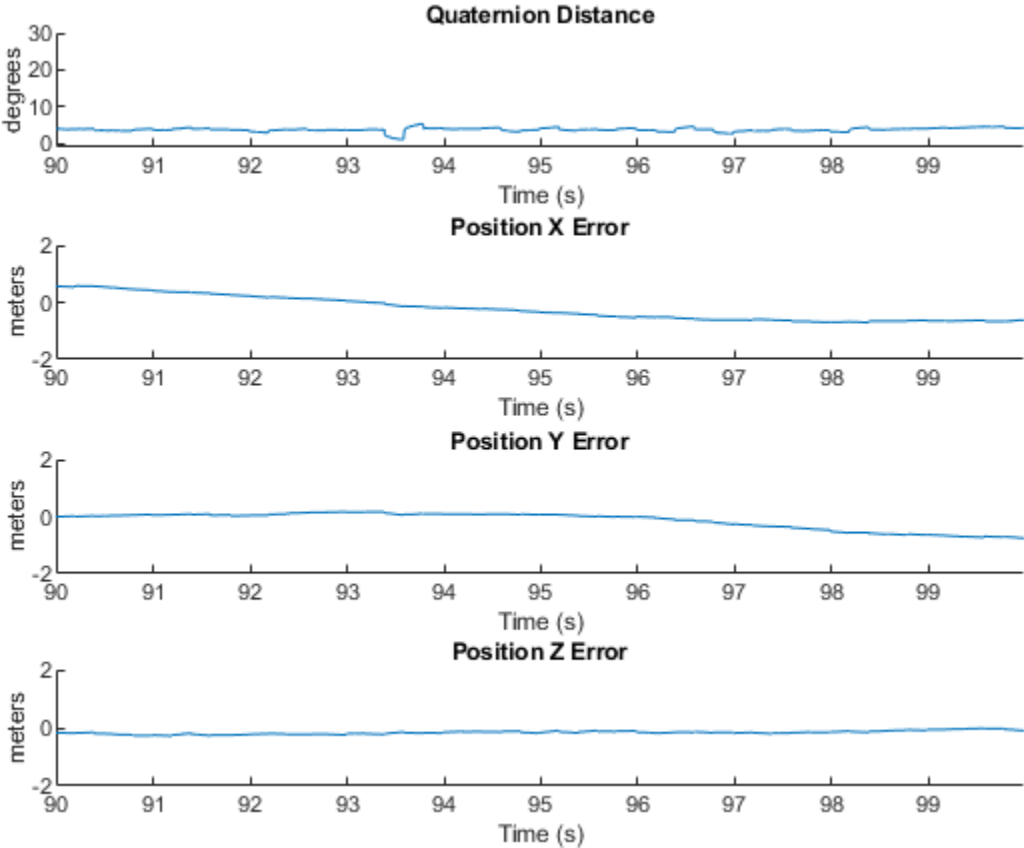
Conclusion

The `insfilterAsync` allows for various and varying sample rates. The quality of the estimated outputs depends heavily on individual sensor fusion rates. Any sensor dropout will have a profound effect on the output.

Magnetometer Calibration

Magnetometers detect magnetic field strength along a sensor's X,Y and Z axes. Accurate magnetic field measurements are essential for sensor fusion and the determination of heading and orientation.

In order to be useful for heading and orientation computation, typical low cost MEMS magnetometers need to be calibrated to compensate for environmental noise and manufacturing defects.

Ideal Magnetometers

An ideal three-axis magnetometer measures magnetic field strength along orthogonal X, Y and Z axes. Absent any magnetic interference, magnetometer readings measure the Earth's magnetic field. If magnetometer measurements are taken as the sensor is rotated through all possible orientations, the measurements should lie on a sphere. The radius of the sphere is the magnetic field strength.

To generate magnetic field samples, use the `imuSensor` object. For these purposes it is safe to assume the angular velocity and acceleration are zero at each orientation.

```
N = 500;
rng(1);
acc = zeros(N,3);
av = zeros(N,3);
q = randrot(N,1); % uniformly distributed random rotations
imu = imuSensor('accel-mag');
[~,x] = imu(acc,av,q);
scatter3(x(:,1),x(:,2),x(:,3));
axis equal
title('Ideal Magnetometer Data');
```

Ideal Magnetometer Data
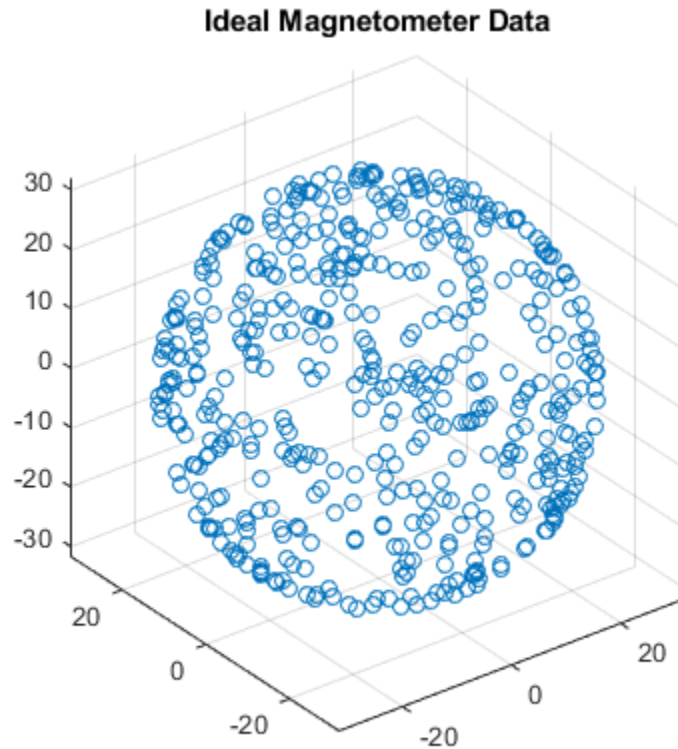
Hard Iron Effects

Noise sources and manufacturing defects degrade a magnetometer's measurement. The most striking of these are hard iron effects. Hard iron effects are stationary interfering magnetic noise sources. Often, these come from other metallic objects on the circuit board with the magnetometer. The hard iron effects shift the origin of the ideal sphere.

```
imu.Magnetometer.ConstantBias = [2 10 40];
[~,x] = imu(acc,av,q);
figure;
scatter3(x(:,1),x(:,2),x(:,3));
axis equal
title('Magnetometer Data With a Hard Iron Offset');
```

Magnetometer Data With a Hard Iron Offset
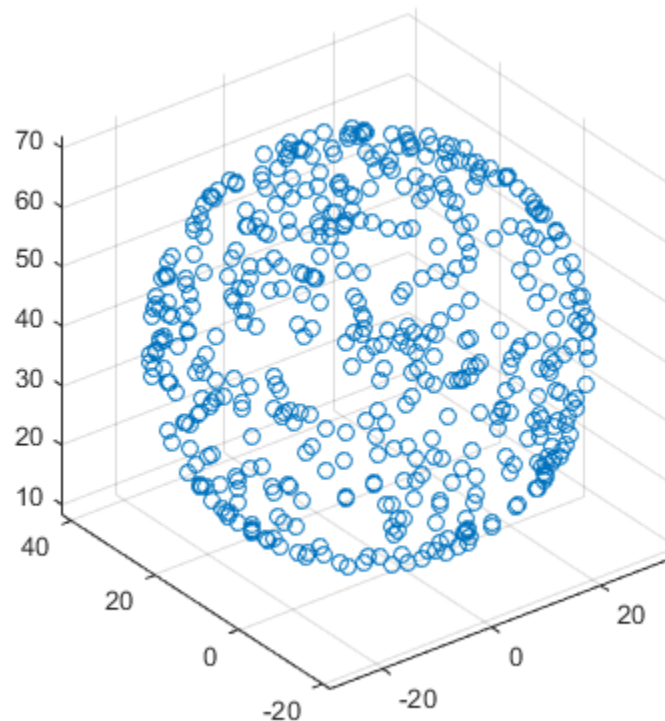
Soft Iron Effects

Soft iron effects are more subtle. They arise from objects near the sensor which distort the surrounding magnetic field. These have the effect of stretching and tilting the sphere of ideal measurements. The resulting measurements lie on an ellipsoid.

The soft iron magnetic field effects can be simulated by rotating the geomagnetic field vector of the IMU to the sensor frame, stretching it, and then rotating it back to the global frame.

```
nedmf = imu.MagneticField;
Rsoft = [2.5 0.3 0.5; 0.3 2 .2; 0.5 0.2 3];
soft = rotateframe(conj(q),rotateframe(q,nedmf)*Rsoft);

for ii=1:numel(q)
    imu.MagneticField = soft(ii,:);
    [~,x(ii,:)] = imu(acc(ii,:),av(ii,:),q(ii));
end
figure;
scatter3(x(:,1),x(:,2),x(:,3));
axis equal
title('Magnetometer Data With Hard and Soft Iron Effects');
```

Magnetometer Data With Hard and Soft Iron Effects
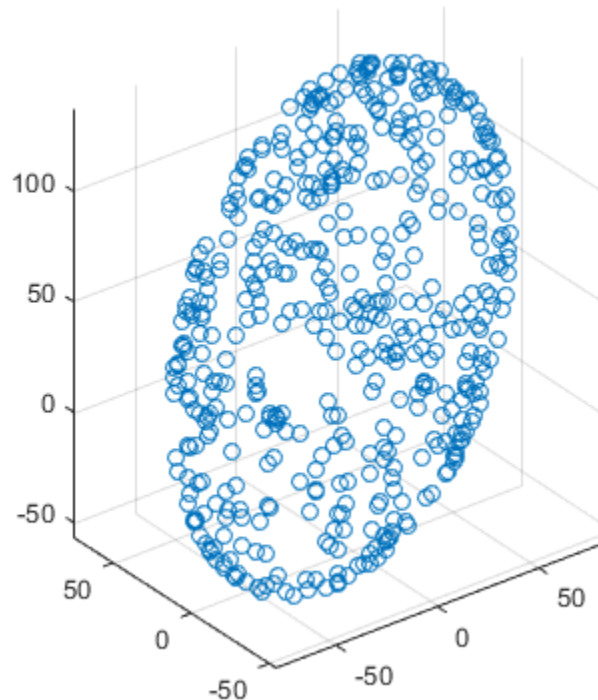
Correction Technique

The `magcal` function can be used to determine magnetometer calibration parameters that account for both hard and soft iron effects. Uncalibrated magnetometer data can be modeled as lying on an ellipsoid with equation

$$(x - b)R(x - b)^T = \beta^2$$

In this equation R is a 3-by-3 matrix, b is a 1-by-3 vector defining the ellipsoid center, x is a 1-by-3 vector of uncalibrated magnetometer measurements, and β is a scalar indicating the magnetic field strength. The above equation is the general form of a conic. For an ellipsoid, R must be positive definite. The `magcal` function uses a variety of solvers, based on different assumptions about R. In the `magcal` function, R can be assumed to be the identity matrix, a diagonal matrix, or a symmetric matrix.

The `magcal` function produces correction coefficients that take measurements which lie on an offset ellipsoid and transform them to lie on an ideal sphere, centered at the origin. The `magcal` function returns a 3-by-3 real matrix A and a 1-by-3 vector b. To correct the uncalibrated data compute

$$m = (x - b)A.$$

Here x is a 1-by-3 array of uncalibrated magnetometer measurements and m is the 1-by-3 array of corrected magnetometer measurements, which lie on a sphere. The matrix A has a determinant of 1 and is the matrix square root of R. Additionally, A has the same form as R : the identity, a diagonal, or

a symmetric matrix. Because these kinds of matrices cannot impart a rotation, the matrix *A* will not rotate the magnetometer data during correction.

The `magcal` function also returns a third output which is the magnetic field strength β. You can use the magnetic field strength to set the `ExpectedMagneticFieldStrength` property of `ahrsfilter`.

Using the `magcal` Function

Use the `magcal` function to determine calibration parameters that correct noisy magnetometer data. Create noisy magnetometer data by setting the `NoiseDensity` property of the `Magnetometer` property in the `imuSensor`. Use the rotated and stretched magnetic field in the variable `soft` to simulate soft iron effects.

```
imu.Magnetometer.NoiseDensity = 0.08;
for ii=1:numel(q)
    imu.MagneticField = soft(ii,:);
    [~,x(ii,:)] = imu(acc(ii,:),av(ii,:),q(ii));
end
```

To find the A and b parameters which best correct the uncalibrated magnetometer data, simply call the function as:

```
[A,b,expMFS]  = magcal(x);
xCorrected = (x-b)*A;
```

Plot the original and corrected data. Show the ellipsoid that best fits the original data. Show the sphere on which the corrected data should lie.

```
de = HelperDrawEllipsoid;
de.plotCalibrated(A,b,expMFS,x,xCorrected,'Auto');
```

Original Data and Best Fit Ellipsoid Using Auto Fitter

Corrected Data Fit to Ideal Sphere

- ● Data inside fit (uT)
- ● Data outside fit (uT)
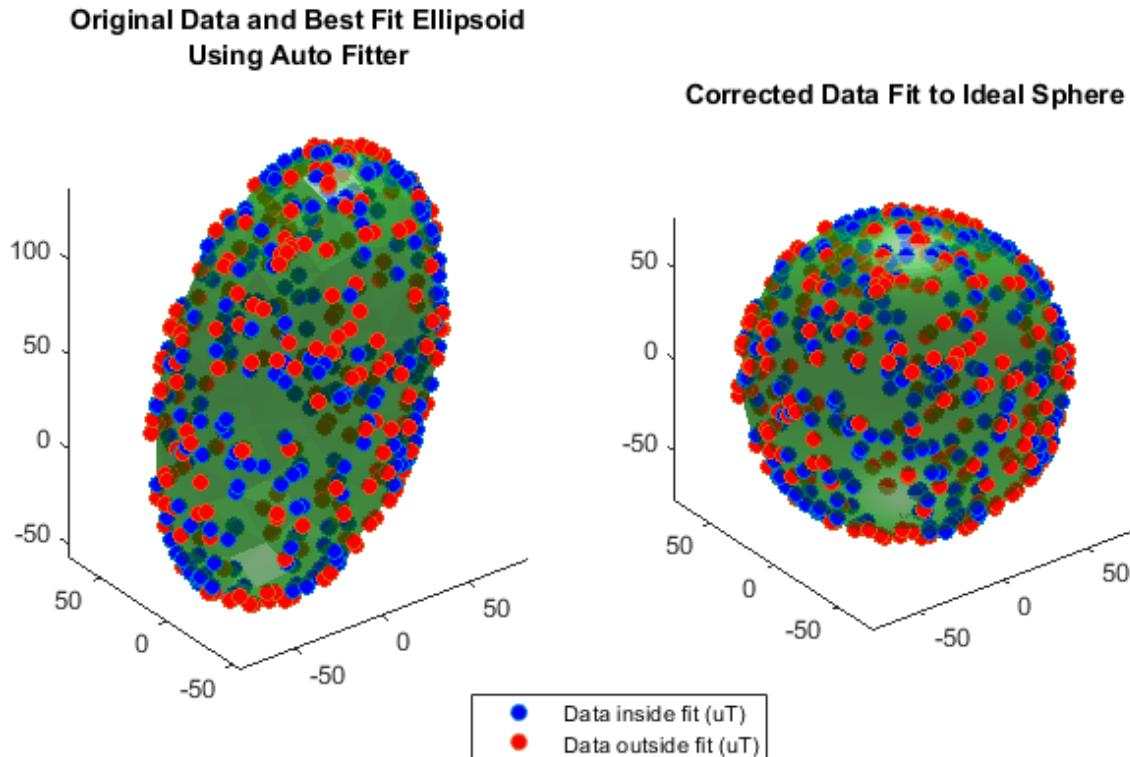
The `magcal` function uses a variety of solvers to minimize the residual error. The residual error is the sum of the distances between the calibrated data and a sphere of radius `expMFS`.

$$E = \frac{1}{2\beta^2}\sqrt{\frac{\sum ||(x-b)A||^2 - \beta^2}{N}}$$

```
r = sum(xCorrected.^2,2) - expMFS.^2;
E = sqrt(r.'*r./N)./(2*expMFS.^2);
fprintf('Residual error in corrected data : %.2f\n\n',E);
```

```
Residual error in corrected data : 0.01
```

You can run the individual solvers if only some defects need to be corrected or to achieve a simpler correction computation.

Offset-Only Computation

Many MEMS magnetometers have registers within the sensor that can be used to compensate for the hard iron offset. In effect, the (x-b) portion of the equation above happens on board the sensor. When only a hard iron offset compensation is needed, the A matrix effectively becomes the identity matrix. To determine the hard iron correction alone, the `magcal` function can be called this way:

```
[Aeye,beye,expMFSeye] = magcal(x,'eye');
xEyeCorrected = (x-beye)*Aeye;
[ax1,ax2] = de.plotCalibrated(Aeye,beye,expMFSeye,x,xEyeCorrected,'Eye');
```

```
view(ax1,[-1 0 0]);
view(ax2,[-1 0 0]);
```
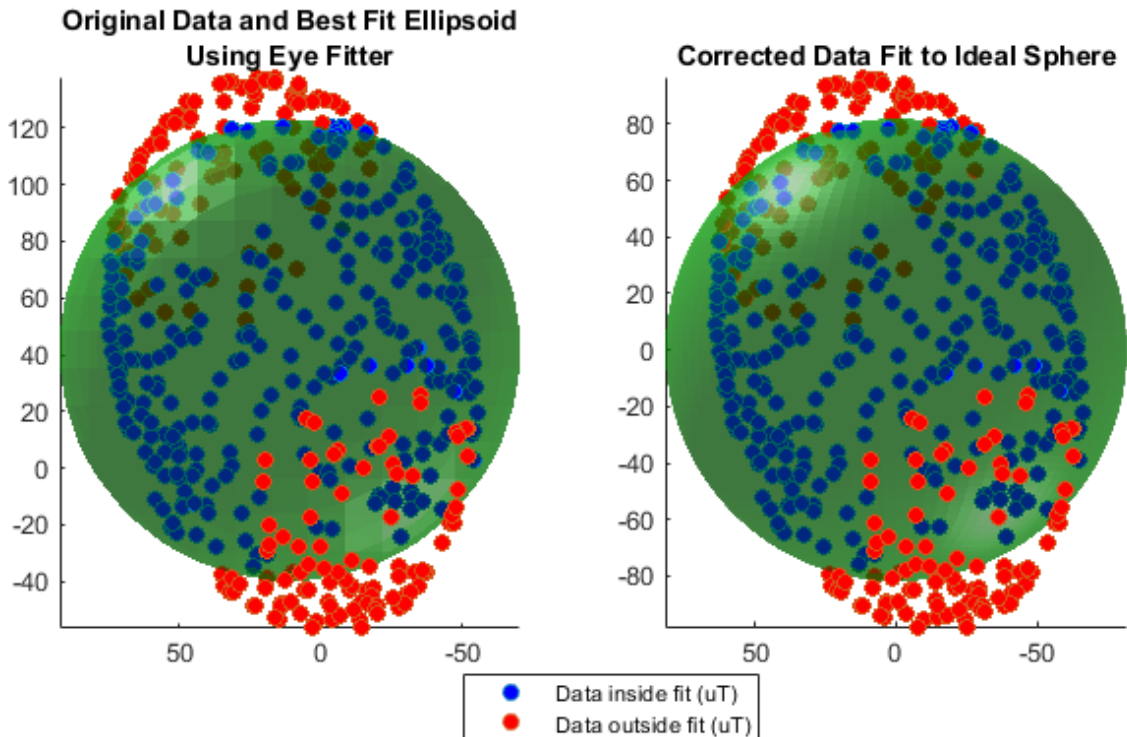

Hard Iron Compensation and Axis Scaling Computation

For many applications, treating the ellipsoid matrix as a diagonal matrix is sufficient. Geometrically, this means the ellipsoid of uncalibrated magnetometer data is approximated to have its semiaxes aligned with the coordinate system axes and a center offset from the origin. Though this is unlikely to be the actual characteristics of the ellipsoid, it reduces the correction equation to a single multiply and single subtract per axis.

```
[Adiag,bdiag,expMFSdiag] = magcal(x,'diag');
xDiagCorrected = (x-bdiag)*Adiag;
[ax1,ax2] = de.plotCalibrated(Adiag,bdiag,expMFSdiag,x,xDiagCorrected,...
    'Diag');
```

Original Data and Best Fit Ellipsoid Using Diag Fitter

Corrected Data Fit to Ideal Sphere

- Data inside fit (uT)
- Data outside fit (uT)
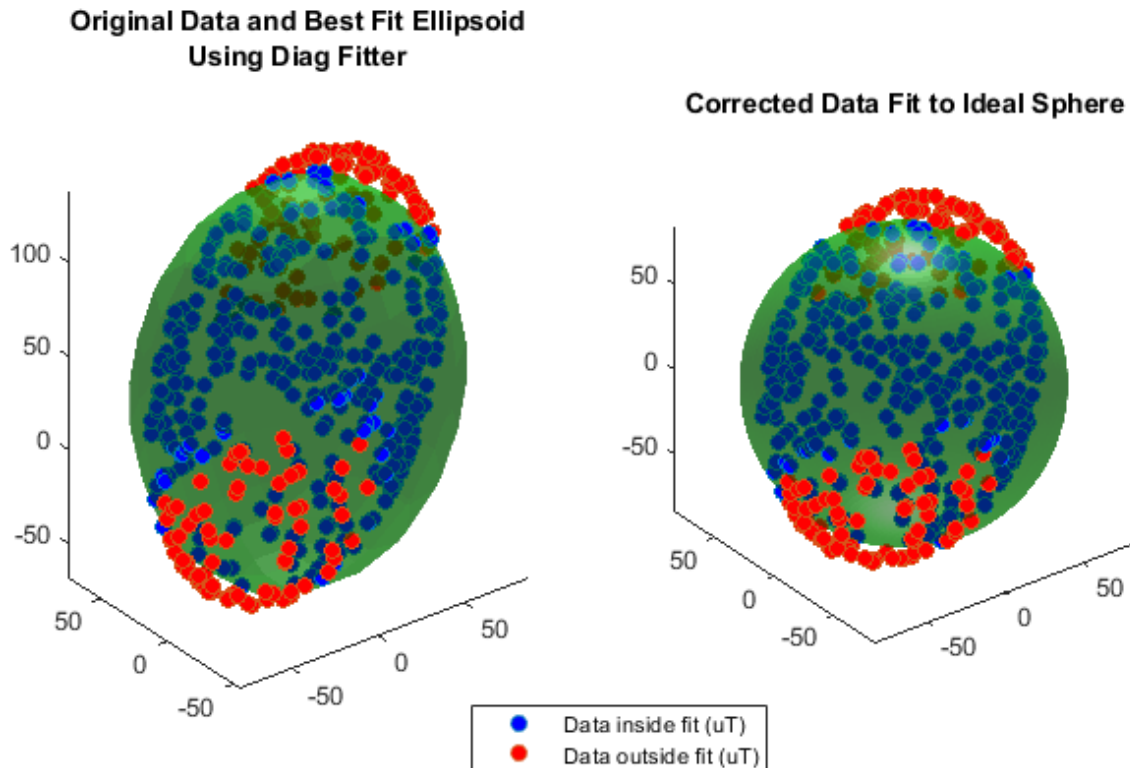
Full Hard and Soft Iron Compensation

To force the `magcal` function to solve for an arbitrary ellipsoid and produce a dense, symmetric A matrix, call the function as:

```
[A,b] = magcal(x,'sym');
```

Auto Fit

The `'eye'`, `'diag'`, and `'sym'` flags should be used carefully and the output values inspected. In some cases, there may be insufficient data for a high order (`'diag'` or `'sym'`) fit and a better set of correction parameters can be found using a simpler A matrix. The `'auto'` fit option, which is the default, handles this situation.

Consider the case when insufficient data is used with a high order fitter.

```
xidx = x(:,3) > 100;
xpoor = x(xidx,:);
[Apoor,bpoor,mfspoor] = magcal(xpoor,'diag');
```

There is not enough data spread over the surface of the ellipsoid to achieve a good fit and proper calibration parameters with the `'diag'` option. As a result, the Apoor matrix is complex.

```
disp(Apoor)

   0.0000 + 0.4722i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.5981i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   3.5407 + 0.0000i
```

Using the `'auto'` fit option avoids this problem and finds a simpler A matrix which is real, symmetric, and positive definite. Calling `magcal` with the `'auto'` option string is the same as calling without any option string.

```
[Abest,bbest,mfsbest] = magcal(xpoor,'auto');
disp(Abest)

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

Comparing the results of using the `'auto'` fitter and an incorrect, high order fitter show the perils of not examining the returned A matrix before correcting the data.

```
de.compareBest(Abest,bbest,mfsbest,Apoor,bpoor,mfspoor,xpoor);
```
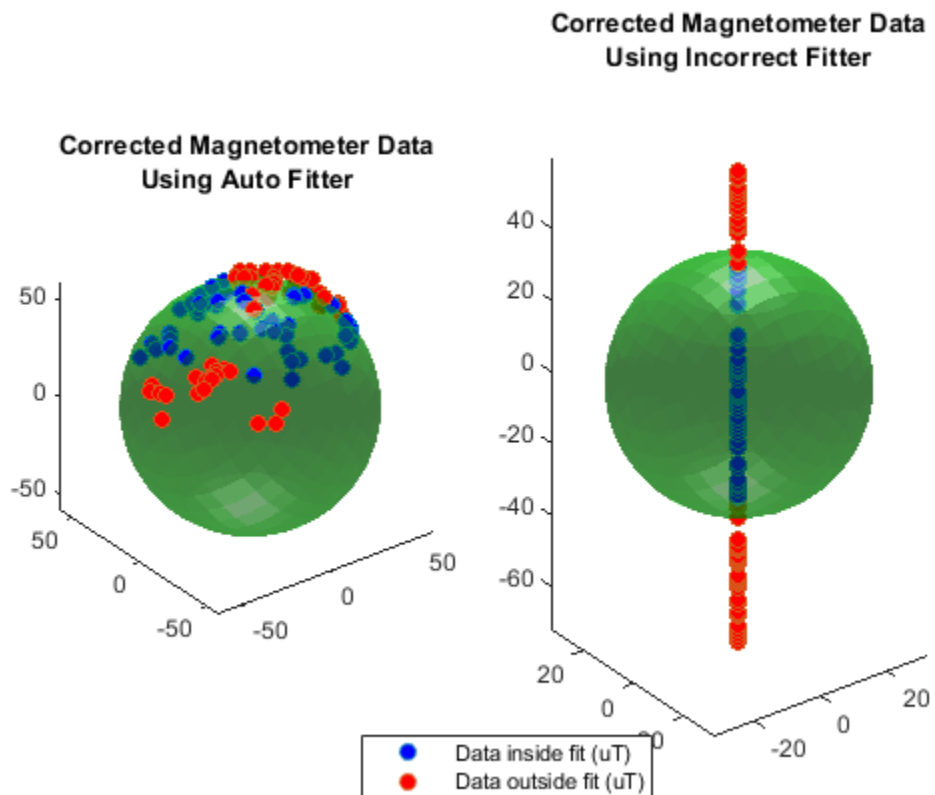

Calling the `magcal` function with the `'auto'` flag, which is the default, will try all possibilities of `'eye'`, `'diag'` and `'sym'` searching for the A and b which minimizes the residual error, keeps A real, and ensures R is positive definite and symmetric.

Conclusion

The `magcal` function can give calibration parameters to correct hard and soft iron offsets in a magnetometer. Calling the function with no option string, or equivalently the `'auto'` option string, produces the best fit and covers most cases.

Track Vehicles Using Lidar: From Point Cloud to Track List

This example shows you how to track vehicles using measurements from a lidar sensor mounted on top of an ego vehicle. Lidar sensors report measurements as a point cloud. The example illustrates the workflow in MATLAB® for processing the point cloud and tracking the objects. For a Simulink® version of the example, refer to "Track Vehicles Using Lidar Data in Simulink" on page 5-403.The lidar data used in this example is recorded from a highway driving scenario. In this example, you use the recorded data to track vehicles with a joint probabilistic data association (JPDA) tracker and an interacting multiple model (IMM) approach.

3-D Bounding Box Detector Model

Due to high resolution capabilities of the lidar sensor, each scan from the sensor contains a large number of points, commonly known as a point cloud. This raw data must be preprocessed to extract objects of interest, such as cars, cyclists, and pedestrians. In this example, you use a classical segmentation algorithm using a distance-based clustering algorithm. For more details about segmentation of lidar data into objects such as the ground plane and obstacles, refer to the "Ground Plane and Obstacle Detection Using Lidar" (Automated Driving Toolbox) example. For a deep learning segmentation workflow, refer to the "Detect, Classify, and Track Vehicles Using Lidar" (Lidar Toolbox) example. In this example, the point clouds belonging to obstacles are further classified into clusters using the `pcsegdist` function, and each cluster is converted to a bounding box detection with the following format:

$$[x \ y \ z \ \theta \ l \ w \ h]$$

x, y and z refer to the x-, y- and z-positions of the bounding box, θ refers to its yaw angle and l, w and h refer to its length, width, and height, respectively. The `pcfitcuboid` (Lidar Toolbox) function uses L-shape fitting algorithm to determine the yaw angle of the bounding box.

The detector is implemented by a supporting class `HelperBoundingBoxDetector`, which wraps around point cloud segmentation and clustering functionalities. An object of this class accepts a `pointCloud` input and returns a list of `objectDetection` objects with bounding box measurements.

The diagram shows the processes involved in the bounding box detector model and the Lidar Toolbox™ functions used to implement each process. It also shows the properties of the supporting class that control each process.
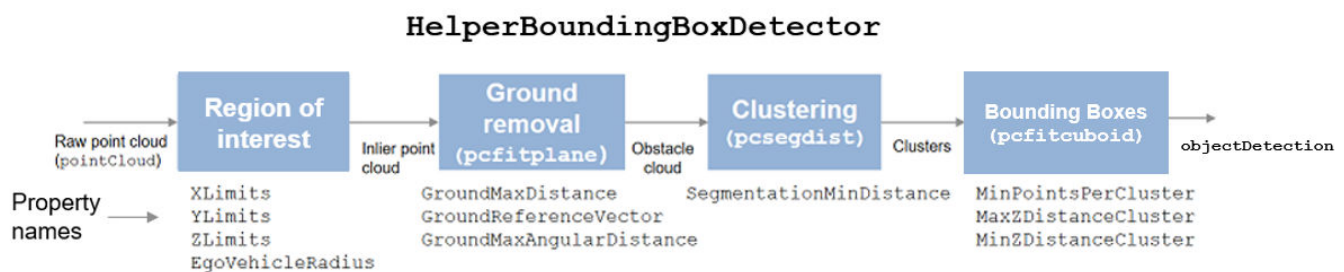
The lidar data is available at the following location: https://ssd.mathworks.com/supportfiles/lidar/data/TrackVehiclesUsingLidarExampleData.zip

Download the data files into your temporary directory, whose location is specified by MATLAB's `tempdir` function. If you want to place the files in a different folder, change the directory name in the subsequent instructions.

```
% Load data if unavailable. The lidar data is stored as a cell array of
% pointCloud objects.
if ~exist('lidarData','var')
    dataURL = 'https://ssd.mathworks.com/supportfiles/lidar/data/TrackVehiclesUsingLidarExampleDa
    datasetFolder = fullfile(tempdir,'LidarExampleDataset');
    if ~exist(datasetFolder,'dir')
        unzip(dataURL,datasetFolder);
    end
    % Specify initial and final time for simulation.
    initTime = 0;
    finalTime = 35;
    [lidarData, imageData] = loadLidarAndImageData(datasetFolder,initTime,finalTime);
end

% Set random seed to generate reproducible results.
S = rng(2018);

% A bounding box detector model.
detectorModel = HelperBoundingBoxDetector(...
    'XLimits',[-50 75],...                   % min-max
    'YLimits',[-5 5],...                     % min-max
    'ZLimits',[-2 5],...                     % min-max
    'SegmentationMinDistance',1.8,...        % minimum Euclidian distance
    'MinDetectionsPerCluster',1,...          % minimum points per cluster
    'MeasurementNoise',blkdiag(0.25*eye(3),25,eye(3)),...      % measurement noise in detection
    'GroundMaxDistance',0.3);                % maximum distance of ground points from ground plane
```

Target State and Sensor Measurement Model

The first step in tracking an object is defining its state, and the models that define the transition of state and the corresponding measurement. These two sets of equations are collectively known as the state-space model of the target. To model the state of vehicles for tracking using lidar, this example uses a cuboid model with following convention:

$$x = [x_{kin} \ \theta \ l \ w \ h]$$

x_{kin} refers to the portion of the state that controls the kinematics of the motion center, and θ is the yaw angle. The length, width, height of the cuboid are modeled as a constants, whose estimates evolve in time during correction stages of the filter.

In this example, you use two state-space models: a constant velocity (cv) cuboid model and a constant turn-rate (ct) cuboid model. These models differ in the way they define the kinematic part of the state, as described below:

$$x_{cv} = [x \ \dot{x} \ y \ \dot{y} \ z \ \dot{z} \ \theta \ l \ w \ h]$$

$$x_{ct} = [x \ \dot{x} \ y \ \dot{y} \ \dot{\theta} \ z \ \dot{z} \ \theta \ l \ w \ h]$$

For information about their state transition, refer to the `helperConstvelCuboid` and `helperConstturnCuboid` functions used in this example.

The `helperCvmeasCuboid` and `helperCtmeasCuboid` measurement models describe how the sensor perceives the constant velocity and constant turn-rate states respectively, and they return

bounding box measurements. Because the state contains information about size of the target, the measurement model includes the effect of center-point offset and bounding box shrinkage, as perceived by the sensor, due to effects like self-occlusion [1]. This effect is modeled by a shrinkage factor that is directly proportional to the distance from the tracked vehicle to the sensor.

The image below demonstrates the measurement model operating at different state-space samples. Notice the modeled effects of bounding box shrinkage and center-point offset as the objects move around the ego vehicle.
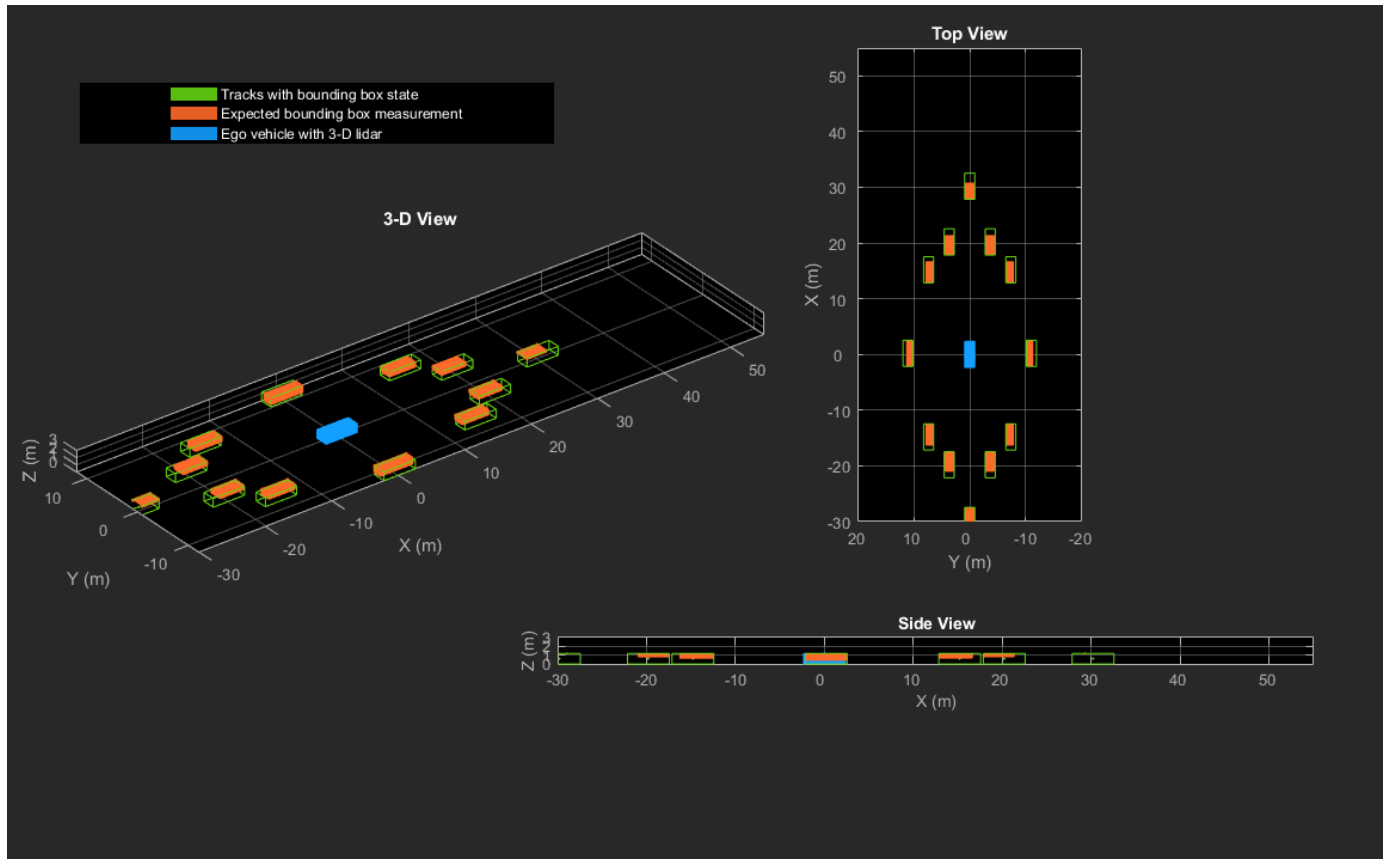
Set Up Tracker and Visualization

The image below shows the complete workflow to obtain a list of tracks from a pointCloud input.
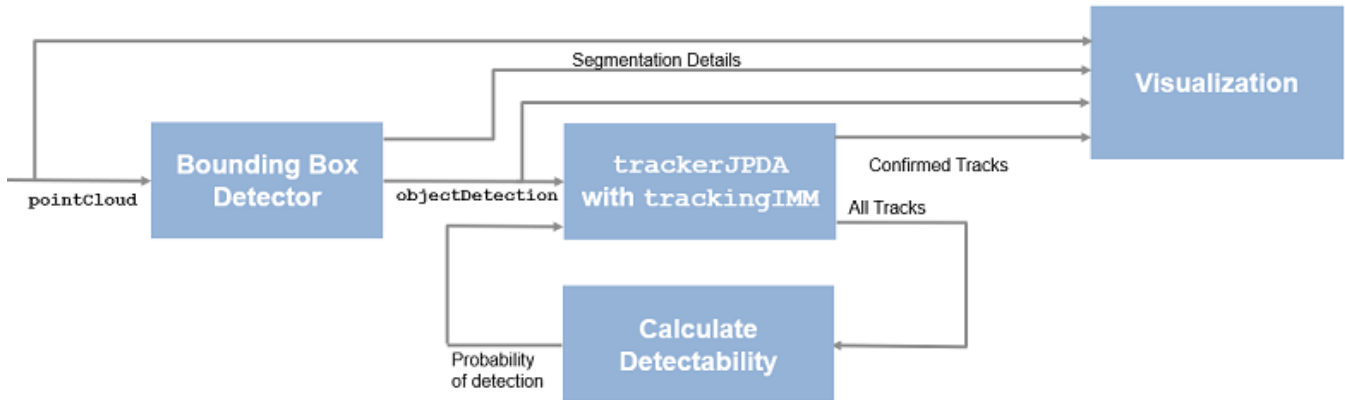
Now, set up the tracker and the visualization used in the example.

A joint probabilistic data association tracker (`trackerJPDA`) coupled with an IMM filter (`trackingIMM`) is used to track objects in this example. The IMM filter uses a constant velocity and constant turn-rate model and is initialized using the supporting function, `helperInitIMMFilter`, included with this example. The IMM approach helps a track to switch between motion models and thus achieve good estimation accuracy during events like maneuvering or lane changing. The animation below shows the effect of mixing the constant velocity and constant turn-rate model during prediction stages of the IMM filter.
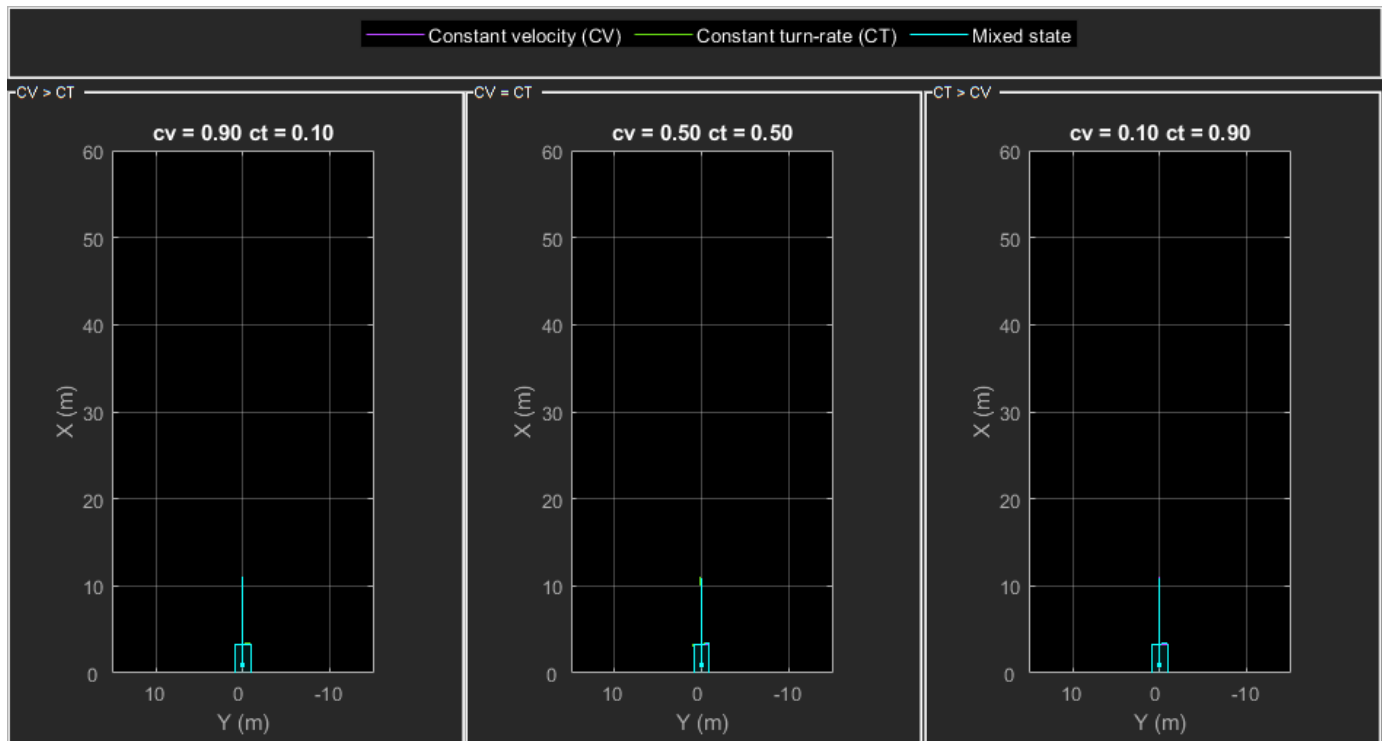
The IMM filter updates the probability of each model when it is corrected with detections from the object. The animation below shows the estimated trajectory of a vehicle during a lane change event and the corresponding estimated probabilities of each model.
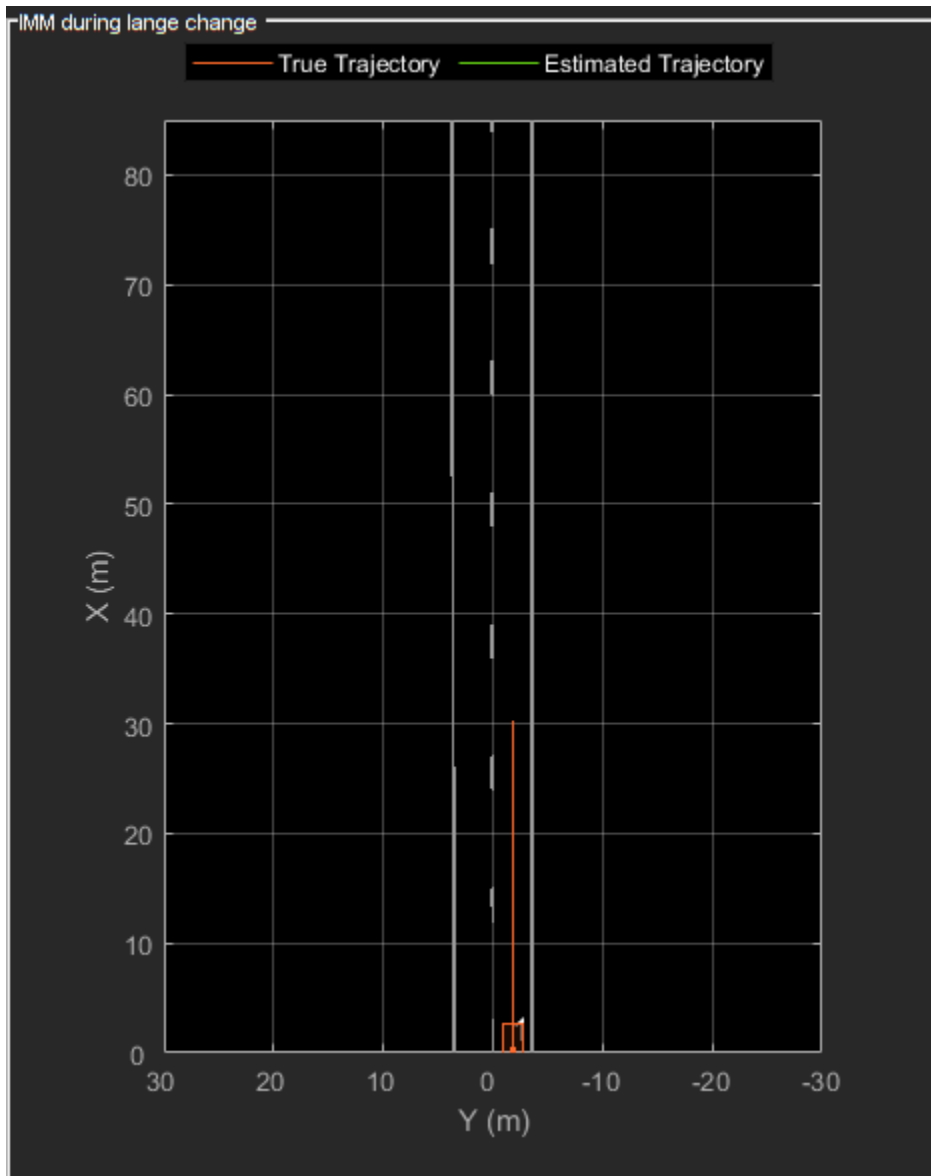
Set the `HasDetectableTrackIDsInput` property of the tracker as `true`, which enables you to specify a state-dependent probability of detection. The detection probability of a track is calculated by the `helperCalcDetectability` function, listed at the end of this example.

```
assignmentGate = [75 1000]; % Assignment threshold;
confThreshold = [7 10];    % Confirmation threshold for history logic
delThreshold = [8 10];     % Deletion threshold for history logic
Kc = 1e-9;                 % False-alarm rate per unit volume

% IMM filter initialization function
filterInitFcn = @helperInitIMMFilter;

% A joint probabilistic data association tracker with IMM filter
tracker = trackerJPDA('FilterInitializationFcn',filterInitFcn,...
    'TrackLogic','History',...
    'AssignmentThreshold',assignmentGate,...
```

```
    'ClutterDensity',Kc,...
    'ConfirmationThreshold',confThreshold,...
    'DeletionThreshold',delThreshold,...
    'HasDetectableTrackIDsInput',true,...
    'InitializationThreshold',0,...
    'HitMissThreshold',0.1);
```

The visualization is divided into these main categories:

1   Lidar Preprocessing and Tracking - This display shows the raw point cloud, segmented ground, and obstacles. It also shows the resulting detections from the detector model and the tracks of vehicles generated by the tracker.

2   Ego Vehicle Display - This display shows the 2-D bird's-eye view of the scenario. It shows the obstacle point cloud, bounding box detections, and the tracks generated by the tracker. For reference, it also displays the image recorded from a camera mounted on the ego vehicle and its field of view.

3   Tracking Details - This display shows the scenario zoomed around the ego vehicle. It also shows finer tracking details, such as error covariance in estimated position of each track and its motion model probabilities, denoted by cv and ct.

```
% Create display
displayObject = HelperLidarExampleDisplay(imageData{1},...
    'PositionIndex',[1 3 6],...
    'VelocityIndex',[2 4 7],...
    'DimensionIndex',[9 10 11],...
    'YawIndex',8,...
    'MovieName','',...  % Specify a movie name to record a movie.
    'RecordGIF',false); % Specify true to record new GIFs
```

Loop Through Data

Loop through the recorded lidar data, generate detections from the current point cloud using the detector model and then process the detections using the tracker.

```
time = 0;       % Start time
dT = 0.1;       % Time step

% Initiate all tracks.
allTracks = struct([]);

% Initiate variables for comparing MATLAB and MEX simulation.
numTracks = zeros(numel(lidarData),2);

% Loop through the data
for i = 1:numel(lidarData)
    % Update time
    time = time + dT;

    % Get current lidar scan
    currentLidar = lidarData{i};

    % Generator detections from lidar scan.
    [detections,obstacleIndices,groundIndices,croppedIndices] = detectorModel(currentLidar,time)

    % Calculate detectability of each track.
    detectableTracksInput = helperCalcDetectability(allTracks,[1 3 6]);
```

```matlab
    % Pass detections to track.
    [confirmedTracks,tentativeTracks,allTracks,info] = tracker(detections,time,detectableTracksIn
    numTracks(i,1) = numel(confirmedTracks);

    % Get model probabilities from IMM filter of each track using
    % getTrackFilterProperties function of the tracker.
    modelProbs = zeros(2,numel(confirmedTracks));
    for k = 1:numel(confirmedTracks)
        c1 = getTrackFilterProperties(tracker,confirmedTracks(k).TrackID,'ModelProbabilities');
        modelProbs(:,k) = c1{1};
    end

    % Update display
    if isvalid(displayObject.PointCloudProcessingDisplay.ObstaclePlotter)
        % Get current image scan for reference image
        currentImage = imageData{i};

        % Update display object
        displayObject(detections,confirmedTracks,currentLidar,obstacleIndices,...
            groundIndices,croppedIndices,currentImage,modelProbs);
    end

    % Snap a figure at time = 18
    if abs(time - 18) < dT/2
        snapnow(displayObject);
    end
end

% Write movie if requested
if ~isempty(displayObject.MovieName)
    writeMovie(displayObject);
end

% Write new GIFs if requested.
if displayObject.RecordGIF
    % second input is start frame, third input is end frame and last input
    % is a character vector specifying the panel to record.
    writeAnimatedGIF(displayObject,10,170,'trackMaintenance','ego');
    writeAnimatedGIF(displayObject,310,330,'jpda','processing');
    writeAnimatedGIF(displayObject,120,140,'imm','details');
end
```
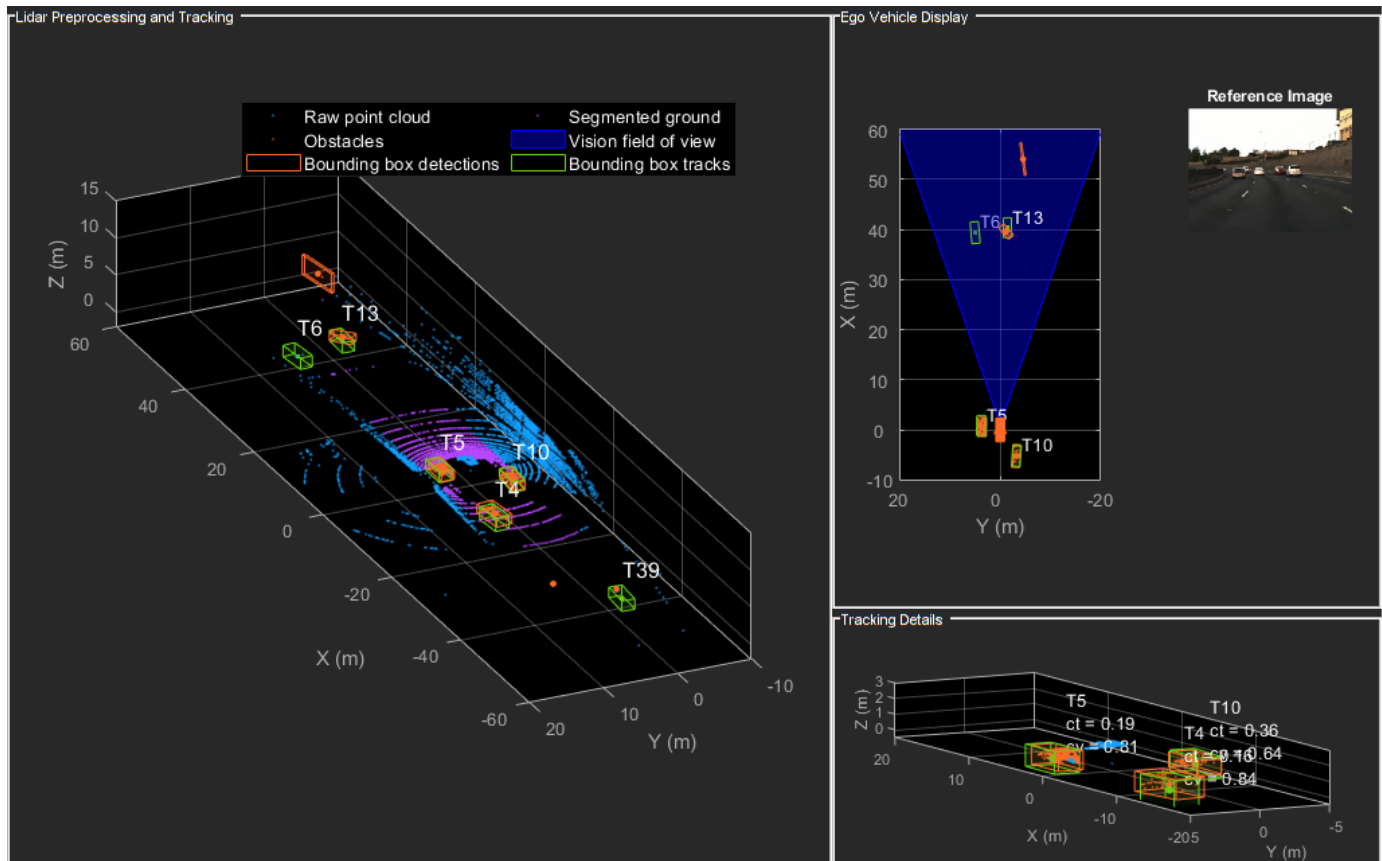
The figure above shows the three displays at time = 18 seconds. The tracks are represented by green bounding boxes. The bounding box detections are represented by orange bounding boxes. The detections also have orange points inside them, representing the point cloud segmented as obstacles. The segmented ground is shown in purple. The cropped or discarded point cloud is shown in blue.

Generate C Code

You can generate C code from the MATLAB® code for the tracking and the preprocessing algorithm using MATLAB Coder™. C code generation enables you to accelerate MATLAB code for simulation. To generate C code, the algorithm must be restructured as a MATLAB function, which can be compiled into a MEX file or a shared library. For this purpose, the point cloud processing algorithm and the tracking algorithm is restructured into a MATLAB function, `mexLidarTracker`. Some variables are defined as `persistent` to preserve their state between multiple calls to the function (see `persistent`). The inputs and outputs of the function can be observed in the function description provided in the "Supporting Files" section at the end of this example.

MATLAB coder requires specifying the properties of all the input arguments. An easy way to do this is by defining the input properties by example at the command line using the `-args` option. For more information, see "Define Input Properties by Example at the Command Line" (MATLAB Coder). Note that the top-level input arguments cannot be objects of the `handle` class. Therefore, the function accepts the *x*, *y* and *z* locations of the point cloud as an input. From the stored point cloud, this information can be extracted using the `Location` property of the `pointCloud` object. This information is also directly available as the raw data from the lidar sensor.

```
% Input lists
inputExample = {lidarData{1}.Location, 0};

% Create configuration for MEX generation
cfg = coder.config('mex');

% Replace cfg with the following to generate static library and perform
% software-in-the-loop simulation. This requires Embedded Coder license.
%
% cfg = coder.config('lib'); % Static library
% cfg.VerificationMode = 'SIL'; % Software-in-the-loop

% Generate code if file does not exist.
if ~exist('mexLidarTracker_mex','file')
    h = msgbox({'Generating code. This may take a few minutes...';'This message box will close wh
    % -config allows specifying the codegen configuration
    % -o allows specifying the name of the output file
    codegen -config cfg -o mexLidarTracker_mex mexLidarTracker -args inputExample
    close(h);
else
    clear mexLidarTracker_mex;
end
```

Rerun simulation with MEX Code

Rerun the simulation using the generated MEX code, mexLidarTracker_mex. Reset time

```
time = 0;

for i = 1:numel(lidarData)
    time = time + dT;

    currentLidar = lidarData{i};

    [detectionsMex,obstacleIndicesMex,groundIndicesMex,croppedIndicesMex,...
        confirmedTracksMex, modelProbsMex] = mexLidarTracker_mex(currentLidar.Location,time);

    % Record data for comparison with MATLAB execution.
    numTracks(i,2) = numel(confirmedTracksMex);
end
```

Compare results between MATLAB and MEX Execution

```
disp(isequal(numTracks(:,1),numTracks(:,2)));
```

```
    1
```

Notice that the number of confirmed tracks is the same for MATLAB and MEX code execution. This assures that the lidar preprocessing and tracking algorithm returns the same results with generated C code as with the MATLAB code.

Results

Now, analyze different events in the scenario and understand how the combination of lidar measurement model, joint probabilistic data association, and interacting multiple model filter, helps achieve a good estimation of the vehicle tracks.

Track Maintenance
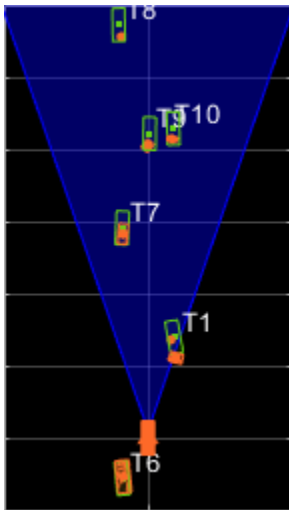
The animation above shows the simulation between time = 3 seconds and time = 16 seconds. Notice that tracks such as T10 and T6 maintain their IDs and trajectory during the time span. However, track T9 is lost because the tracked vehicle was missed (not detected) for a long time by the sensor. Also, notice that the tracked objects are able to maintain their shape and kinematic center by positioning the detections onto the visible portions of the vehicles. For example, as Track T7 moves forward, bounding box detections start to fall on its visible rear portion and the track maintains the actual size of the vehicle. This illustrates the offset and shrinkage effect modeled in the measurement functions.

Capturing Maneuvers
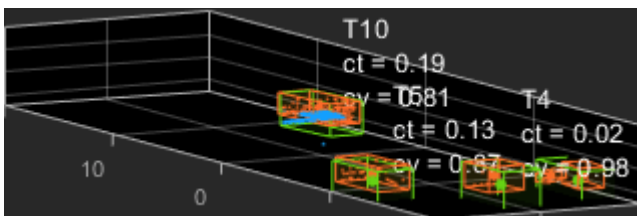
The animation shows that using an IMM filter helps the tracker to maintain tracks on maneuvering vehicles. Notice that the vehicle tracked by T4 changes lanes behind the ego vehicle. The tracker is able maintain a track on the vehicle during this maneuvering event. Also notice in the display that its probability of following the constant turn model, denoted by ct, increases during the lane change maneuver.

Joint Probabilistic Data Association
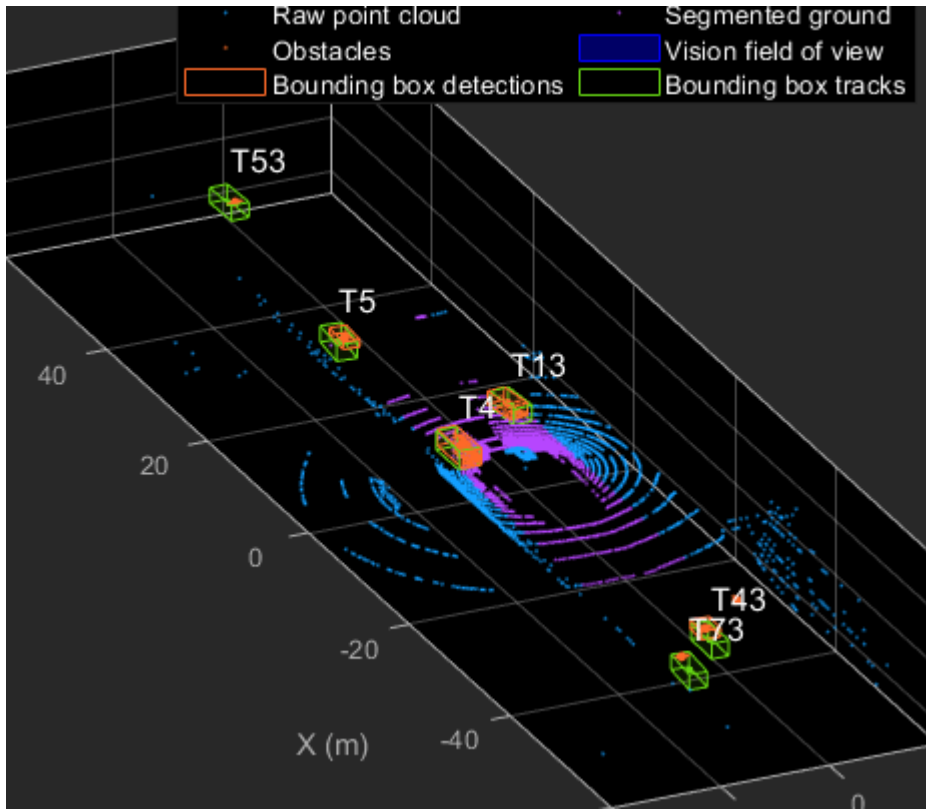
This animation shows that using a joint probabilistic data association tracker helps in maintaining tracks during ambiguous situations. Here, vehicles tracked by T43 and T73, have a low probability of detection due to their large distance from the sensor. Notice that the tracker is able to maintain tracks during events when one of the vehicles is not detected. During the event, the tracks first coalesce, which is a known phenomenon in JPDA, and then separate as soon as the vehicle was detected again.

Summary

This example showed how to use a JPDA tracker with an IMM filter to track objects using a lidar sensor. You learned how a raw point cloud can be preprocessed to generate detections for conventional trackers, which assume one detection per object per sensor scan. You also learned how to define a cuboid model to describe the kinematics, dimensions, and measurements of extended objects being tracked by the JPDA tracker. In addition, you generated C code from the algorithm and verified its execution results with the MATLAB simulation.

Supporting Files

helperLidarModel

This function defines the lidar model to simulate shrinkage of the bounding box measurement and center-point offset. This function is used in the `helperCvmeasCuboid` and `helperCtmeasCuboid` functions to obtain bounding box measurement from the state.

```
function meas = helperLidarModel(pos,dim,yaw)
% This function returns the expected bounding box measurement given an
% object's position, dimension, and yaw angle.
```

```
% Copyright 2019 The MathWorks, Inc.

% Get x,y and z.
x = pos(1,:);
y = pos(2,:);
z = pos(3,:) - 2; % lidar mounted at height = 2 meters.

% Get spherical measurement.
[az,~,r] = cart2sph(x,y,z);

% Shrink rate
s = 3/50; % 3 meters radial length at 50 meters.
sz = 2/50; % 2 meters height at 50 meters.

% Get length, width and height.
L = dim(1,:);
W = dim(2,:);
H = dim(3,:);

az = az - deg2rad(yaw);

% Shrink length along radial direction.
Lshrink = min(L,abs(s*r.*(cos(az))));
Ls = L - Lshrink;

% Shrink width along radial direction.
Wshrink = min(W,abs(s*r.*(sin(az))));
Ws = W - Wshrink;

% Shrink height.
Hshrink = min(H,sz*r);
Hs = H - Hshrink;

% Similar shift is for x and y directions.
shiftX = Lshrink.*cosd(yaw) + Wshrink.*sind(yaw);
shiftY = Lshrink.*sind(yaw) + Wshrink.*cosd(yaw);
shiftZ = Hshrink;

% Modeling the affect of box origin offset
x = x - sign(x).*shiftX/2;
y = y - sign(y).*shiftY/2;
z = z + shiftZ/2 + 2;

% Measurement format
meas = [x;y;z;yaw;Ls;Ws;Hs];

end
```

helperInverseLidarModel

This function defines the inverse lidar model to initiate a tracking filter using a lidar bounding box measurement. This function is used in the `helperInitIMMFilter` function to obtain state estimates from a bounding box measurement.

```
function [pos,posCov,dim,dimCov,yaw,yawCov] = helperInverseLidarModel(meas,measCov)
```

```
% This function returns the position, dimension, yaw using a bounding
% box measurement.

% Copyright 2019 The MathWorks, Inc.

% Shrink rate.
s = 3/50;
sz = 2/50;

% x,y and z of measurement
x = meas(1,:);
y = meas(2,:);
z = meas(3,:);

[az,~,r] = cart2sph(x,y,z);

% Shift x and y position.
Lshrink = abs(s*r.*(cos(az)));
Wshrink = abs(s*r.*(sin(az)));
Hshrink = sz*r;

shiftX = Lshrink;
shiftY = Wshrink;
shiftZ = Hshrink;

x = x + sign(x).*shiftX/2;
y = y + sign(y).*shiftY/2;
z = z - shiftZ/2;

pos = [x;y;z];
posCov = measCov(1:3,1:3,:);

yaw = meas(4,:);
yawCov = measCov(4,4,:);

% Dimensions are initialized for a standard passenger car with low
% uncertainty.
dim = [4.7;1.8;1.4];
dimCov = 0.01*eye(3);
end
```

HelperBoundingBoxDetector

This is the supporting class `HelperBoundingBoxDetector` to accept a point cloud input and return a list of `objectDetection`

```
classdef HelperBoundingBoxDetector < matlab.System
    % HelperBoundingBoxDetector A helper class to segment the point cloud
    % into bounding box detections.
    % The step call to the object does the following things:
    %
    % 1. Removes point cloud outside the limits.
    % 2. From the survived point cloud, segments out ground
    % 3. From the obstacle point cloud, forms clusters and puts bounding
    %    box on each cluster.
```

```matlab
        % Cropping properties
        properties
            % XLimits XLimits for the scene
            XLimits = [-70 70];
            % YLimits YLimits for the scene
            YLimits = [-6 6];
            % ZLimits ZLimits fot the scene
            ZLimits = [-2 10];
        end

        % Ground Segmentation Properties
        properties
            % GroundMaxDistance Maximum distance of point to the ground plane
            GroundMaxDistance = 0.3;
            % GroundReferenceVector Reference vector of ground plane
            GroundReferenceVector = [0 0 1];
            % GroundMaxAngularDistance Maximum angular distance of point to reference vector
            GroundMaxAngularDistance = 5;
        end

        % Bounding box Segmentation properties
        properties
            % SegmentationMinDistance Distance threshold for segmentation
            SegmentationMinDistance = 1.6;
            % MinDetectionsPerCluster Minimum number of detections per cluster
            MinDetectionsPerCluster = 2;
            % MaxZDistanceCluster Maximum Z-coordinate of cluster
            MaxZDistanceCluster = 3;
            % MinZDistanceCluster Minimum Z-coordinate of cluster
            MinZDistanceCluster = -3;
        end

        % Ego vehicle radius to remove ego vehicle point cloud.
        properties
            % EgoVehicleRadius Radius of ego vehicle
            EgoVehicleRadius = 3;
        end

        properties
            % MeasurementNoise Measurement noise for the bounding box detection
            MeasurementNoise = blkdiag(eye(3),10,eye(3));
        end

        properties (Nontunable)
            MeasurementParameters = struct.empty(0,1);
        end

        methods
            function obj = HelperBoundingBoxDetector(varargin)
                setProperties(obj,nargin,varargin{:})
            end
        end

        methods (Access = protected)
            function [bboxDets,obstacleIndices,groundIndices,croppedIndices] = stepImpl(obj,currentP
                % Crop point cloud
                [pcSurvived,survivedIndices,croppedIndices] = cropPointCloud(currentPointCloud,obj.XI
                % Remove ground plane
```

```
            [pcObstacles,obstacleIndices,groundIndices] = removeGroundPlane(pcSurvived,obj.Ground
            % Form clusters and get bounding boxes
            detBBoxes = getBoundingBoxes(pcObstacles,obj.SegmentationMinDistance,obj.MinDetection
            % Assemble detections
            if isempty(obj.MeasurementParameters)
                measParams = {};
            else
                measParams = obj.MeasurementParameters;
            end
            bboxDets = assembleDetections(detBBoxes,obj.MeasurementNoise,measParams,time);
        end
    end
end

function detections = assembleDetections(bboxes,measNoise,measParams,time)
% This method assembles the detections in objectDetection format.
numBoxes = size(bboxes,2);
detections = cell(numBoxes,1);
for i = 1:numBoxes
    detections{i} = objectDetection(time,cast(bboxes(:,i),'double'),...
        'MeasurementNoise',double(measNoise),'ObjectAttributes',struct,...
        'MeasurementParameters',measParams);
end
end

function bboxes = getBoundingBoxes(ptCloud,minDistance,minDetsPerCluster,maxZDistance,minZDistan
    % This method fits bounding boxes on each cluster with some basic
    % rules.
    % Cluster must have at least minDetsPerCluster points.
    % Its mean z must be between maxZDistance and minZDistance.
    % length, width and height are calculated using min and max from each
    % dimension.
    [labels,numClusters] = pcsegdist(ptCloud,minDistance);
    pointData = ptCloud.Location;
    bboxes = nan(7,numClusters,'like',pointData);
    isValidCluster = false(1,numClusters);
    for i = 1:numClusters
        thisPointData = pointData(labels == i,:);
        meanPoint = mean(thisPointData,1);
        if size(thisPointData,1) > minDetsPerCluster && ...
                meanPoint(3) < maxZDistance && meanPoint(3) > minZDistance
            cuboid = pcfitcuboid(pointCloud(thisPointData));
            yaw = cuboid.Orientation(3);
            L = cuboid.Dimensions(1);
            W = cuboid.Dimensions(2);
            H = cuboid.Dimensions(3);
            if abs(yaw) > 45
                possibles = yaw + [-90;90];
                [~,toChoose] = min(abs(possibles));
                yaw = possibles(toChoose);
                temp = L;
                L = W;
                W = temp;
            end
            bboxes(:,i) = [cuboid.Center yaw L W H]';
            isValidCluster(i) = L < 20 & W < 20;
        end
    end
```

```matlab
        bboxes = bboxes(:,isValidCluster);
    end

    function [ptCloudOut,obstacleIndices,groundIndices] = removeGroundPlane(ptCloudIn,maxGroundDist,
        % This method removes the ground plane from point cloud using
        % pcfitplane.
        [~,groundIndices,outliers] = pcfitplane(ptCloudIn,maxGroundDist,referenceVector,maxAngularDis
        ptCloudOut = select(ptCloudIn,outliers);
        obstacleIndices = currentIndices(outliers);
        groundIndices = currentIndices(groundIndices);
    end

    function [ptCloudOut,indices,croppedIndices] = cropPointCloud(ptCloudIn,xLim,yLim,zLim,egoVehicle
        % This method selects the point cloud within limits and removes the
        % ego vehicle point cloud using findNeighborsInRadius
        locations = ptCloudIn.Location;
        locations = reshape(locations,[],3);
        insideX = locations(:,1) < xLim(2) & locations(:,1) > xLim(1);
        insideY = locations(:,2) < yLim(2) & locations(:,2) > yLim(1);
        insideZ = locations(:,3) < zLim(2) & locations(:,3) > zLim(1);
        inside = insideX & insideY & insideZ;

        % Remove ego vehicle
        nearIndices = findNeighborsInRadius(ptCloudIn,[0 0 0],egoVehicleRadius);
        nonEgoIndices = true(ptCloudIn.Count,1);
        nonEgoIndices(nearIndices) = false;
        validIndices = inside & nonEgoIndices;
        indices = find(validIndices);
        croppedIndices = find(~validIndices);
        ptCloudOut = select(ptCloudIn,indices);
    end
```

mexLidarTracker

This function implements the point cloud preprocessing display and the tracking algorithm using a functional interface for code generation.

```matlab
function [detections,obstacleIndices,groundIndices,croppedIndices,...
    confirmedTracks, modelProbs] = mexLidarTracker(ptCloudLocations,time)

persistent detectorModel tracker detectableTracksInput currentNumTracks

if isempty(detectorModel) || isempty(tracker) || isempty(detectableTracksInput) || isempty(curren

    % Use the same starting seed as MATLAB to reproduce results in SIL
    % simulation.
    rng(2018);

    % A bounding box detector model.
    detectorModel = HelperBoundingBoxDetector(...
                    'XLimits',[-50 75],...              % min-max
                    'YLimits',[-5 5],...                % min-max
                    'ZLimits',[-2 5],...                % min-max
```

```
                      'SegmentationMinDistance',1.8,...   % minimum Euclidian distance
                      'MinDetectionsPerCluster',1,...     % minimum points per cluster
                      'MeasurementNoise',blkdiag(0.25*eye(3),25,eye(3)),...        % measurement noi
                      'GroundMaxDistance',0.3);           % maximum distance of ground points from

    assignmentGate = [75 1000]; % Assignment threshold;
    confThreshold = [7 10];     % Confirmation threshold for history logic
    delThreshold = [8 10];      % Deletion threshold for history logic
    Kc = 1e-9;                  % False-alarm rate per unit volume

    filterInitFcn = @helperInitIMMFilter;

    tracker = trackerJPDA('FilterInitializationFcn',filterInitFcn,...
                      'TrackLogic','History',...
                      'AssignmentThreshold',assignmentGate,...
                      'ClutterDensity',Kc,...
                      'ConfirmationThreshold',confThreshold,...
                      'DeletionThreshold',delThreshold,...
                      'HasDetectableTrackIDsInput',true,...
                      'InitializationThreshold',0,...
                      'MaxNumTracks',30,...
                      'HitMissThreshold',0.1);

    detectableTracksInput = zeros(tracker.MaxNumTracks,2);

    currentNumTracks = 0;
end

ptCloud = pointCloud(ptCloudLocations);

% Detector model
[detections,obstacleIndices,groundIndices,croppedIndices] = detectorModel(ptCloud,time);

% Call tracker
[confirmedTracks,~,allTracks] = tracker(detections,time,detectableTracksInput(1:currentNumTracks
% Update the detectability input
currentNumTracks = numel(allTracks);
detectableTracksInput(1:currentNumTracks,:) = helperCalcDetectability(allTracks,[1 3 6]);

% Get model probabilities
modelProbs = zeros(2,numel(confirmedTracks));
if isLocked(tracker)
    for k = 1:numel(confirmedTracks)
        c1 = getTrackFilterProperties(tracker,confirmedTracks(k).TrackID,'ModelProbabilities');
        probs = c1{1};
        modelProbs(1,k) = probs(1);
        modelProbs(2,k) = probs(2);
    end
end

end
```

helperCalcDetectability

The function calculate the probability of detection for each track. This function is used to generate the "DetectableTracksIDs" input for the `trackerJPDA`.

```
function detectableTracksInput = helperCalcDetectability(tracks,posIndices)
% This is a helper function to calculate the detection probability of
% tracks for the lidar tracking example. It may be removed in a future
% release.

% Copyright 2019 The MathWorks, Inc.

% The bounding box detector has low probability of segmenting point clouds
% into bounding boxes are distances greater than 40 meters. This function
% models this effect using a state-dependent probability of detection for
% each tracker. After a maximum range, the Pd is set to a high value to
% enable deletion of track at a faster rate.
if isempty(tracks)
    detectableTracksInput = zeros(0,2);
    return;
end
rMax = 75;
rAmbig = 40;
stateSize = numel(tracks(1).State);
posSelector = zeros(3,stateSize);
posSelector(1,posIndices(1)) = 1;
posSelector(2,posIndices(2)) = 1;
posSelector(3,posIndices(3)) = 1;
pos = getTrackPositions(tracks,posSelector);
if coder.target('MATLAB')
    trackIDs = [tracks.TrackID];
else
    trackIDs = zeros(1,numel(tracks),'uint32');
    for i = 1:numel(tracks)
        trackIDs(i) = tracks(i).TrackID;
    end
end
[~,~,r] = cart2sph(pos(:,1),pos(:,2),pos(:,3));
probDetection = 0.9*ones(numel(tracks),1);
probDetection(r > rAmbig) = 0.4;
probDetection(r > rMax) = 0.99;
detectableTracksInput = [double(trackIDs(:)) probDetection(:)];
end
```

loadLidarAndImageData

Stitches Lidar and Camera data for processing using initial and final time specified.

```
function [lidarData,imageData] = loadLidarAndImageData(datasetFolder,initTime,finalTime)
initFrame = max(1,floor(initTime*10));
lastFrame = min(350,ceil(finalTime*10));
load (fullfile(datasetFolder,'imageData_35seconds.mat'),'allImageData');
imageData = allImageData(initFrame:lastFrame);

numFrames = lastFrame - initFrame + 1;
lidarData = cell(numFrames,1);

% Each file contains 70 frames.
initFileIndex = floor(initFrame/70) + 1;
lastFileIndex = ceil(lastFrame/70);
```

```
frameIndices = [1:70:numFrames numFrames + 1];

counter = 1;
for i = initFileIndex:lastFileIndex
    startFrame = frameIndices(counter);
    endFrame = frameIndices(counter + 1) - 1;
    load(fullfile(datasetFolder,['lidarData_',num2str(i)]),'currentLidarData');
    lidarData(startFrame:endFrame) = currentLidarData(1:(endFrame + 1 - startFrame));
    counter = counter + 1;
end
end
```

References

[1] Arya Senna Abdul Rachman, Arya. "3D-LIDAR Multi Object Tracking for Autonomous Driving: Multi-target Detection and Tracking under Urban Road Uncertainties." (2017).

Extended Object Tracking With Radar For Marine Surveillance

This example shows how to generate a marine scenario, simulate radar detections from a marine surveillance radar, and configure a multi-target Probability Hypothesis Density (PHD) tracker to estimate the location and size of the simulated ships using the radar detections.

Marine Surveillance Scenario

Simulate a marine surveillance radar mounted at the top of a tower overlooking ships in a harbor. Simulation of the tower's location and the motion of the ships in the scenario is managed by `trackingScenario`.

```
% Create tracking scenario.
scenario = trackingScenario('StopTime',30);

% Define unit conversions.
nmi2m = 1852;            % nautical miles to meters
hr2s = 3600;            % hours to seconds
kts2mps = nmi2m/hr2s;   % knots to meters per second
```

Marine Surveillance Radar

Add a marine surveillance radar to the tower. The radar is mounted 20 meters above sea level (ASL). The radar stares into the harbor, surveying a 30 degree azimuth sector. Common specifications for a marine surveillance radar are listed:

- Sensitivity: 0 dBsm @ 5 km
- Field of View: 30 deg azimuth, 10 deg elevation
- Azimuth Resolution: 2 deg
- Range Resolution: 5 m

Model the marine radar with the above specifications using the `monostaticRadarSensor`.

```
% Create surveillance radar.
sensor = monostaticRadarSensor(1,'No scanning', ...
    'MountingLocation',[0 0 -20], ...  % 20 meters (ASL)
    'MountingAngles',[45 0 0], ...     % [yaw pitch roll] deg
    'FieldOfView',[30 10], ...         % [az el] deg
    'ReferenceRange',5e3, ...          % m
    'AzimuthResolution',2, ...         % deg
    'RangeResolution',5, ...           % m
    'HasINS',true,'DetectionCoordinates','Sensor spherical');
```

Add the tower to the scenario as a stationary platform with the radar mounted on top of it.

```
platform(scenario,'Sensors',sensor);
tower = scenario.Platforms{1}

tower =

  Platform with properties:

        PlatformID: 1
          ClassID: 0
          Position: [0 0 0]
```

```
       Orientation: [0 0 0]
        Dimensions: [1x1 struct]
              Mesh: [1x1 extendedObjectMesh]
        Trajectory: [1x1 kinematicTrajectory]
     PoseEstimator: [1x1 insSensor]
          Emitters: {}
           Sensors: {[1x1 monostaticRadarSensor]}
        Signatures: {[1x1 rcsSignature]  [1x1 irSignature]  [1x1 tsSignature]}
```

Add three ships in the harbor within the radar's surveillance sector. The two smaller ships are turning at 20 and 30 knots, the large ship is traveling at a constant heading at 10 knots.

```matlab
% Define the dimensions for the two small ships.
dim = struct( ...
    'Length',80, ... % m
    'Width',15, ...  % m
    'Height',5, ...  % m
    'OriginOffset', [0 0 5/2]); % [x y z] m

% Model the radar cross section (RCS) of the small ships as 30 dBsm.
rcs = rcsSignature('Pattern',30);

% Create a turning trajectory.
speed = 20;     % knots
initYaw = 130;  % deg
initPos = [1050 790 0];
radius = 200;   % m

initOrient = quaternion([initYaw 0 0], 'eulerd', 'ZYX', 'frame');
initVel = speed*kts2mps*rotatepoint(initOrient,[1 0 0])';
accBody = [0 (speed*kts2mps)^2/radius 0];
angVelBody = [0 0 speed*kts2mps/radius];
traj = kinematicTrajectory('Position',initPos,'Velocity',initVel,'Orientation',initOrient, ...
    'AccelerationSource','Property','Acceleration',accBody, ...
    'AngularVelocitySource','Property','AngularVelocity', angVelBody);

% Add the first small ship, traveling at 20 knots to the scenario. This is
% the closest ship to the radar tower.
platform(scenario,'Dimensions',dim,'Signatures',rcs,'Trajectory',traj);

% Create the other small ship, traveling at 30 knots. This is the ship
% which is farthest from the radar tower.
speed = 30;     % knots
initYaw = 120;  % deg
initPos = [1410 1180 0];
radius = 400;   % m

initOrient = quaternion([initYaw 0 0],'eulerd','ZYX','frame');
initVel = speed*kts2mps*rotatepoint(initOrient,[1 0 0])';
accBody = [0 (speed*kts2mps)^2/radius 0];
angVelBody = [0 0 speed*kts2mps/radius];
traj = kinematicTrajectory('Position',initPos,'Velocity',initVel,'Orientation',initOrient, ...
    'AccelerationSource','Property','Acceleration',accBody, ...
    'AngularVelocitySource','Property','AngularVelocity', angVelBody);

platform(scenario,'Dimensions',dim,'Signatures',rcs,'Trajectory',traj);
```

```matlab
% Define the dimensions for the large ship.
dim = struct( ...
    'Length',400, ... % m
    'Width',60, ...   % m
    'Height',15, ...  % m
    'OriginOffset', [0 0 15/2]); % [x y z] m

% Model the radar cross section (RCS) of the large ship as 75 dBsm.
rcs = rcsSignature('Pattern',75);

% Create the large ship's trajectory, traveling at a constant heading at 10 knots.
speed = 10;     % knots
initYaw = -135; % deg
initPos = [1150 1100 0];

initOrient = quaternion([initYaw 0 0],'eulerd','ZYX','frame');
initVel = speed*kts2mps*rotatepoint(initOrient,[1 0 0])';
traj = kinematicTrajectory('Position',initPos,'Velocity',initVel,'Orientation',initOrient, ...
    'AccelerationSource','Property','AngularVelocitySource','Property');

% Add the large ship to the scenario.
platform(scenario,'Dimensions',dim,'Signatures',rcs,'Trajectory',traj);

% Create a display to show the true, measured, and tracked positions of the ships.
theaterDisplay = helperMarineSurveillanceDisplay(scenario, ...
    'IsSea',true,'DistanceUnits','m', ...
    'XLim',450*[-1 1]+1e3,'YLim',450*[-1 1]+1e3,'ZLim',[-1000 10], ...
    'Movie','MarineSurveillanceExample.gif');
slctTrkPos = zeros(3,7); slctTrkPos(1,1) = 1; slctTrkPos(2,3) = 1; slctTrkPos(3,6) = 1;
slctTrkVel = circshift(slctTrkPos,[0 1]);
theaterDisplay.TrackPositionSelector = slctTrkPos;
theaterDisplay.TrackVelocitySelector = slctTrkVel;
theaterDisplay();
snapnow(theaterDisplay);
```
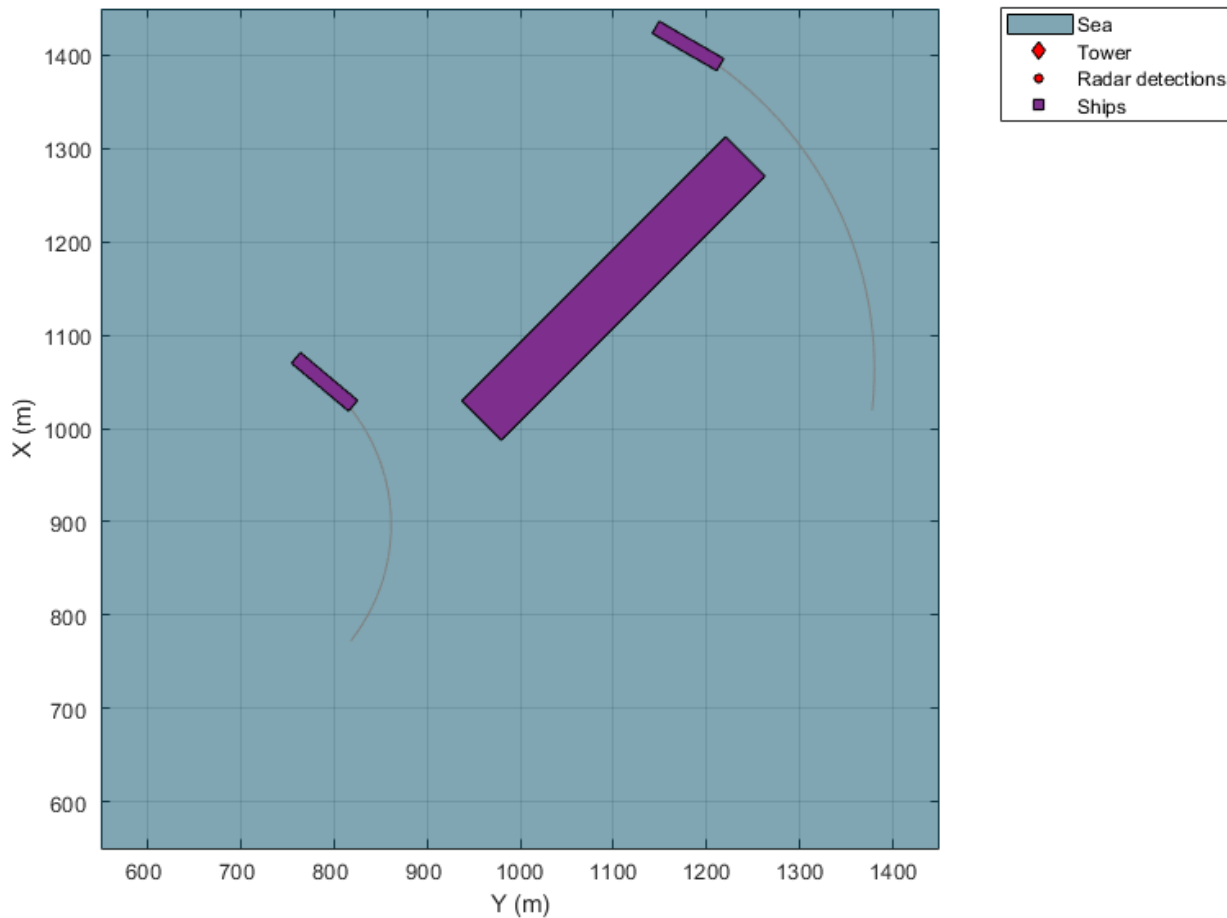
Multi-Target GGIW-PHD Tracker

Create a `trackerPHD` to form tracks from the radar detections generated from the three ships in the harbor. The PHD tracker enables the estimation of the size of the ships by allowing multiple detections to be associated to a single object. This is important in situations such as marine surveillance where the size of the objects detected by the sensor is greater than the sensor's resolution, resulting in multiple detections generated along the surfaces of the ships.

The tracker uses the `filterInitFcn` supporting function to initialize a constant turn-rate Gamma Gaussian Inverse Wishart (GGIW) PHD filter. `filterInitFcn` adds birth components to the PHD-intensity at every time step. These birth components are added uniformly inside the field of view of the sensor. Their sizes and expected number of detections are specified using prior information about the types of ships expected in the harbor.

The tracker uses the gamma distribution of the GGIW-PHD components to estimate how many detections should be generated from an object. The tracker also calculates the detectability of each component in the density using the sensor's limits. Use `trackingSensorConfiguration` to model the sensor's configuration for `trackerPHD`.

```
% Define radar's measurement limits and resolution.
azLimits = sensor.FieldOfView(1)/2*[-1 1]; % deg
rangeLimits = [0 15e3];                     % m
sensorLimits = [azLimits;rangeLimits];
sensorResolution = [sensor.AzimuthResolution;sensor.RangeResolution];

% Define the sensor's mounting location and orientation on the tower.
params(1) = struct('Frame','Spherical', ...
    'OriginPosition',sensor.MountingLocation(:), ...
    'OriginVelocity',[0;0;0], ...
    'Orientation',rotmat(quaternion(sensor.MountingAngles,'eulerd','zyx','frame'),'frame'),...
    'IsParentToChild', true, ...
    'HasRange',true,'HasElevation',sensor.HasElevation,'HasVelocity',false);

% Define the tower's location, velocity, and orientation in the scenario.
params(2) = struct('Frame','Rectangular', ...
    'OriginPosition',tower.Trajectory.Position(:), ...
    'OriginVelocity',tower.Trajectory.Velocity(:), ...
    'Orientation',rotmat(tower.Trajectory.Orientation,'frame'), ...
    'IsParentToChild', true, ...
    'HasRange',true,'HasElevation',false,'HasVelocity',false);

% Create a trackingSensorConfiguration to model the detectability of the
% tracks by the sensor.
sensorConfig = trackingSensorConfiguration('SensorIndex',sensor.SensorIndex, ...
    'SensorLimits',sensorLimits,...
    'SensorResolution',sensorResolution,...
    'SensorTransformParameters',params);

% Fields on the sensor configuration used to update trackingSensorConfiguration.
configFlds = {'SensorIndex','IsValidTime'};

% Noise covariance corresponding to a resolution cell of the radar.
resolutionNoise = diag((sensorResolution/2).^2);

sensorConfig.FilterInitializationFcn = @(varargin)filterInitFcn(varargin{:},params);
sensorConfig.SensorTransformFcn = @ctmeas;
sensorConfig.ClutterDensity = sensor.FalseAlarmRate/(sensor.AzimuthResolution*sensor.RangeResolut

% Create a PHD tracker using the trackingSensorConfiguration.
tracker = trackerPHD('SensorConfigurations',sensorConfig, ...
    'HasSensorConfigurationsInput',true, ...
    'PartitioningFcn',@(x)partitionDetections(x,1.5,6), ...
    'DeletionThreshold',1e-6,...
    'BirthRate',1e-5);
```

Simulate and Track Ships

The following loop advances the positions of the ships until the end of the scenario. For each step forward in the scenario, the tracker is updated with the detections from the ships in the radar's field of view.

```
% Initialize scenario and tracker.
restart(scenario);
reset(tracker);

% Set simulation to advance at the update rate of the radar.
scenario.UpdateRate = sensor.UpdateRate;
```

```matlab
% Set random seed for repeatable results.
rng(2019);

% Run simulation.
snapTimes = [2 7 scenario.StopTime]; % seconds
while advance(scenario)
    % Get current simulation time.
    time = scenario.SimulationTime;

    % Generate detections from the tower's radar.
    [dets,~,config] = detect(tower,time);

    % Update measurement noise of detections to match radar's resolution.
    dets = updateMeasurementNoise(dets,resolutionNoise);

    % Update tracker.
    trackSensorConfig = computeTrackingSensorConfig(config,configFlds);
    tracks = tracker(dets,trackSensorConfig,time);

    % Update display with current beam position, detections, and track positions.
    theaterDisplay(dets,config,tracks);

    % Take snapshot.
    snapFigure(theaterDisplay,any(time==snapTimes));
end
writeMovie(theaterDisplay);
```
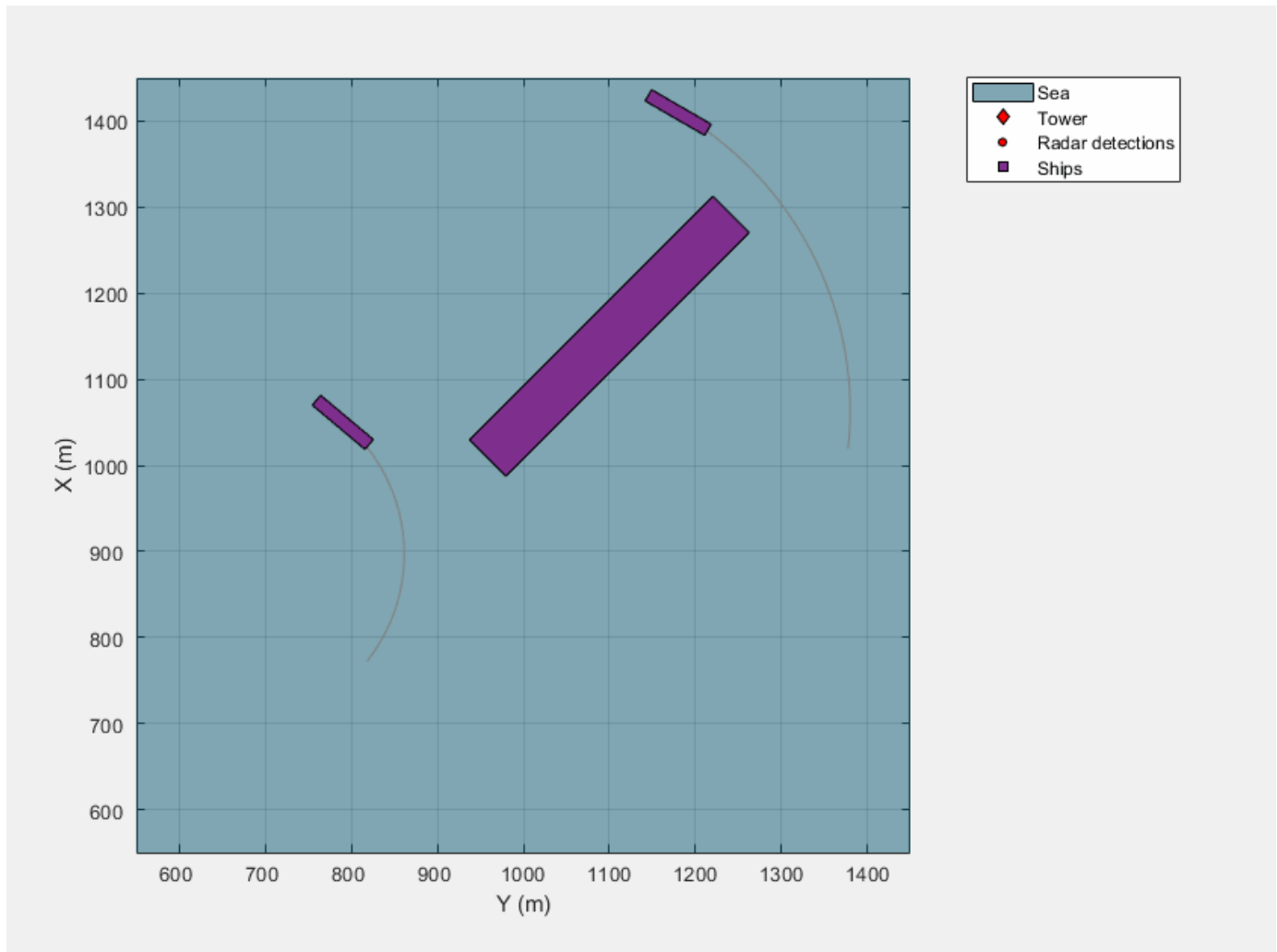
The following figure shows the radar detections, shown as red dots, and the estimated track locations, shown as yellow squares annotated with the track ID, and the estimated tracked object's extent, shown as a yellow ellipse. The radar tower is located at the origin, (0,0), which is not shown in the figure. The radar's field of view is indicated by the two red lines crossing the top and bottom of the figure. All of the ships lie within the radar's field of view, and because the size of the ships is much larger than the radar's range and azimuth resolution, multiple detections are made along the faces of the ships visible to the radar.

```
showSnapshot(theaterDisplay,1)
```
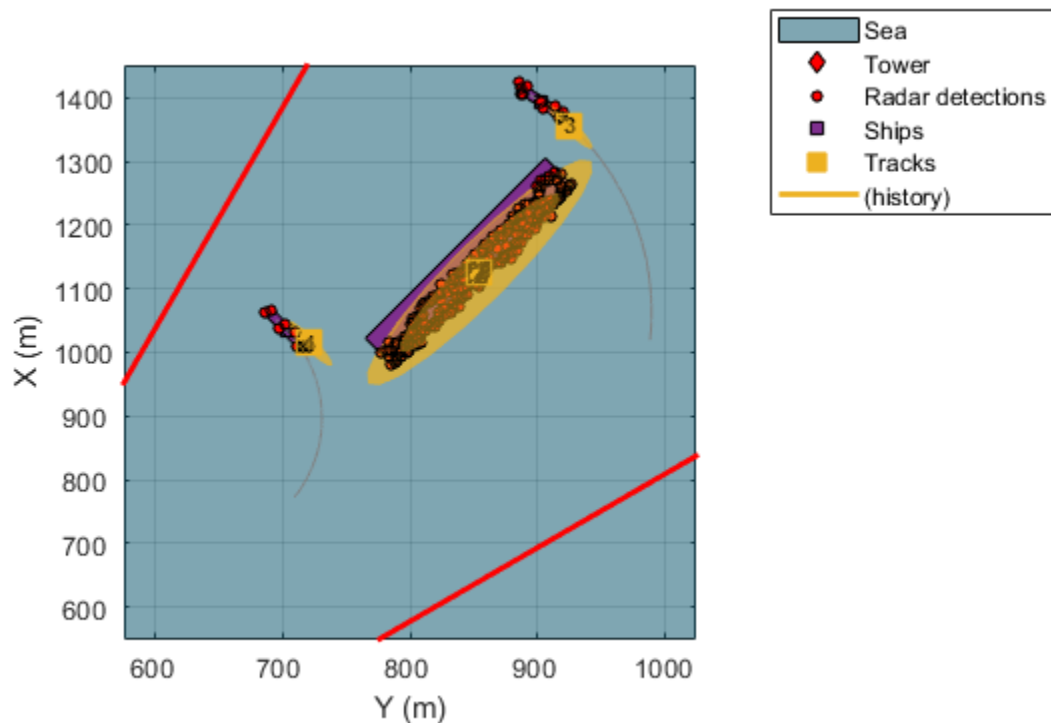
Because the ships are modeled as extended objects and not point targets, detections of a ship can be occluded by the presence of another ship between the ship and the radar. This is shown in the following figure. In this case, the smaller ship at the top of the figure is not detected by the radar. The radar's line of sight is occluded by both the other small ship at the bottom of the figure and the large ship in the center. The tracker maintains an estimate of the occluded ship and associates detections in the following steps to the track without ever dropping the track.

```
showSnapshot(theaterDisplay,2)
axis([1250 1450 1150 1350]); view([-90 90]);
```
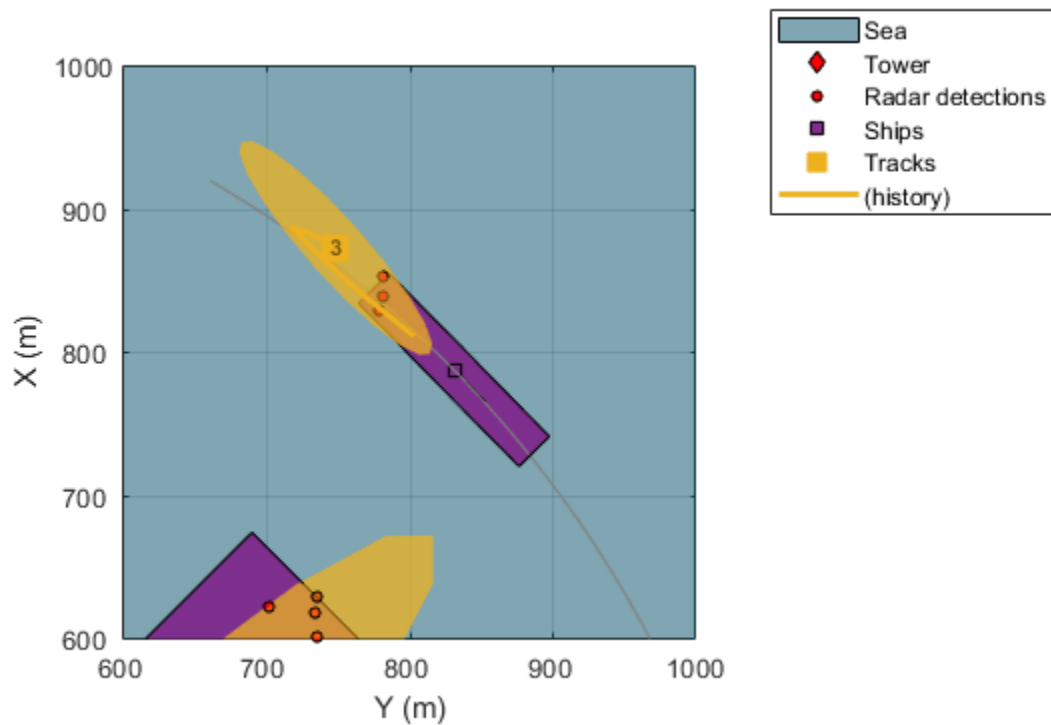
The following figure shows the PHD estimate of the smaller ship located closest to the radar in the scenario. You can verify the PHD estimate of the ship's location lies near the center of the ship and the estimated size is reasonably close to the actual size of the ship, indicated by the overlap of the track's ellipse with the ship.

```
showSnapshot(theaterDisplay,3)
axis([650 850 700 900]); view([-90 90]);
```
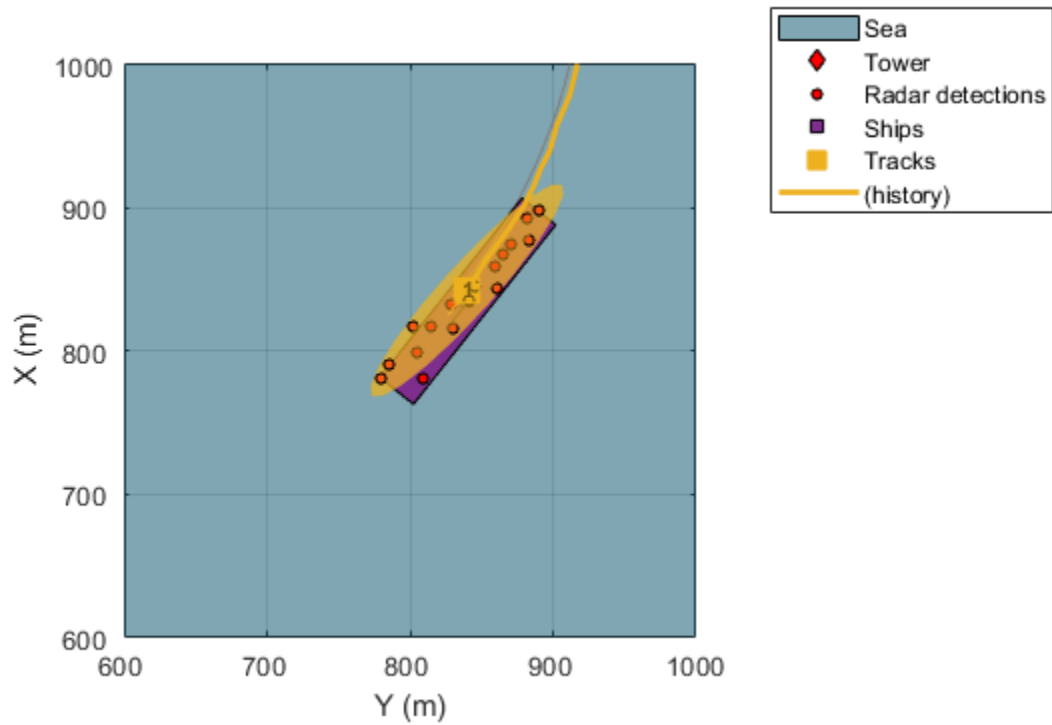
The next figure also shows that the PHD tracker estimated both the location, size, and heading of the other small ship in the scenario. This is the ship that was previously occluded by the other two ships. Despite the occlusion, the estimated location, size, and orientation closely matches the ship.

```
showSnapshot(theaterDisplay,3)
axis([900 1100 1250 1450]); view([-90 90]);
```
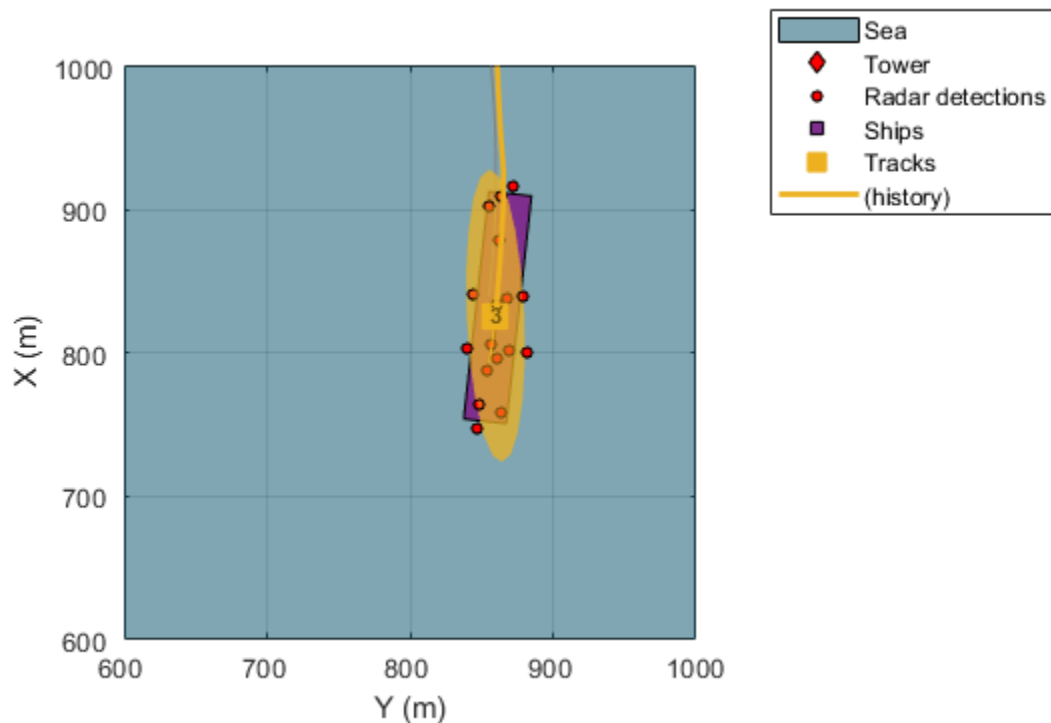
The track states of the 3 ships report the estimated size of each ship using a 3D positional covariance matrix. Take the eigen decomposition of the covariance matrices to compute the estimated length, width, and height for each of the ships.

```
numTrks = numel(tracks);
TrackID = [tracks.TrackID]';
Length = zeros(numTrks,1);
Width = zeros(numTrks,1);
Height = zeros(numTrks,1);
for iTrk = 1:numTrks
    ext = tracks(iTrk).Extent;
    [Q,D] = eig(ext);
    d = 2*sqrt(diag(D));
    iDims = 1:3;

    up = [0 0 -1];
    [~,iUp] = max(abs(up*Q));
    Height(iTrk) = d(iDims(iUp));
    iDims(iUp) = [];

    Length(iTrk) = max(d(iDims));
    Width(iTrk) = min(d(iDims));
end

% Display a table of the estimated dimensions of the ships.
dims = table(TrackID,Length,Width,Height)
```

```
dims =

  3x4 table

    TrackID     Length     Width      Height
    _____     _____     _____     _____

       1         99.38     17.597     16.464
       2        474.46     52.653     8.6582
       3        102.33     20.182     17.635
```

Recall that the true dimensions of the ships are given by:

Large Ship

- Length: 400 m
- Width: 60 m
- Height: 15 m

Small Ship

- Length: 80 m
- Width: 15 m
- Height: 5 m

The tracker is able to differentiate between the size of the large and smaller ships by estimating the shape of each ship as an ellipse. In the simulation, the true shape of each ship is modeled using a cuboid. This mismatch between the shape assumption made by the tracker and the true shapes of the modeled ships results in overestimates the length and width of the ships. The radar is a 2D sensor, only measuring range and azimuth, so the height of each ship is not observable. This results in inaccurate height estimates reported by the tracker.

Summary

This example shows how to generate a marine scenario, simulate radar detections from a marine surveillance radar, and configure a multi-target PHD tracker to track the simulated ships using the radar detections. In this example, you learned how to model extended objects in a scenario, generating multiple detections from these objects. You also learned how to use a multi-target PHD tracker to process the information provided by the multiple detections to estimate not only the location but also the size of the tracked objects.

Supporting Functions

computeTrackingSensorConfig

Returns a structure containing configuration information returned by the radar. This structure is used to update the trackingSensorConfiguration model used within the tracker.

```
function config = computeTrackingSensorConfig(configIn,flds)
config = struct();
for iFld = 1:numel(flds)
    thisFld = flds{iFld};
    config.(thisFld) = configIn.(thisFld);
```

```
    end
end
```

updateMeasurementNoise

Sets the measurement noise of the detections based on the specified noise covariance.

```
function dets = updateMeasurementNoise(dets,noise)
    for iDet = 1:numel(dets)
        dets{iDet}.MeasurementNoise(:) = noise(:);
    end
end
```

filterInitFcn

Modifies the filter returned by `initctggiwphd` to match the velocities and expected number of detections for the ships being tracked.

```
function phd = filterInitFcn(measParam,varargin)
% This function uses the predictive birth density only to simulate birth in
% the scenario.

if nargin == 1 % Set predictive birth uniformly inside the FOV.
    phdDefault = initctggiwphd;
    % 1. Create uniformly distributed states using Azimuth and Range.
    az = 0; % All components at azimuth.
    ranges = linspace(1000,10000,5); % 5 components for range
    [Az,R] = meshgrid(az,ranges);

    % create a PHD filter to allocate memory.
    phd = ggiwphd(zeros(7,numel(Az)),repmat(eye(7),[1 1 numel(Az)]),...
        'ScaleMatrices',repmat(eye(3),[1 1 numel(Az)]),...
        'StateTransitionFcn',@constturn,'StateTransitionJacobianFcn',@constturnjac,...
        'MeasurementFcn',@ctmeas,'MeasurementJacobianFcn',@ctmeasjac,...
        'PositionIndex',[1 3 6],'ExtentRotationFcn',phdDefault.ExtentRotationFcn,...
        'HasAdditiveProcessNoise',false,'ProcessNoise',2*eye(4),...
        'TemporalDecay',1e3,'GammaForgettingFactors',1.1*ones(1,numel(Az)),...
        'MaxNumComponents',10000);

    for i = 1:numel(Az)
        [sensorX,sensorY,sensorZ] = sph2cart(deg2rad(Az(i)),0,R(i));
        globalPos = measParam(1).Orientation'*[sensorX;sensorY;sensorZ] + measParam(1).OriginPos:
        phd.States([1 3 6],i) = globalPos;
        phd.StateCovariances([1 3 6],[1 3 6],i) = diag([1e5 1e5 1000]); % Cover gaps between comp
    end

    % 2. You have described the "kinematic" states of each of the ships
    % inside the field of view. Next, add information about their sizes and
    % expected number of detections.
    %
    % It is expected that there are 2 types of ships in the sea, small and
    % large. You create components for each size.
    phdSmall = phd;

    % Clone the PHD filter for large ships.
    phdLarge = clone(phd);

    % Set initial number of components.
```

```
numComps = phdSmall.NumComponents;

% For small ships, the expected size is about 100 meters in length and
% 20 meters in width. As the orientation is unknown, we will create 4
% orientations for each size. First, you must add components to the
% density at same states. This can be done by simply appending it
% Setup values for small boats
append(phdSmall,phdSmall);
append(phdSmall,phdSmall);

% Degrees of freedom for defining shape. A large number represents
% higher certainty in dimensions.
dof = 1000;

% Covariance in vx, vy and omega.
smallStateCov = diag([300 300 100]);

% Scale matrix for small boats
smallShape = (dof - 4)*diag([100/2 20/2 10].^2); % l, w and h

% Create 4 orientations at 45 degrees from each other.
for i = 1:4
    thisIndex = (i-1)*numComps + (1:numComps);
    R = rotmat(quaternion([45*(i-1) 0 0],'eulerd','ZYX','frame'),'frame');
    phdSmall.ScaleMatrices(:,:,thisIndex) = repmat(R*smallShape*R',[1 1 numComps]);
    phdSmall.StateCovariances([2 4 5],[2 4 5],thisIndex) = repmat(R*smallStateCov*R',[1 1 num
    phdSmall.StateCovariances([6 7],[6 7],thisIndex) = repmat(diag([100 100]),[1 1 numComps]
end

% Small ships generate approximately 10-20 detections.
expNumDets = 15;
uncertainty = 5^2;
phdSmall.Rates(:) = expNumDets/uncertainty;
phdSmall.Shapes(:) = expNumDets^2/uncertainty;
phdSmall.DegreesOfFreedom(:) = dof;

% Follow similar process for large ships.
append(phdLarge,phdLarge);
append(phdLarge,phdLarge);
largeStateCov = diag([100 5 10]);
largeShape = (dof - 4)*diag([500/2 100/2 10].^2);
for i = 1:4
    thisIndex = (i-1)*numComps + (1:numComps);
    R = rotmat(quaternion([45*(i-1) 0 0],'eulerd','ZYX','frame'),'frame');
    phdLarge.ScaleMatrices(:,:,thisIndex) = repmat(R*largeShape*R',[1 1 numComps]);
    phdLarge.StateCovariances([2 4 5],[2 4 5],thisIndex) = repmat(R*largeStateCov*R',[1 1 num
    phdLarge.StateCovariances([6 7],[6 7],thisIndex) = repmat(diag([100 100]),[1 1 numComps]
end

% Generate approximately 100-200 detections.
expNumDets = 150;
uncertainty = 50^2;
phdLarge.Rates(:) = expNumDets/uncertainty;
phdLarge.Shapes(:) = expNumDets^2/uncertainty;
phdLarge.DegreesOfFreedom(:) = dof;

% Add large ships to small ships to create total density. This density
% is added to the total density every step.
```

```
    phd = phdSmall;
    append(phd,phdLarge);
end

% When called with detection input i.e. the adaptive birth density, do not
% add any new components.
if nargin > 1
    % This creates 0 components in the density.
    phd = initctggiwphd;
end

end
```

Track Vehicles Using Lidar Data in Simulink

This example shows you how to track vehicles using measurements from a lidar sensor mounted on top of an ego vehicle. Due to high resolution capabilities of the lidar sensor, each scan from the sensor contains a large number of points, commonly known as a point cloud. The example illustrates the workflow in Simulink for processing the point cloud and tracking the objects. The lidar data used in this example is recorded from a highway driving scenario. You use the recorded data to track vehicles with a joint probabilistic data association (JPDA) tracker and an interacting multiple model (IMM) approach. The example closely follows the "Track Vehicles Using Lidar: From Point Cloud to Track List" on page 5-369 MATLAB® example.

Overview of the Model

```
load_system('TrackVehiclesSimulinkExample');
set_param('TrackVehiclesSimulinkExample','SimulationCommand','update');
open_system('TrackVehiclesSimulinkExample');
```
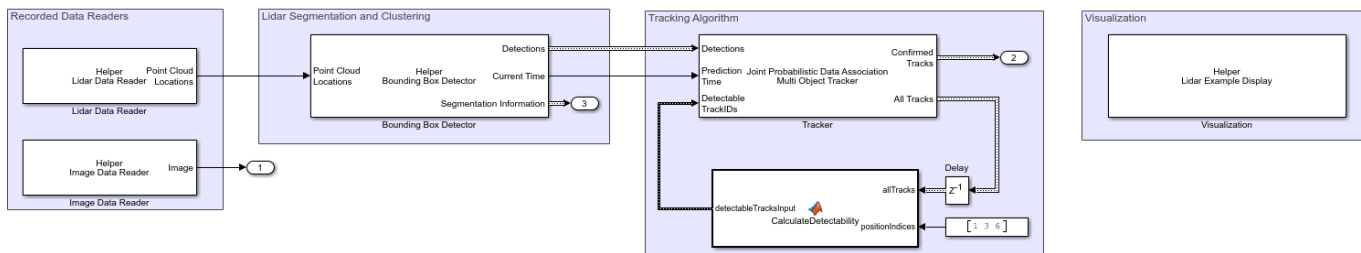

Lidar and Image Data Reader

The Lidar Data Reader and Image Data Reader blocks are implemented using a MATLAB System (Simulink) block. The code for the blocks is defined by helper classes, `HelperLidarDataReader` and `HelperImageDataReader` respectively. The image and lidar data readers read the recorded data from the MAT files and output the reference image and the locations of points in the point cloud respectively.

Bounding Box Detector

As described earlier, the raw data from sensor contains a large number of points. This raw data must be preprocessed to extract objects of interest, such as cars, cyclists, and pedestrian. The preprocessing is done using the Bounding Box Detector block. The Bounding Box Detector is also implemented as a MATLAB System™ block defined by a helper class, `HelperBoundingBoxDetectorBlock`. It accepts the point cloud locations as an input and outputs bounding box detections corresponding to obstacles. The diagram shows the processes involved in the bounding box detector model and the Computer Vision Toolbox™ functions used to implement each process. It also shows the parameters of the block that control each process.

HelperBoundingBoxDetector

Raw point cloud (pointCloud) → **Region of interest** → Inlier point cloud → **Ground removal (pcfitplane)** → Obstacle cloud → **Clustering (pcsegdist)** → Clusters → **Bounding Boxes (pcfitcuboid)** → objectDetection

Property names →

Region of interest	Ground removal	Clustering	Bounding Boxes
XLimits	GroundMaxDistance	SegmentationMinDistance	MinPointsPerCluster
YLimits	GroundReferenceVector		MaxZDistanceCluster
ZLimits	GroundMaxAngularDistance		MinZDistanceCluster
EgoVehicleRadius			

The block outputs the detections and segmentation information as a `Simulink.Bus` (Simulink) object named `detectionBus` and `segmentationBus`. These buses are created in the base workspace using helper function `helperCreateDetectorBus` specified in the `PreLoadFcn` callback. See "Model Callbacks" (Simulink) for more information about callback functions.

Tracking algorithm

The tracking algorithm is implemented using the joint probabilistic data association (JPDA) tracker, which uses an interacting multiple model (IMM) approach to track targets. The IMM filter is implemented by the `helperInitIMMFilter`, which is specified as the "Filter initialization function" parameter of the block. In this example, the IMM filter is configured to use two models, a constant velocity cuboid model and a constant turn-rate cuboid model. The models define the dimensions of the cuboid as constants during state-transition and their estimates evolve in time during correction stages of the filter. The animation below shows the effect of mixing the constant velocity and constant turn-rate models with different probabilities during prediction stages of the filter.
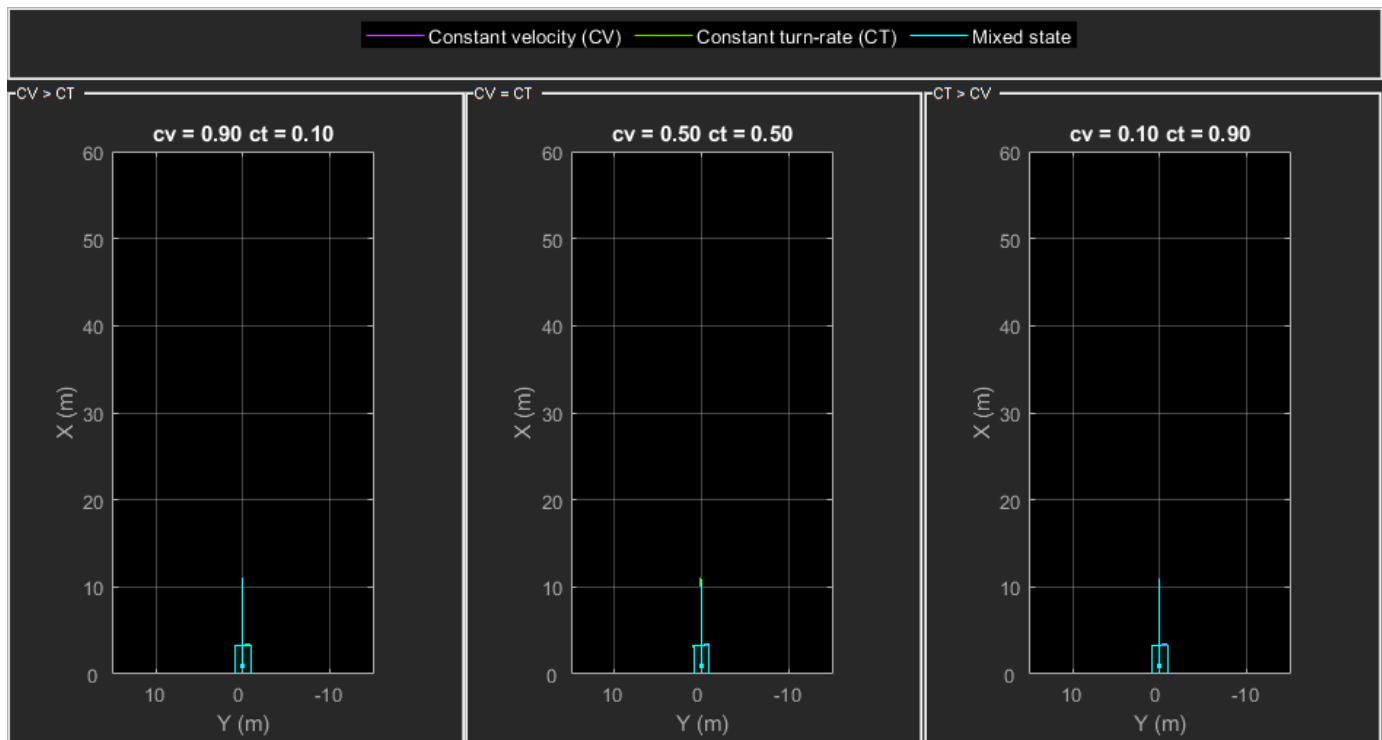
The IMM filter automatically computes the probability of each model when the filter is corrected with detections. The animation below shows the estimated trajectory and the probability of models during a lane change event.
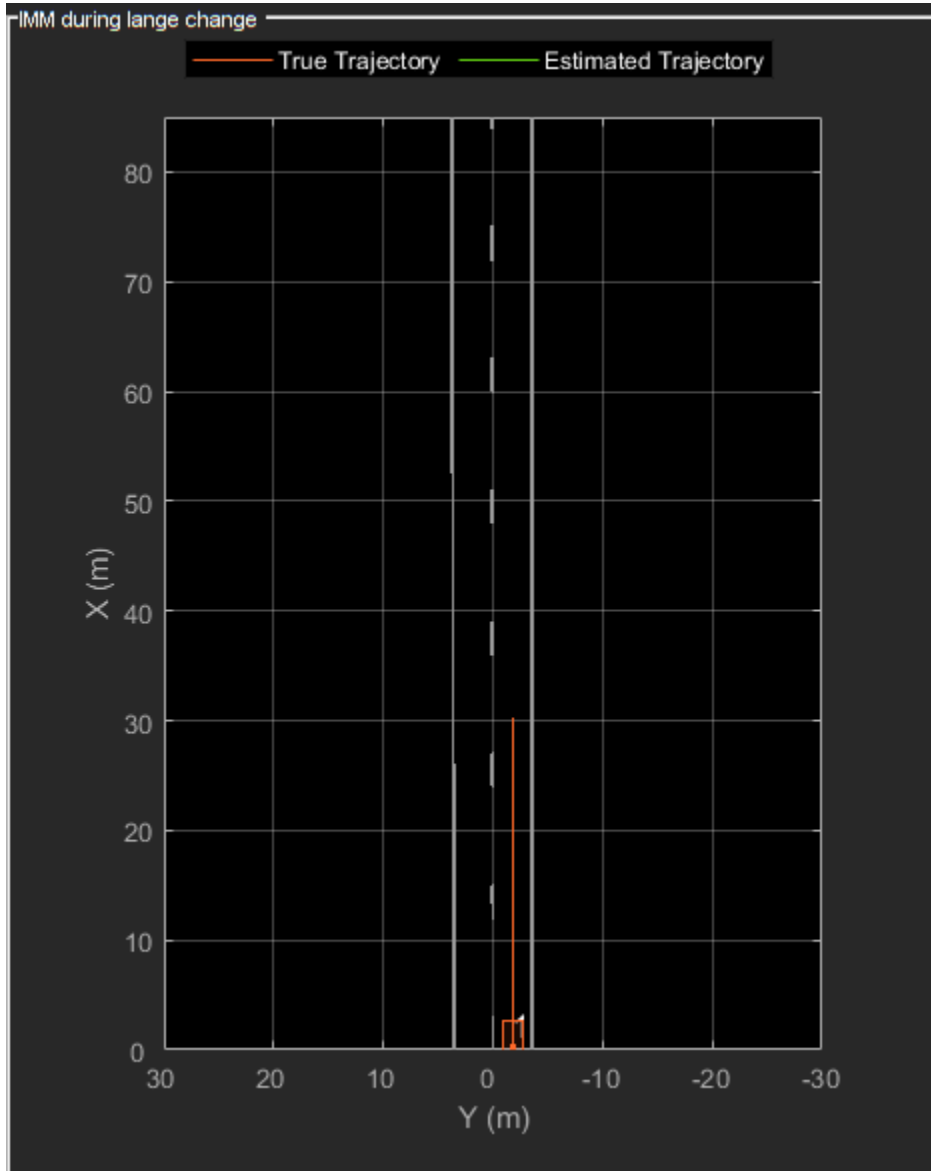
For a detailed description of the state transition and measurement models, refer to the "Target State and Sensor Measurement Model" section of the MATLAB example.

The tracker block selects the check box "Enable all tracks output" and "Enable detectable track IDs input" to output all tracks from the tracker and calculate their detection probability as a function of their state.

Calculate Detectability

The Calculate Detectability block is implemented using a MATLAB Function (Simulink) block. The block calculates the Detectable TrackIDs input for the tracker and outputs it as an array with 2

columns. The first column represents the TrackIDs of the tracks and the second column specifies their probability of detection by the sensor and bounding box detector.

Visualization

The Visualization block is also implemented using the MATLAB System block and is defined using `HelperLidarExampleDisplayBlock`. The block uses `RunTimeObject` parameter of the blocks to display their outputs. See "Access Block Data During Simulation" (Simulink) for further information on how to access block outputs during simulation.

Detections and Tracks Bus Objects

As described earlier, the inputs and outputs of different blocks are bus objects. You can visualize the structure of each bus using the Bus Editor (Simulink). The following images show the structure of the bus for detections and tracks.

Detections

The `detectionBus` outputs a nested bus object with 2 elements, `NumDetections` and `Detections`.
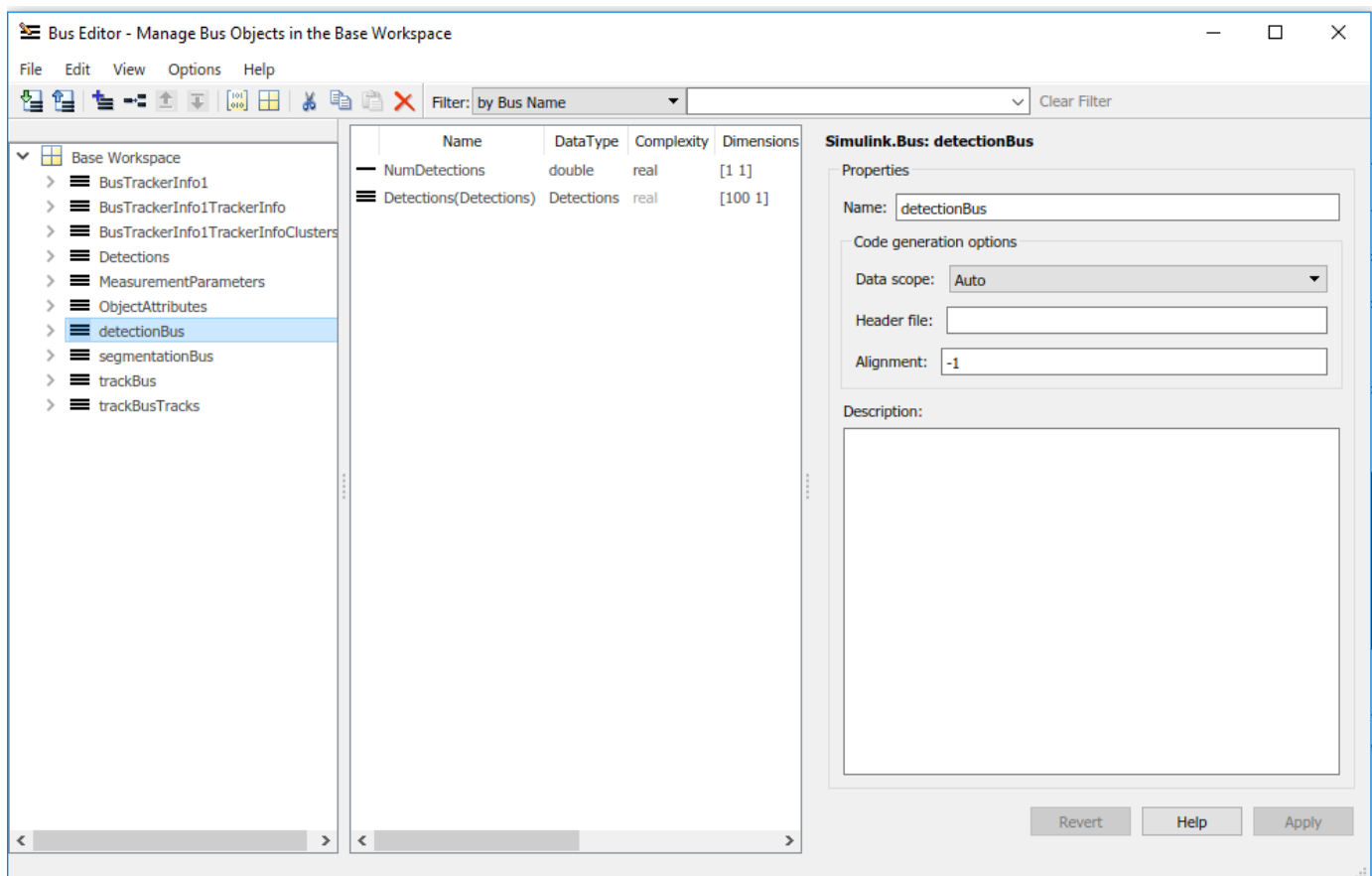
The first element, `NumDetections`, represents the number of detections. The second element `Detections` is a bus object of a fixed size representing all detections. The first `NumDetections` elements of the bus object represent the current set of detections. Notice that the structure of the bus is similar to the `objectDetection` class.
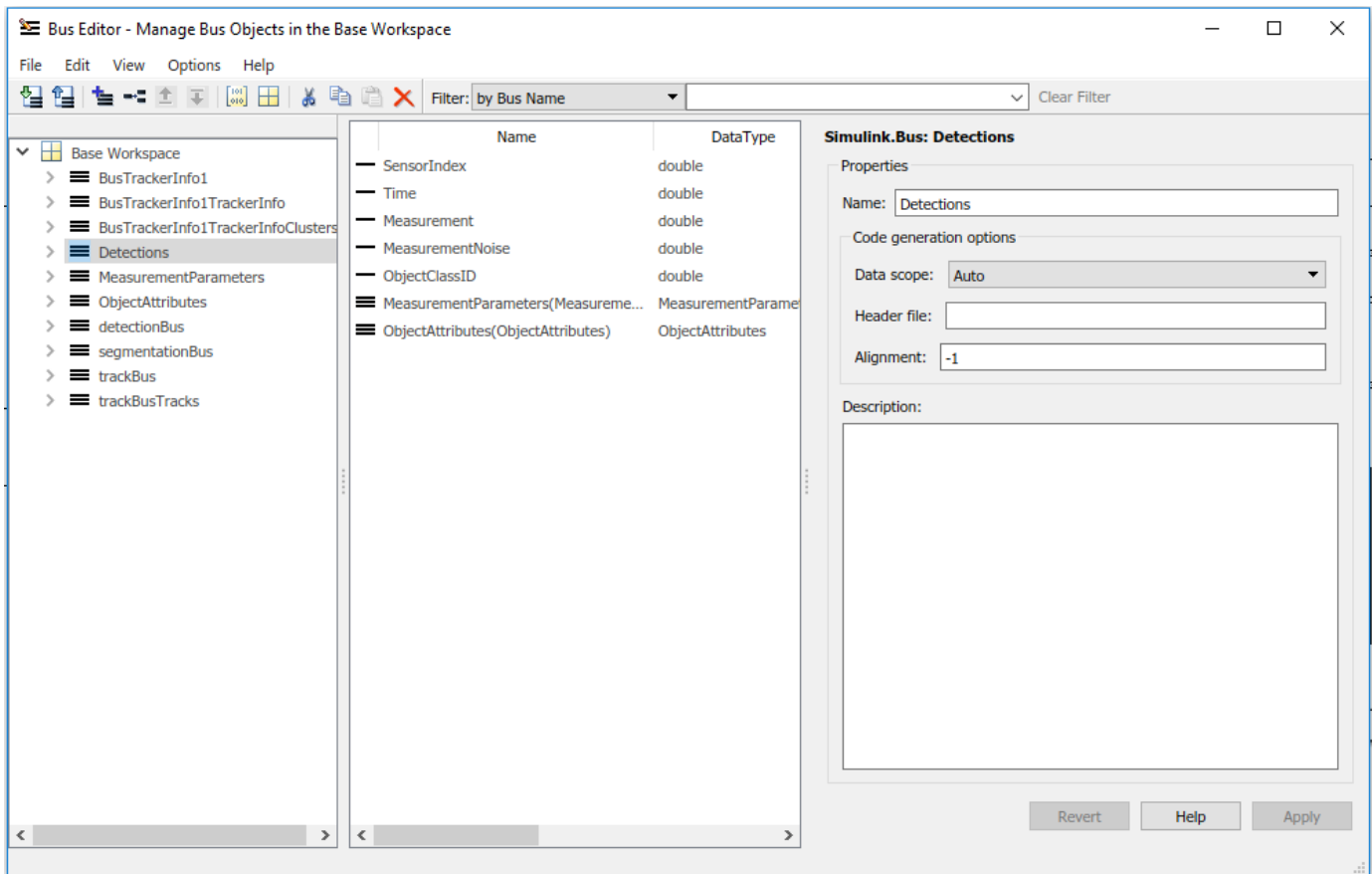
Tracks

The track bus is similar to the detections bus. It is a nested bus, where `NumTracks` defines the number of tracks in the bus and `Tracks` define a fixed size of tracks. The size of the tracks is governed by the block parameter "Maximum number of tracks".
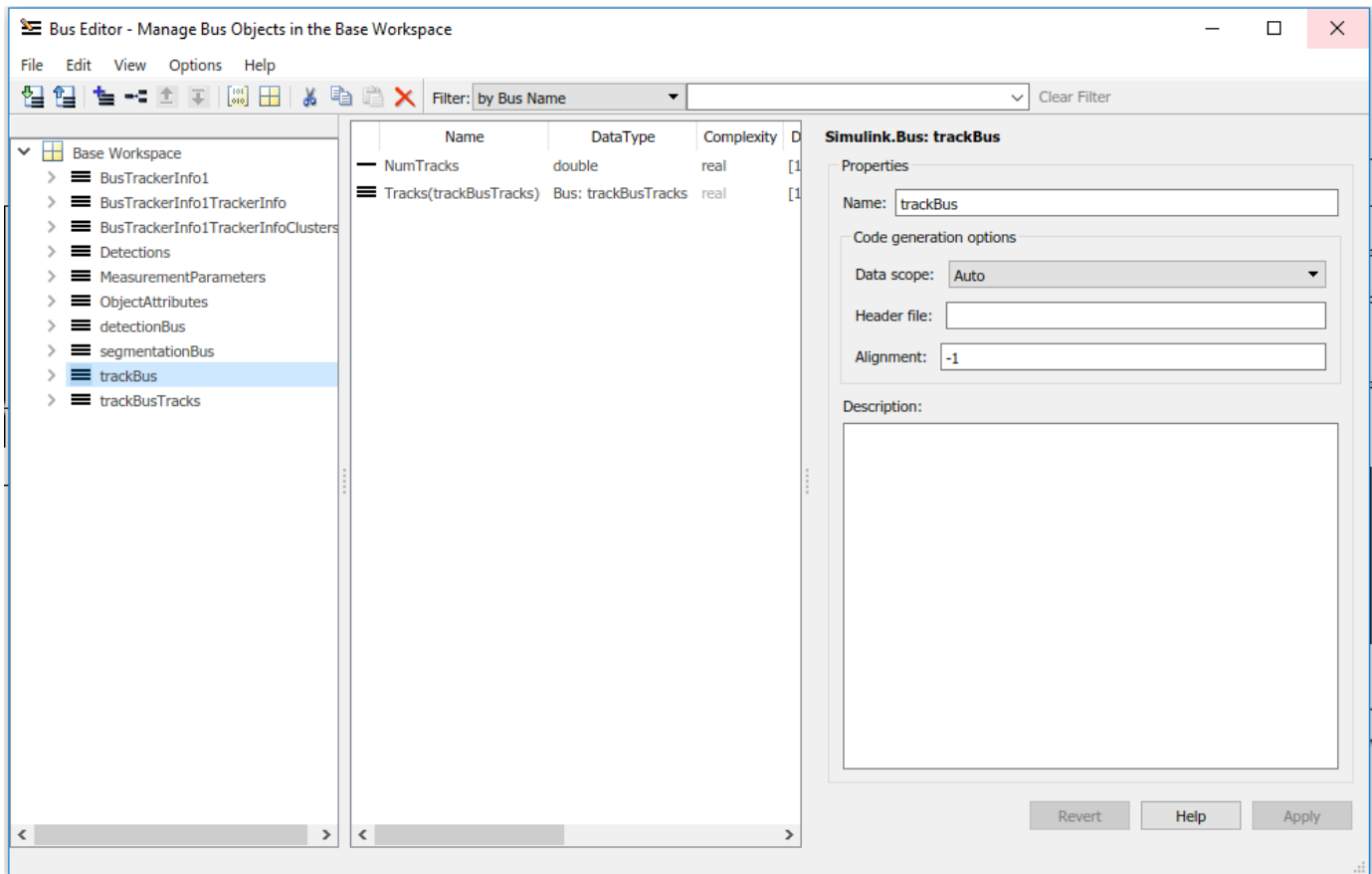
The second element `Tracks` is a bus object defined by `trackBusTracks`. This bus is automatically created by the tracker block by using the bus name specified as the prefix. Notice that the structure of the bus is similar to the `objectTrack` class.
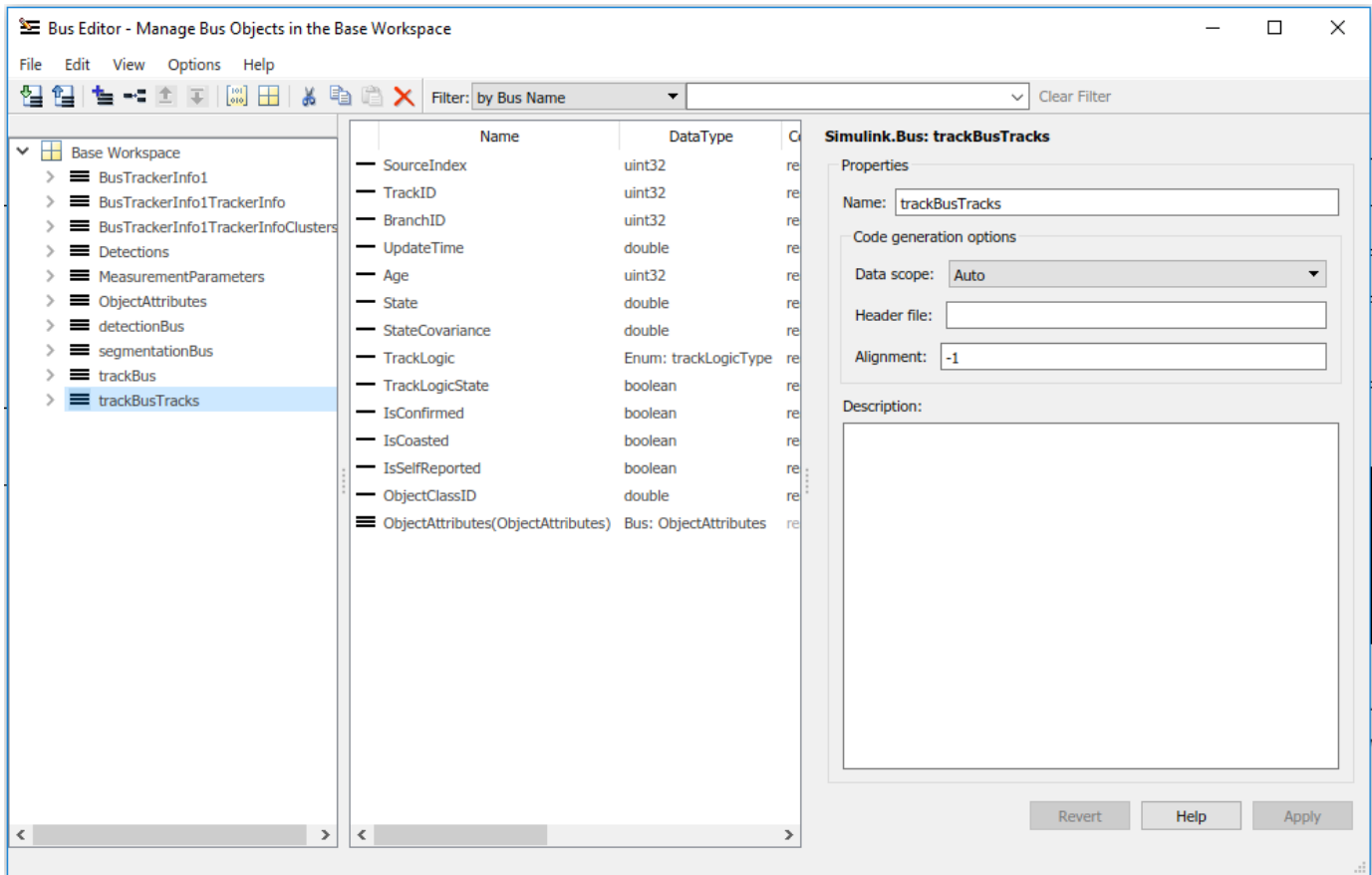
Results

The detector and tracker algorithm is configured exactly as the "Track Vehicles Using Lidar: From Point Cloud to Track List" on page 5-369 MATLAB example. After running the model, you can visualize the results on the figure. The animation below shows the results from time 0 to 4 seconds. The tracks are represented by green bounding boxes. The bounding box detections are represented by orange bounding boxes. The detections also have orange points inside them, representing the point cloud segmented as obstacles. The segmented ground is shown in purple. The cropped or discarded point cloud is shown in blue. Notice that the tracked objects are able to maintain their shape and kinematic center by positioning the detections onto visible portions of the vehicles. This illustrates the offset and shrinkage effect modeled in the measurement functions.
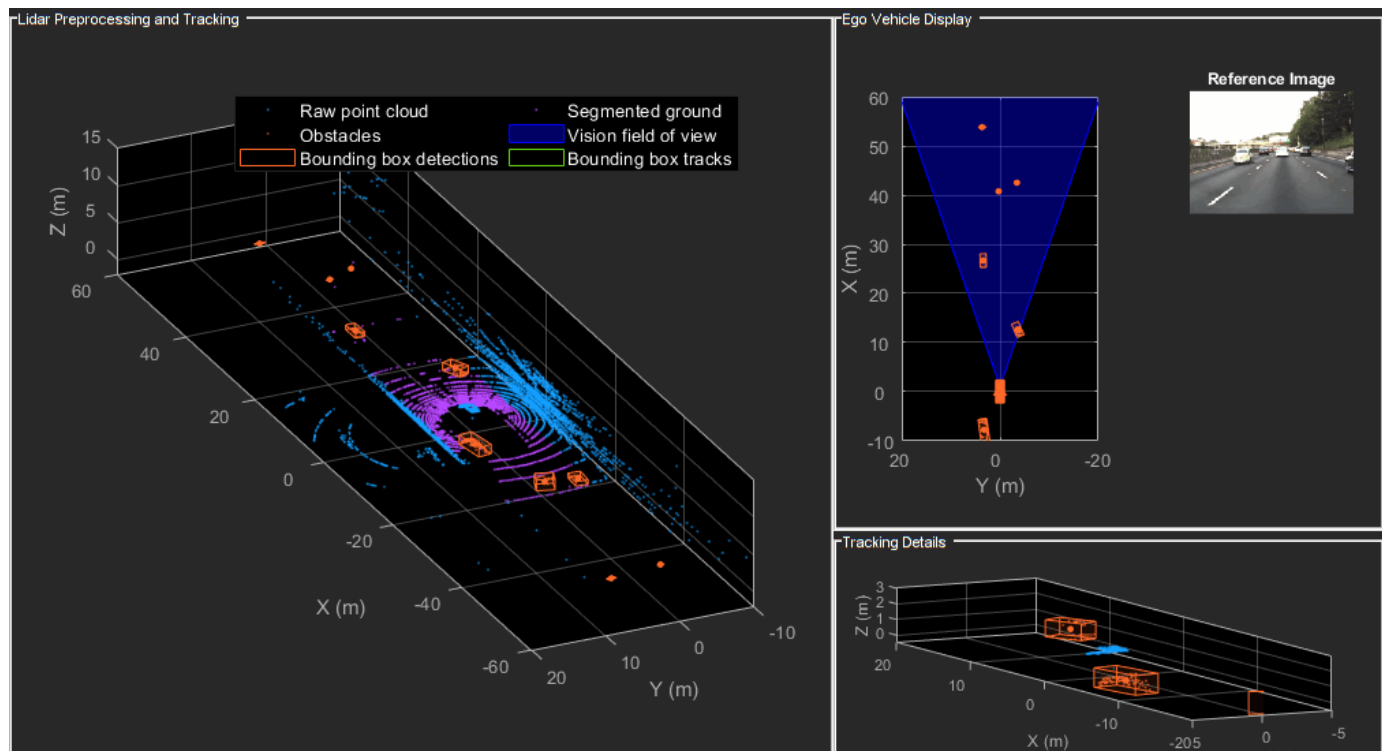
```
close_system('TrackVehiclesSimulinkExample');
```

Summary

This example showed how to use a JPDA tracker with an IMM filter to track objects using a lidar sensor. You learned how a raw point cloud can be preprocessed to generate detections for conventional trackers, which assume one detection per object per sensor scan. You also learned how to use a cuboid model to describe the extended objects being tracked by the JPDA tracker.

Track Closely Spaced Targets Under Ambiguity in Simulink

This example shows how to track objects in Simulink® with Sensor Fusion and Tracking Toolbox™ when the association of sensor detections to tracks is ambiguous. It closely follows the "Tracking Closely Spaced Targets Under Ambiguity" on page 5-167 MATLAB® example.

Introduction

Sensors report a single detection for multiple targets when the targets are so closely spaced that the sensors cannot resolve them spatially. This example illustrates the workflow for tracking targets under such ambiguity using the global nearest neighbor (GNN), joint probabilistic data association (JPDA), and track-oriented multiple-hypothesis (TOMHT) trackers.

Setup and Overview of the Model

Prior to running this example, the scenario was generated as described in "Tracking Closely Spaced Targets Under Ambiguity" on page 5-167. The detection and time data from this scenario were then saved to the scenario file CloselySpacedData.mat.
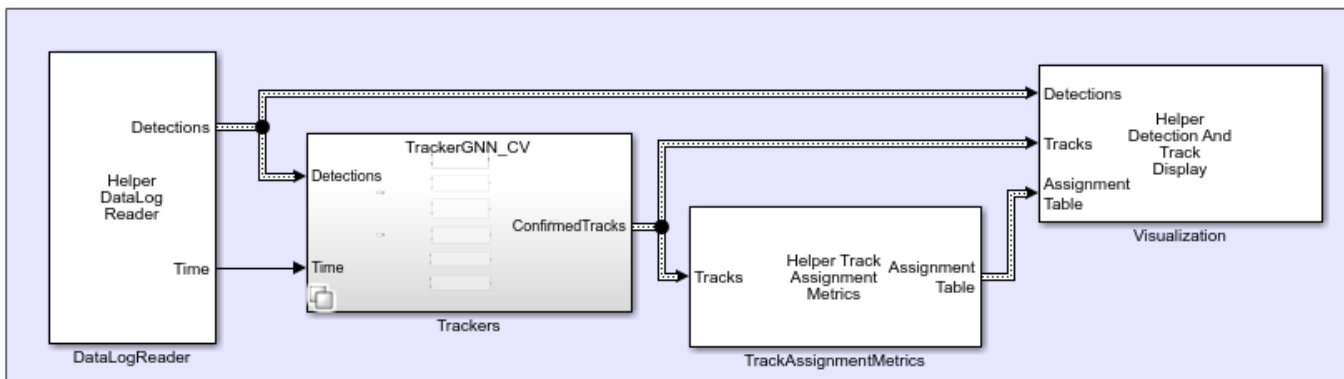
DataLogReader

The DataLogReader block is implemented as a MATLAB System (Simulink) block. The code for the block is defined by a helper class, HelperDataLogReader. The block reads the recorded data from the CloselySpacedData.mat file and outputs the detection and time for each time stamp.

Trackers

The Trackers block is a variant subsystem block, which has six subsystems defined internally. Each subsystem is composed of one of the three trackers and one of the two motion models.

Condition	Selected Tracker Configuration
Tracker = 1	GNN tracker with a constant velocity model
Tracker = 2	GNN tracker with interacting multiple models
Tracker = 3	JPDA tracker with a constant velocity model
Tracker = 4	JPDA tracker with interacting multiple models
Tracker = 5	TOMHT tracker with a constant velocity model
Tracker = 6	TOMHT tracker with interacting multiple models

The first motion model is a constant velocity model with an extended kalman filter. The `helperCVFilter` function modifies the filter `initcvekf` returns to allow for a higher uncertainty in the velocity terms and a higher horizontal acceleration in the process noise.

The second filter you use is an interacting multiple-model (IMM) filter, which allows you to define two or more motion models for the targets. The filter automatically updates the likelihood of each motion model based on the give measurements and estimates the target state and uncertainty based on these models and likelihoods. In this example, the targets switch between straight legs at a constant velocity motion and legs of constant turn rate. Therefore, you define an IMM filter with a constant velocity model and a constant turn-rate model using the `helperIMMFilter` function.

You can run the different configurations by changing the value of `Tracker` in the workspace as shown in the above table. You can also use the "Edit and Manage Workspace Variables by Using Model Explorer" (Simulink) as shown below to change the value of `Tracker`.
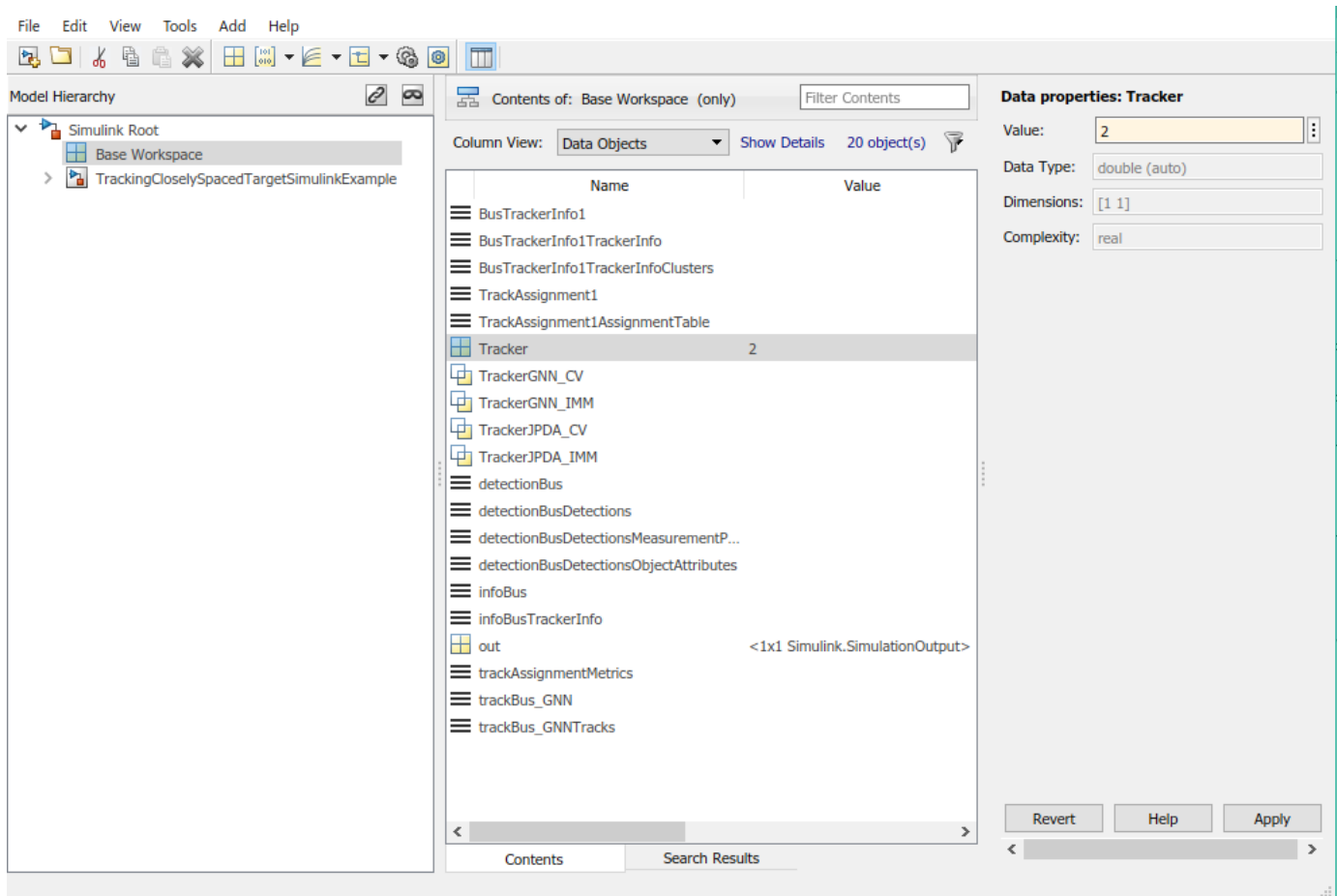
TrackAssignmentMetrics

The TrackAssignmentMetrics is implemented using a MATLAB System (Simulink) block. The code for the block is defined by a helper class, `HelperTrackAssignmentMetrics`.

Visualization

The visualization block is implemented using a MATLAB System (Simulink) block. The code for the block is defined by a helper class, `HelperDetectionAndTrackDisplay`.

Detections and Tracks Bus Objects

The trackers variant subsystem block receives the detections as a bus object with time values and outputs the tracks as a bus object to the visualization block. You can visualize the structure of each bus using the Bus Editor (Simulink). The following images show the structure of the bus for detections and tracks.

Detections Bus

The `detectionBus` outputs a nested bus object with 2 elements, `NumDetections` and `Detections`.
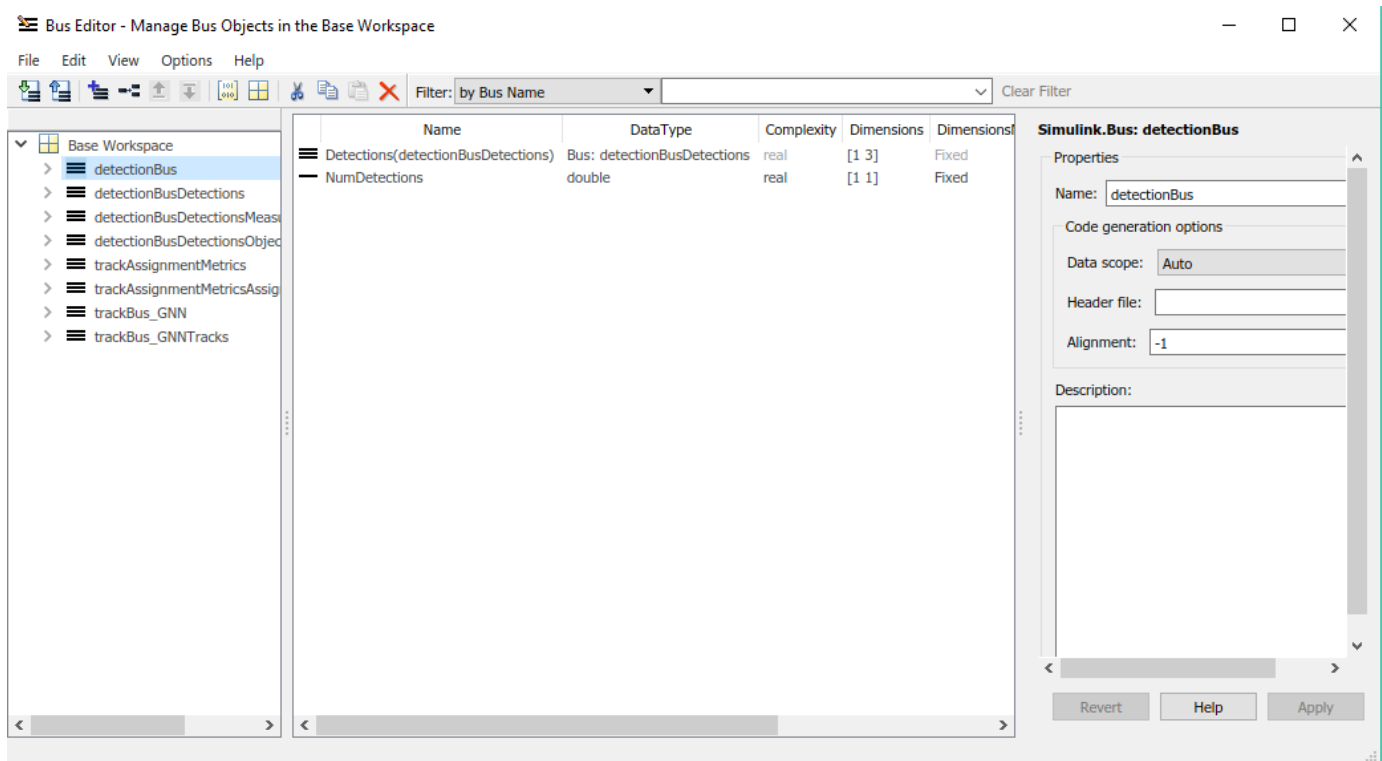
The first element `Detections` is a fixed-size bus object representing all detections. The second element, `NumDetections`, represents the number of detections. The structure of the bus is similar to the `objectDetection` class.
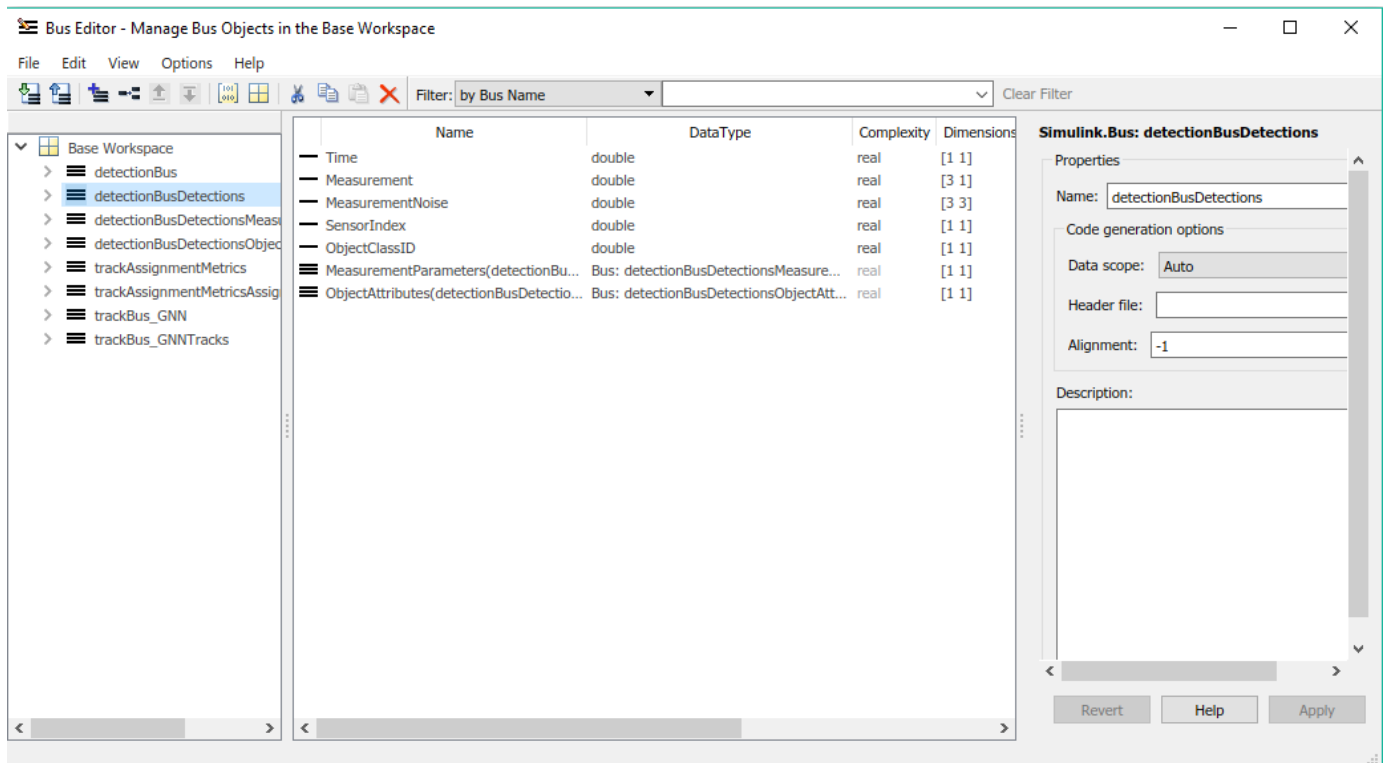
Tracks Bus

The track bus is similar to the detections bus. The track bus is a nested bus, where `NumTracks` defines the number of tracks in the bus and `Tracks` define a fixed size of tracks. The size of the tracks is governed by the block parameter `Maximum number of tracks`. The figure below shows the configuration of the tracks bus object for `trackerGNN`. The configuration of the tracks bus object for `trackerJPDA` and `trackerTOMHT` is similar.
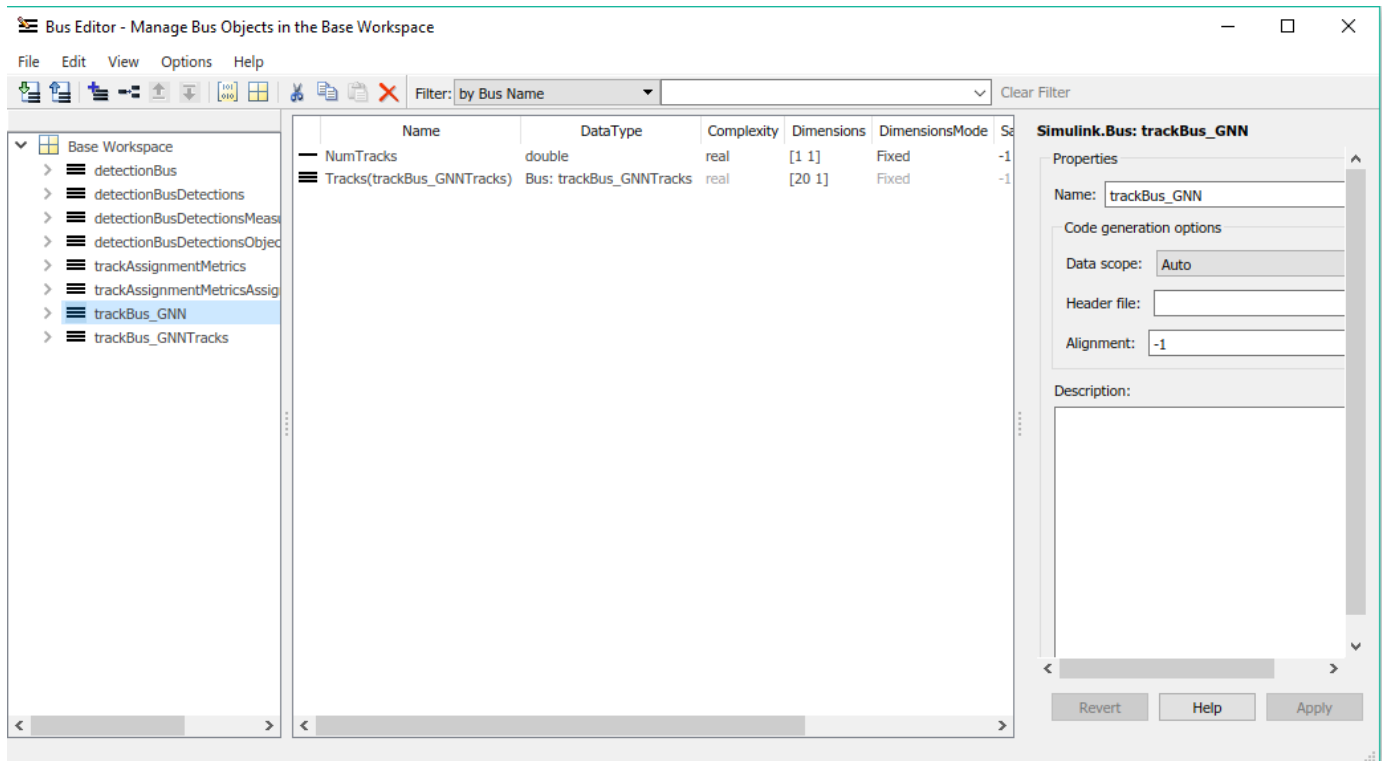
The second element `Tracks` is a bus object defined by `trackBus_GNNTracks` for the trackerGNN configuration. This bus is automatically created by the tracker block using the bus name specified as the prefix. The structure of the bus is similar to the `objectTrack` class.
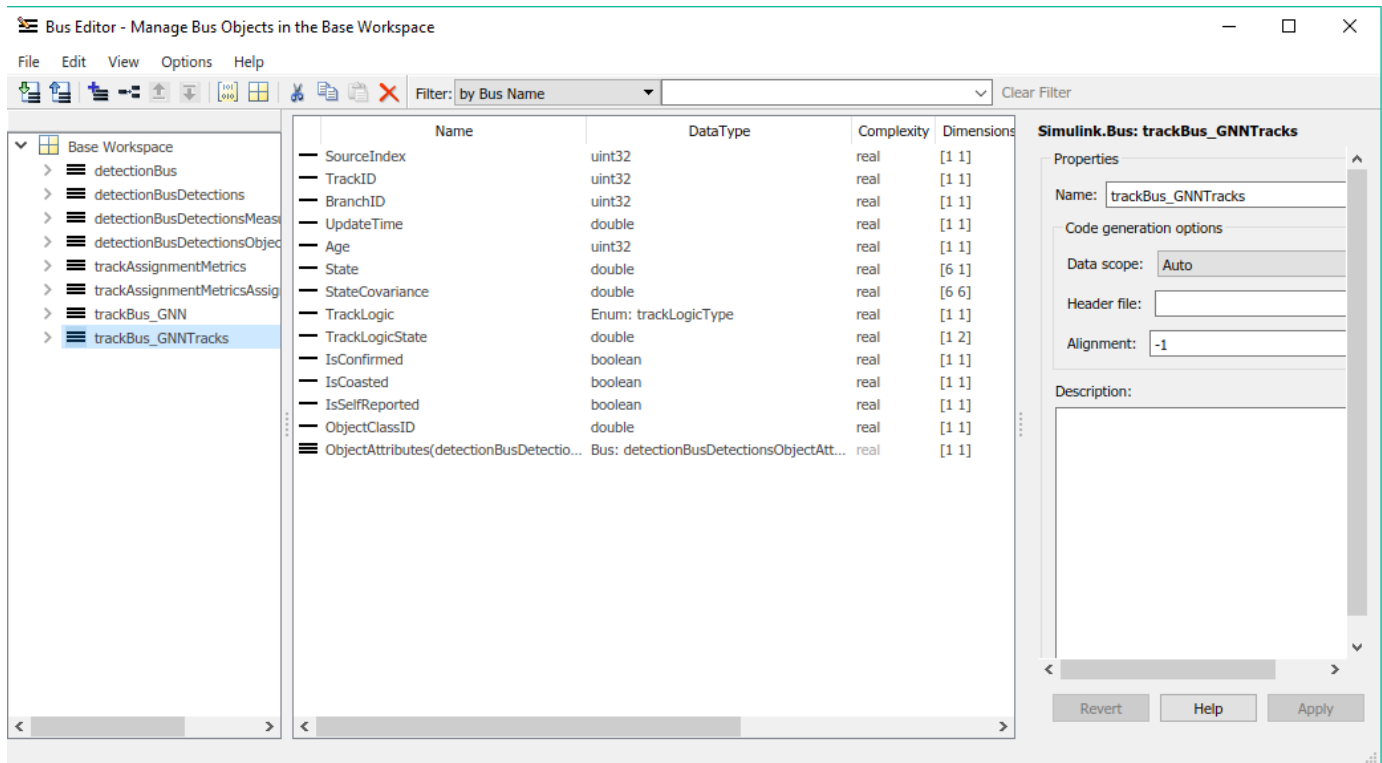
Results

Tracker configuration in the Trackers variant subsystem block is similar to that of the "Tracking Closely Spaced Targets Under Ambiguity" on page 5-167 MATLAB® example.

You can run the `trackerGNN`, `trackerJPDA`, and `trackerTOMHT` blocks in Simulink® models via interpreted execution or code generation. With interpreted execution, the model simulates the block using the MATLAB® execution engine which allows faster startup time but longer execution time. With code generation, the model uses the subset of MATLAB code supported for code generation which allows better performance than the interpreted execution.

After running the model you can visualize the results on the figures.
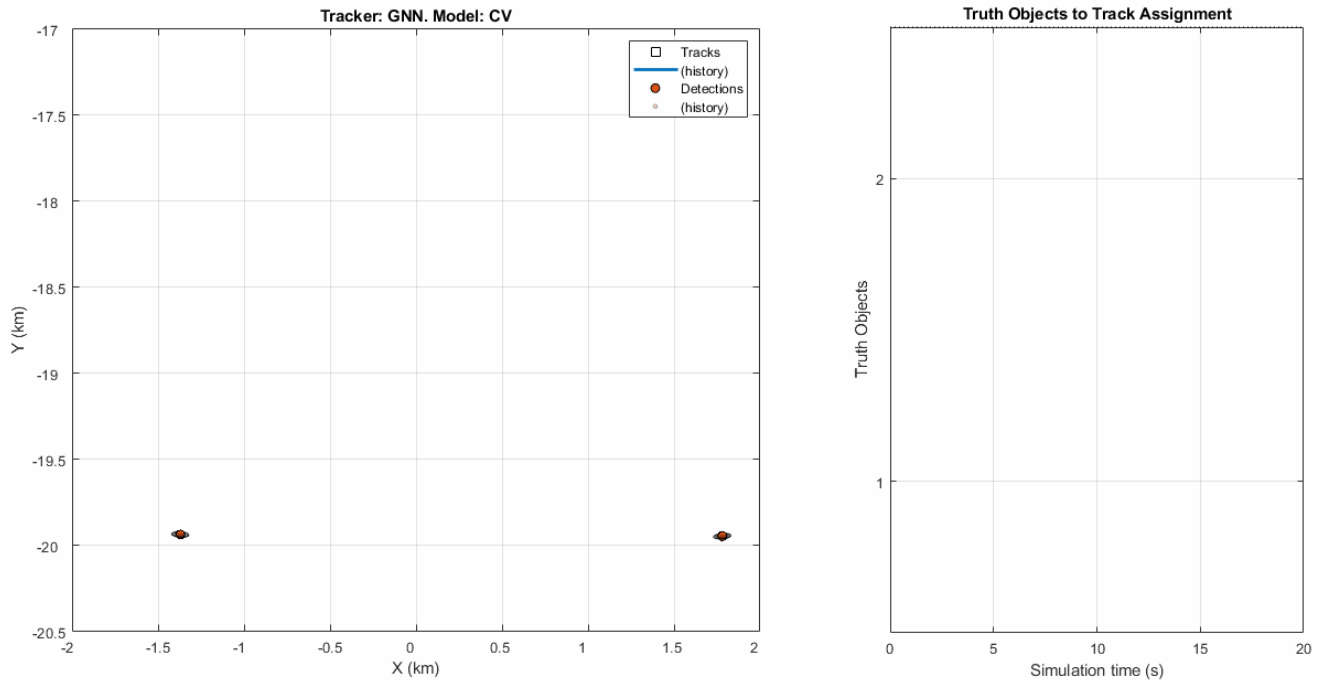
The above results are achieved from the first configuration where `Tracker = 1` in MATLAB workspace. These results show that there are two truth objects. However, the tracker generate three confirmed tracks, and one of the tracks did not survive until the end of the scenario. At the end of the scenario, the tracker associates truth object 1 with track 8. The tracker created track 8 through the scenario after dropping track 2. The tracker assigned truth object 2 to track 1 at the end of the scenario after there were two track breaks in the tracking history.
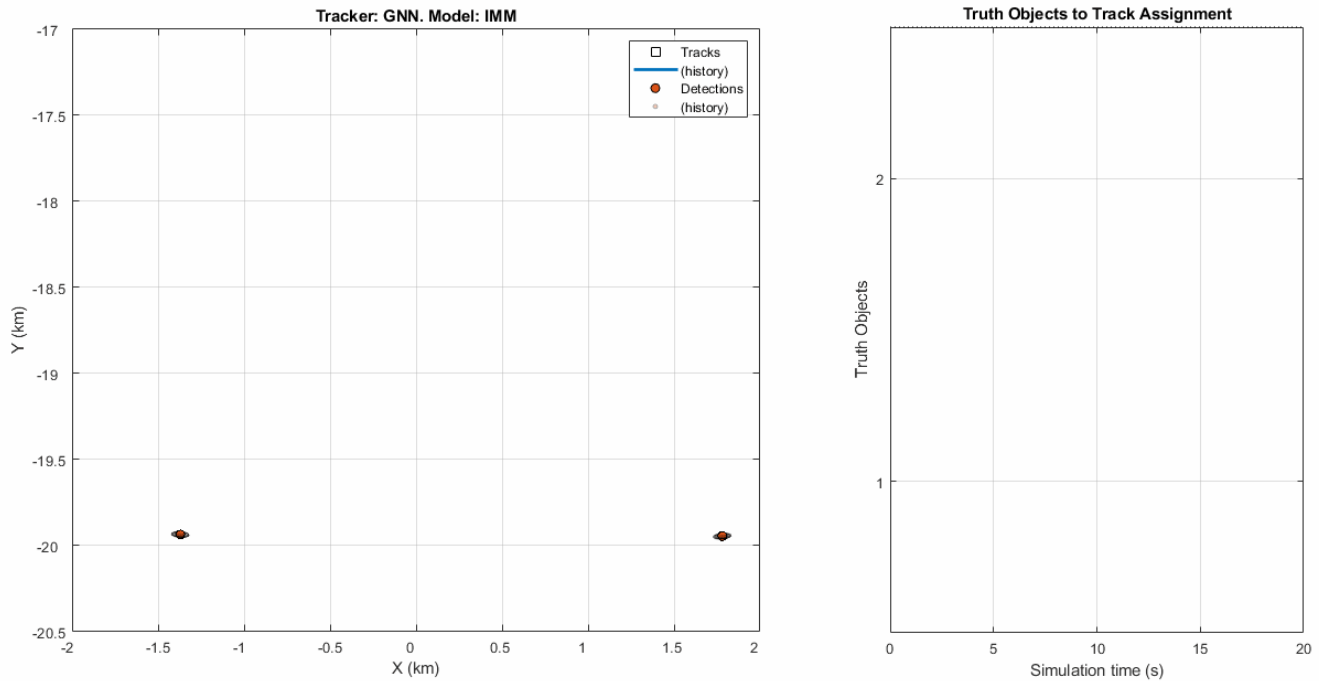
The above results are achieved from the second configuration where `Tracker` = 2 in MATLAB workspace. The IMM filter enables the trackerGNN to track the maneuvering target correctly. Notice that truth object 1 has zero breaks due to continuous history of its associated track.

However, even with the IMM filter, one of the tracks breaks in the ambiguity region. The `trackerGNN` receives only one detection in that ambiguity region, and therefore can update only one of the tracks with it. After a few updates, the score of the coasted track falls below the deletion threshold and the tracker drops the track.
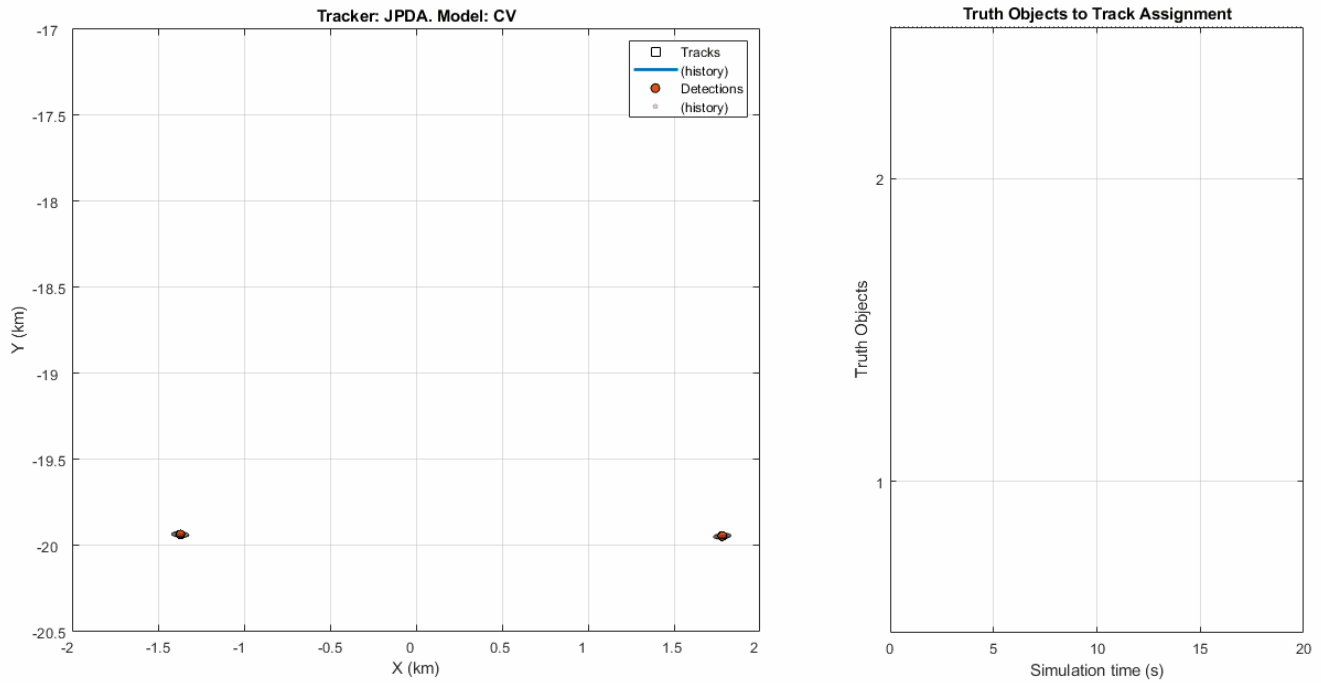
The above results are achieved from the third configuration where `Tracker` = 3 in MATLAB workspace. You can observe that the trackerJPDA maintains both the tracks confirmed in the ambiguity region. However, there is a swap between the two tracks.
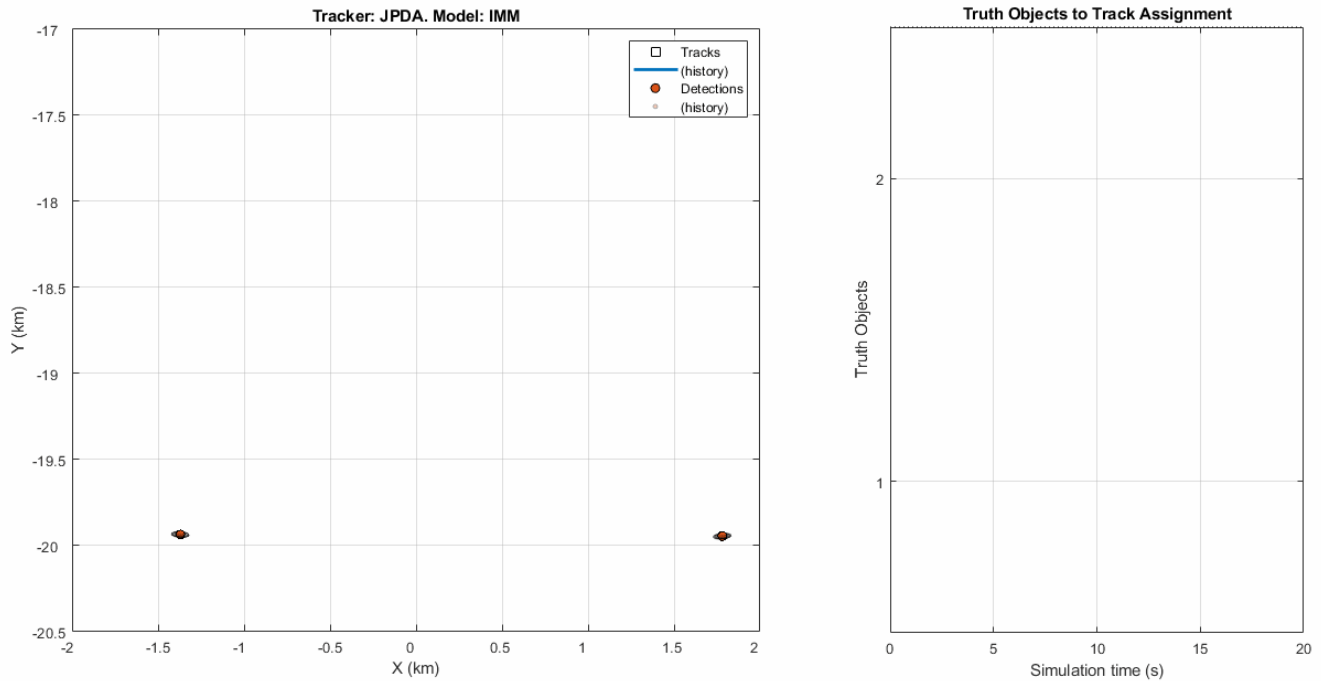
The above results are achieved from the fourth configuration where `Tracker` = 4 in MATLAB workspace. You can observe that trackerJPDA with IMM filter tracks the maneuvering targets more precisely and did not break or lose the track even during the turns.
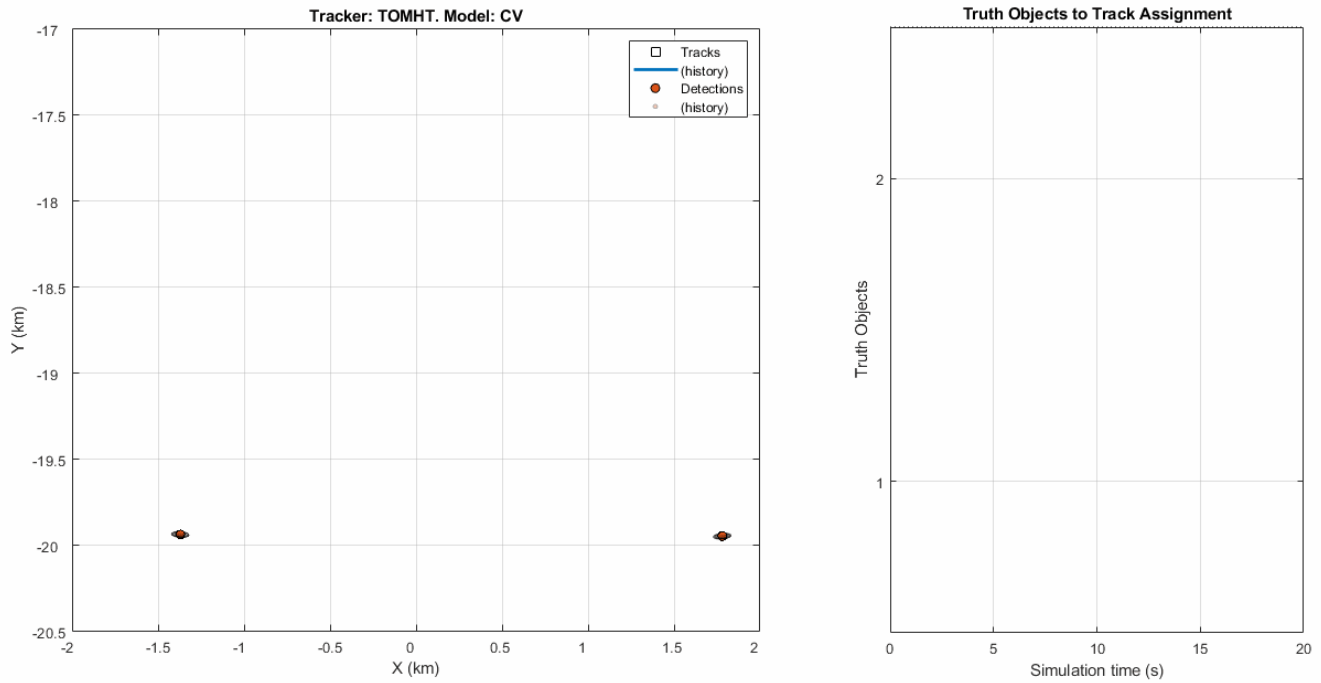
Next, the fifth configuration is used, with `trackerTOMHT` and a constant velocity model by setting `Tracker` = 5 in MATLAB workspace. You can observe that the trackerTOMHT result is similar to the one obtained by the trackerJPDA: it maintains the tracks through the ambiguity region, but relying only on a constant velocity model causes the tracks to swap.
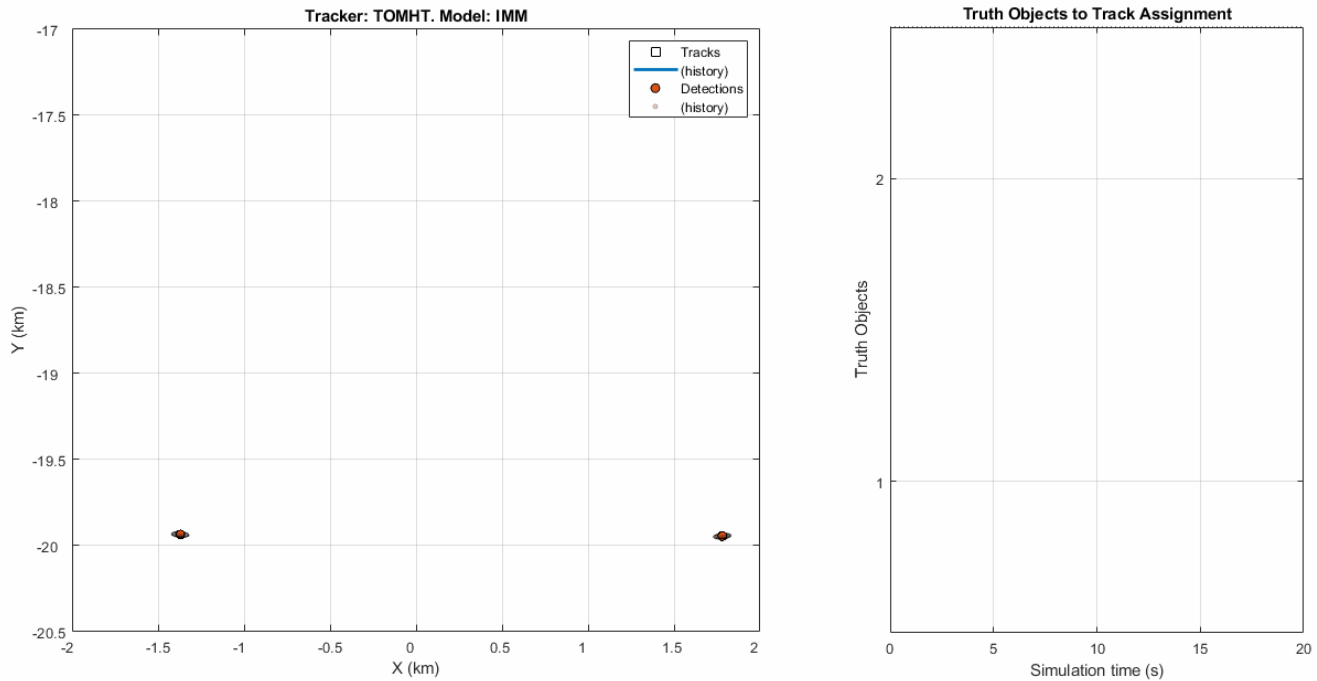
Finally, we use `trackerTOMHT` and IMM filter by setting `Tracker = 6` in MATLAB workspace. This time, the trackerTOMHT with IMM filter tracks the maneuvering targets more precisely and did not break or lose the track even during the turns. However, the runtime for a trackerTOMHT is significantly longer than using trackerJPDA.

Summary

In this example, you learned how to track closely spaced targets using three types of trackers: global nearest neighbor, joint probabilistic data association, and track-oriented multiple hypothesis. You saw how to use a variant subsystem in Simulink to select which tracker and filter to run. You also learned how you can utilize and configure `trackerGNN`, `trackerJPDA`, and `trackerTOMHT` Simulink blocks for tracking the maneuvering targets.

You observed that a constant velocity filter is insufficient when tracking the targets during their maneuver. In this case, an interacting multiple-model filter is required. You also observed that the JPDA and TOMHT trackers can more accurately handle the case of ambiguous association of detections to tracks compared with the GNN tracker.

Track-to-Track Fusion for Automotive Safety Applications

This example shows how to fuse tracks from two vehicles in order to provide a more comprehensive estimate of the environment that can be seen by each vehicle. The example demonstrates the use of a track-level fuser and the object track data format. In this example, you use the driving scenario and models from Automated Driving Toolbox™ and the tracking and track fusion models from Sensor Fusion and Tracking Toolbox™.

Motivation

Automotive safety applications rely on the fusion of data from different sensor systems mounted on the vehicle. Individual vehicles fuse sensor detections by using either a centralized tracker or by taking a more decentralized approach and fusing tracks produced by individual sensors. In addition to intravehicle data fusion, the fusion of data from multiple vehicles provides added benefits, which include better coverage, situational awareness, and safety. [1] This intervehicle sensor fusion approach takes advantage of the variety of sensors and provides better coverage to each vehicle, because it uses data updated by sensors on other vehicles in the area. Governments and vehicle manufacturers have long recognized the need to share information between vehicles in order to increase automotive safety. For example, the Dedicated Short-Range Communications (DSRC) Service was established to provide a communications service for intervehicle information sharing. [2]

While sensor fusion across multiple vehicles is beneficial, most vehicles are required to meet certain safety requirements even if only internal sensors are available. Therefore, the vehicle is likely to be equipped with a tracker and/or a track fuser that provide situational awareness at the single vehicle level. As a result, the assumption made in this example is that vehicles share situational awareness by broadcasting tracks and performing track-to-track fusion.

This example demonstrates the benefit of fusing tracks from two vehicles to enhance situational awareness and safety. Note that this example does not simulate the communications systems. Instead, the example assumes that a communications system provides the bandwidth required to transmit tracks between the two vehicles.

Track-to-Track Architecture

The following block diagram depicts the main functions in the two vehicles.

Vehicle 1 has two sensors, each providing detections to a local tracker. The tracker uses the detections from the local sensors to track objects and outputs these local tracks to the vehicle track fuser. Vehicle 2 has a single sensor, which feeds detections to the local tracker on vehicle 2. The local tracks from vehicle 2 are the input to the local track fuser on vehicle 2.

The track fuser on each vehicle fuses the local vehicle tracks with the tracks received from the other vehicle's track fuser. After each update, the track fuser on each vehicle broadcasts its fused tracks, which feed into the next update of the track fuser on the other vehicle.
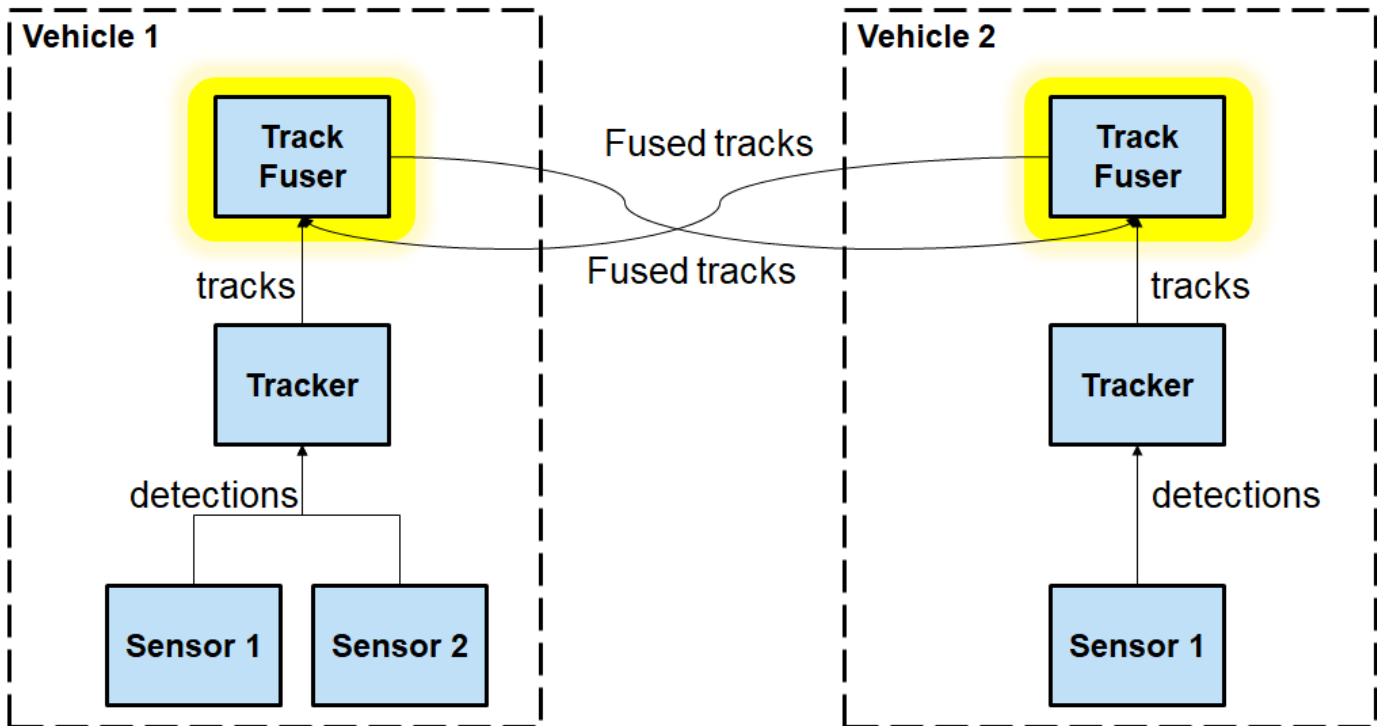
In this example, you use a `trackerJPDA` object to define each vehicle tracker.

```
% Create trackers for each vehicle
v1Tracker = trackerJPDA('TrackerIndex',1, 'DeletionThreshold', [4 4]); % Vehicle 1 tracker
v2Tracker = trackerJPDA('TrackerIndex',2, 'DeletionThreshold', [4 4]); % Vehicle 2 tracker
posSelector = [1 0 0 0 0 0; 0 0 1 0 0 0];
```

Note that in this architecture, the fused tracks from one vehicle are used to update the fused tracks on the other vehicle. These fused tracks are then broadcast back to the first vehicle. To avoid rumor propagation, be careful how tracks from another vehicle are used to update the track fuser.

Consider the following rumor propagation example: at some update step, vehicle 1 tracks an object using its internal sensors. Vehicle 1 then fuses the object track and transmits it to vehicle 2, which now fuses the track with its own tracks and becomes aware of the object. Up to this point, this is exactly the goal of track-to-track fusion: to enhance the situational awareness of vehicle 2 with information from vehicle 1. Since vehicle 2 now knows about the object, it starts broadcasting the track as well, perhaps for the benefit of another vehicle (not shown in the example).

However, vehicle 1 now receives track information from vehicle 2 about the object that only vehicle 1 actually tracks. So, the track fuser on vehicle 1 has to be aware that the tracks it gets from vehicle 2 about this object do not actually contain any new information updated by an independent source. To make the distinction between tracks that contain new information and tracks that just repeat information, you must define vehicle 2 as an *external source* to the track fuser on vehicle 1. Similarly, vehicle 1 must be defined as an external source to the track fuser on vehicle 2. Furthermore, you need to define only tracks that are updated by a track fuser based on information from an internal source as *self-reported*. By doing so, the track fuser in each vehicle is able to ignore updates from tracks that bounce back and forth between the track fusers without any new information in them.

The local tracker of each vehicle tracks objects relative to the vehicle reference frame, called the ego frame. The track-to-track fusion is done at the scenario frame, which is the global level frame. The

helper `egoToScenario` function transforms tracks from ego frame to the scenario frame. Similarly, the function `scenarioToEgo` transforms tracks from scenario frame to any of the ego frames. Both transformations rely on the `StateParameters` property of the `objectTrack` objects. Note that when the `trackFuser` object calculates the distance of a central track (in the scenario frame) to a local track (in any frame), it uses the `StateParameters` of the local track to perform the coordinate transformation.

To achieve the above `trackFuser` definitions, define the following sources as a `fuserSourceConfiguration` object.

```matlab
% Define sources for each vehicle
v1TrackerConfiguration = fuserSourceConfiguration('SourceIndex',1,'IsInternalSource',true, ...
    "CentralToLocalTransformFcn", @scenarioToEgo, 'LocalToCentralTransformFcn', @egoToScenario);
v2FuserConfiguration = fuserSourceConfiguration('SourceIndex',4,'IsInternalSource',false);
v1Sources = {v1TrackerConfiguration; v2FuserConfiguration};
v2TrackerConfiguration = fuserSourceConfiguration('SourceIndex',2,'IsInternalSource',true, ...
    "CentralToLocalTransformFcn", @scenarioToEgo, 'LocalToCentralTransformFcn', @egoToScenario);
v1FuserConfiguration = fuserSourceConfiguration('SourceIndex',3,'IsInternalSource',false);
v2Sources = {v2TrackerConfiguration; v1FuserConfiguration};
```

You can now define each vehicle track fuser as a `trackFuser` object.

```matlab
stateParams = struct('Frame','Rectangular','Position',[0 0 0],'Velocity',[0 0 0]);
v1Fuser = trackFuser('FuserIndex',3,...
    'MaxNumSources',2,'SourceConfigurations',v1Sources,...
    'StateFusion','Intersection','DeletionThreshold',[3 3],...
    'StateParameters',stateParams);
v2Fuser = trackFuser('FuserIndex',4,...
    'MaxNumSources',2,'SourceConfigurations',v2Sources,'StateFusion',...
    'Intersection','DeletionThreshold',[3 3],...
    'StateParameters',stateParams);
```

```matlab
% Initialize the following variables
fusedTracks1 = objectTrack.empty(0,1);
fusedTracks2 = objectTrack.empty(0,1);
wasFuser1Updated = false;
wasFuser2Updated = false;
```

Scenario Definition

The following scenario shows two vehicles driving down a street. Vehicle 1 is in the lead, equipped with two forward-looking sensors: a short-range radar and a vision sensor. Vehicle 2, driving 10 meters behind vehicle 1, is equipped with a long-range radar. The right side of the street contains parked vehicles. A pedestrian stands between the vehicles. This pedestrian is shown as a dot at about X = 60 meters.

Due to the short distance between vehicle 2 and vehicle 1, most of the vehicle 2 radar sensor coverage is occluded by vehicle 1. As a result, most of the tracks that the track fuser on vehicle 2 maintains are first initialized by tracks broadcast from vehicle 1.

```matlab
% Create the drivingScenario object and the two vehicles
[scenario, vehicle1, vehicle2] = createDrivingScenario;

% Create all the sensors
[sensors, numSensors, attachedVehicle] = createSensors(scenario);
```

```
% Create display
[f,plotters] = createT2TDisplay(scenario, sensors, attachedVehicle);
```

The following chase plot is seen from the point of view of the second vehicle. An arrow indicates the position of the pedestrian that is almost entirely occluded by the parked vehicles and the first vehicle.

Pedestrian at the side of the road
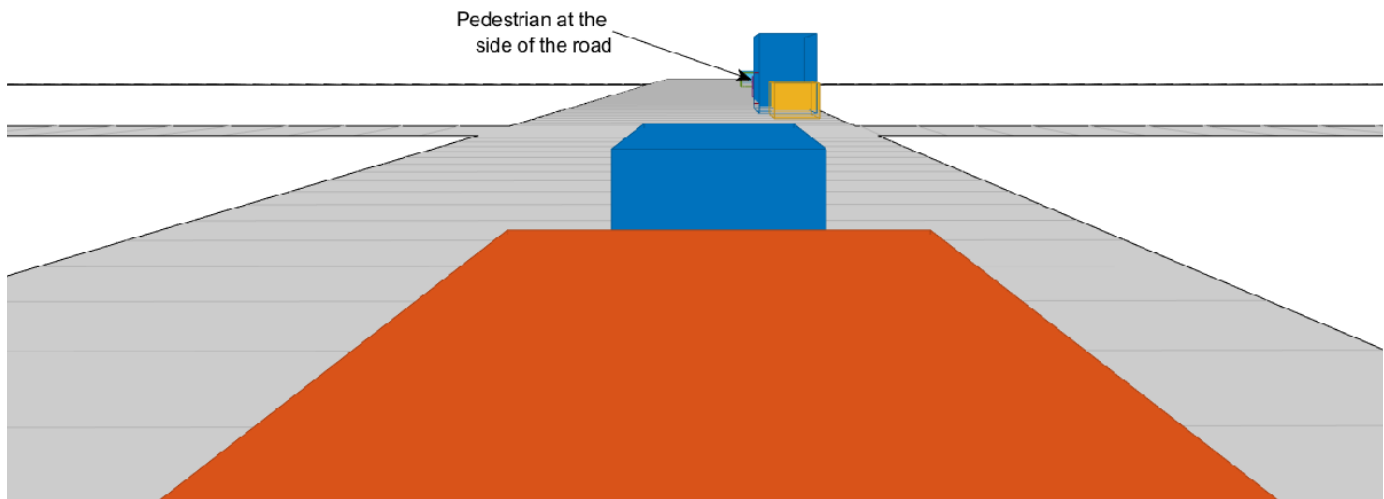
```
% Define each vehicle as a vehicle, sensors, a tracker, and plotters
v1 = struct('Actor', {vehicle1}, 'Sensors', {sensors(attachedVehicle==1)}, 'Tracker', {v1Tracker
v2 = struct('Actor', {vehicle2}, 'Sensors', {sensors(attachedVehicle==2)}, 'Tracker', {v2Tracker
```

Simulation and Results

The following code runs the simulation.

```
running = true;
```

```
% For repeatable results, set the random number seed
s = rng;
rng(2019)
snaptimes = [0.5, 2.6, 4.4, 6.3, inf];
snaps = cell(numel(snaptimes,1));
i = 1;
f.Visible = 'on';
while running && ishghandle(f)
    time  = scenario.SimulationTime;

    % Detect and track at the vehicle level
    [tracks1,wasTracker1Updated] = detectAndTrack(v1,time,posSelector);
```

```
    [tracks2,wasTracker2Updated] = detectAndTrack(v2,time,posSelector);

    % Keep the tracks from the previous fuser update
    oldFusedTracks1 = fusedTracks1;
    oldFusedTracks2 = fusedTracks2;

    % Update the fusers
    if wasTracker1Updated || wasFuser2Updated
        tracksToFuse1 = [tracks1;oldFusedTracks2];
        if isLocked(v1Fuser) || ~isempty(tracksToFuse1)
            [fusedTracks1,~,~,info1] = v1Fuser(tracksToFuse1,time);
            wasFuser1Updated = true;
            pos = getTrackPositions(fusedTracks1,posSelector);
            plotTrack(plotters.veh1FusePlotter,pos);
        else
            wasFuser1Updated = false;
            fusedTracks1 = objectTrack.empty(0,1);
        end
    else
        wasFuser1Updated = false;
        fusedTracks1 = objectTrack.empty(0,1);
    end
    if wasTracker2Updated || wasFuser1Updated
        tracksToFuse2 = [tracks2;oldFusedTracks1];
        if isLocked(v2Fuser) || ~isempty(tracksToFuse2)
            [fusedTracks2,~,~,info2] = v2Fuser(tracksToFuse2,time);
            wasFuser2Updated = true;
            pos = getTrackPositions(fusedTracks2,posSelector);
            ids = string([fusedTracks2.TrackID]');
            plotTrack(plotters.veh2FusePlotter,pos,ids);
        else
            wasFuser2Updated = false;
            fusedTracks2 = objectTrack.empty(0,1);
        end
    else
        wasFuser2Updated = false;
        fusedTracks2 = objectTrack.empty(0,1);
    end

    % Update the display
    updateT2TDisplay(plotters, scenario, sensors, attachedVehicle)

    % Advance the scenario one time step and exit the loop if the scenario is complete
    running = advance(scenario);

    % Snap a shot at required times
    if time >= snaptimes(i)
        snaps{i} = getframe(f);
        i = i + 1;
    end
end
```
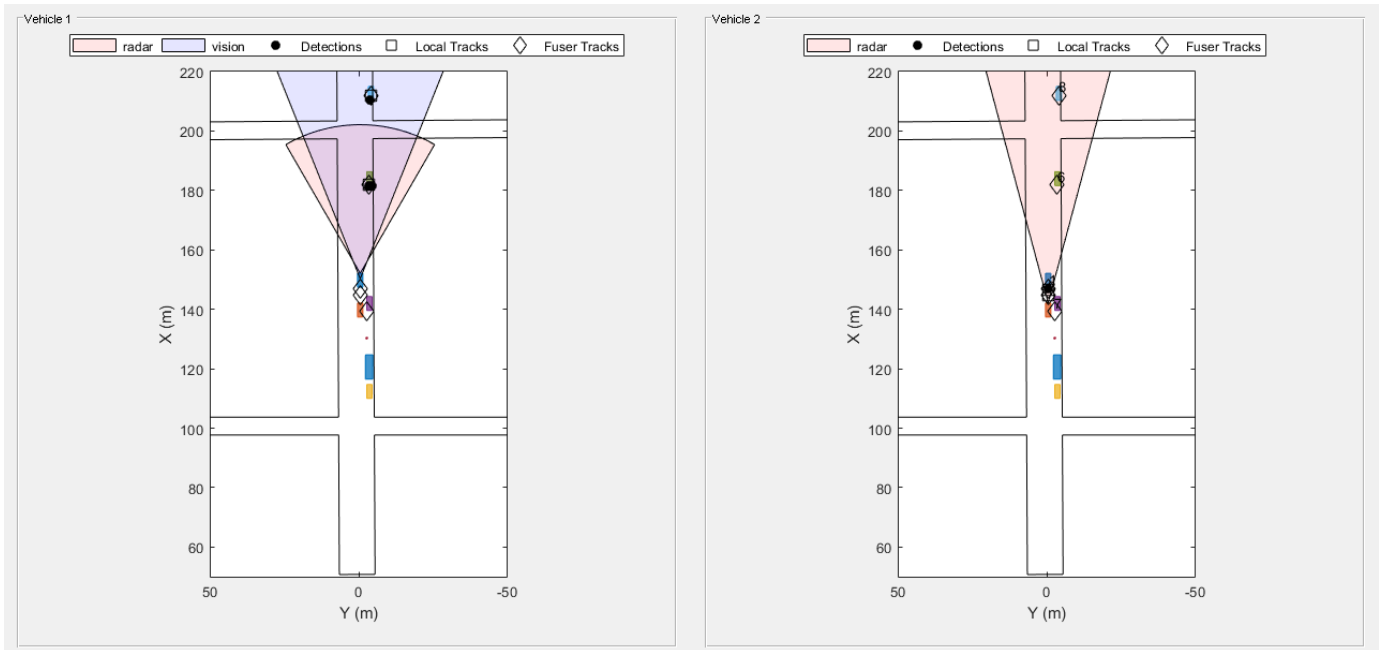
The figure shows the scene and tracking results at the end of the scenario.

Tracking at the Beginning of the Simulation

When the simulation begins, vehicle 1 detects the vehicles parked on the right side of the street, then the tracks associated with the parked vehicles are confirmed. At this time, the only object detected and tracked by vehicle 2 tracker is vehicle 1 immediately in front of it. Once the vehicle 1 track fuser confirms the tracks, it broadcasts them, and the vehicle 2 track fuser fuses them. As a result, vehicle 2 becomes aware of the parked vehicles before it can detect them on its own.

```
showsnap(snaps, 1)
```
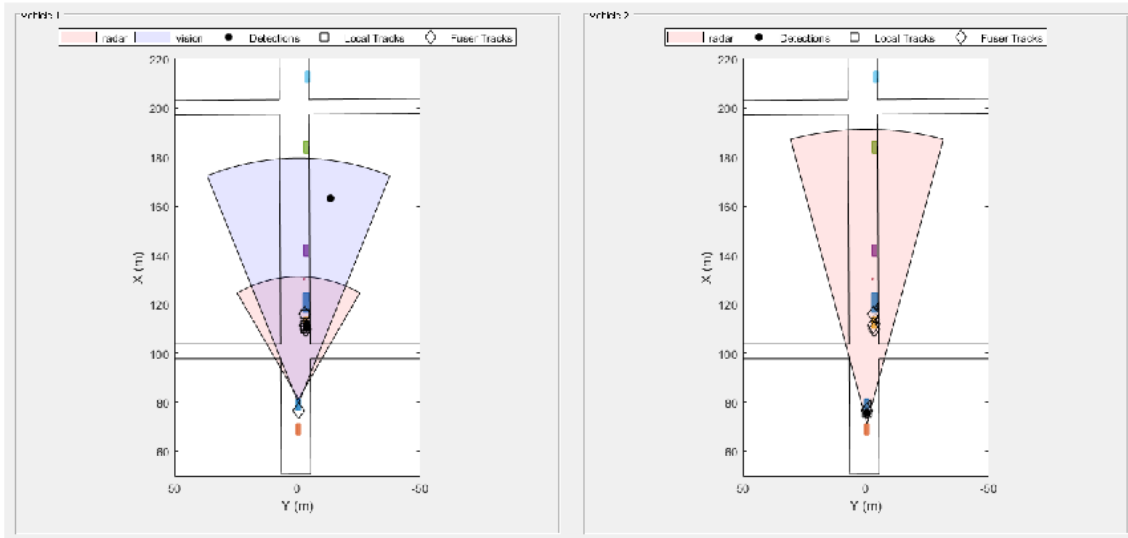
Tracking the Pedestrian at the Side of the Street

As the simulation continues, vehicle 2 is able to detect and track the vehicles parked at the side as well, and fuses them with the tracks coming from vehicle 1. Vehicle 2 is able to detect and track the pedestrian about 4 seconds into the simulation, and vehicle 2 fuses the track associated with the pedestrian around 4.4 seconds into the simulation (see snapshot 2). However, it takes vehicle 2 about two seconds before it can detect and track the pedestrian by its own sensors (see snapshot 3). These two seconds could make a huge impact on the safety of the pedestrian if that pedestrian started crossing the street.

```
showsnap(snaps, 2)
```
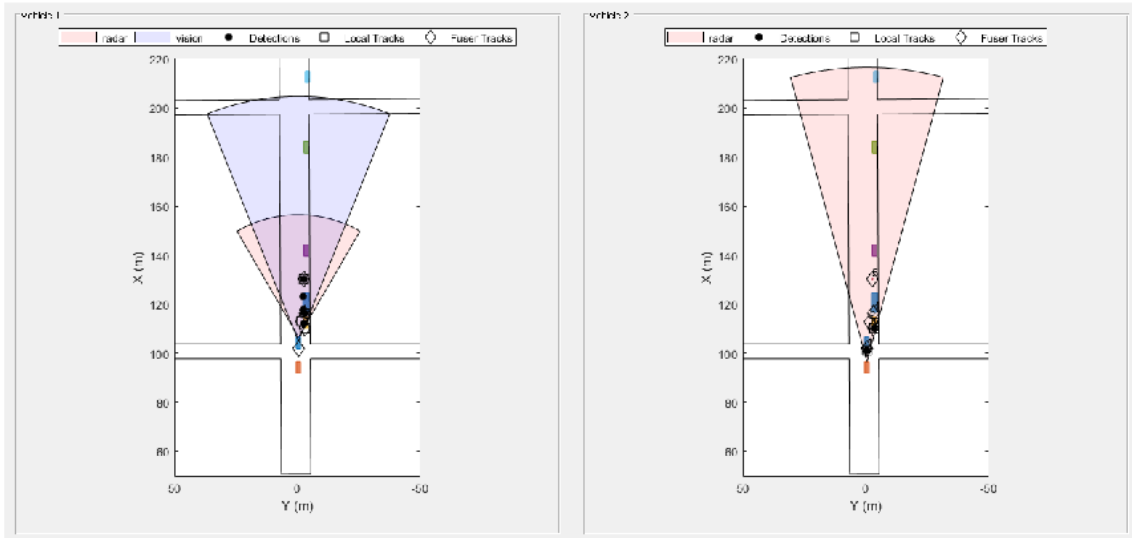
```
showsnap(snaps, 3)
```
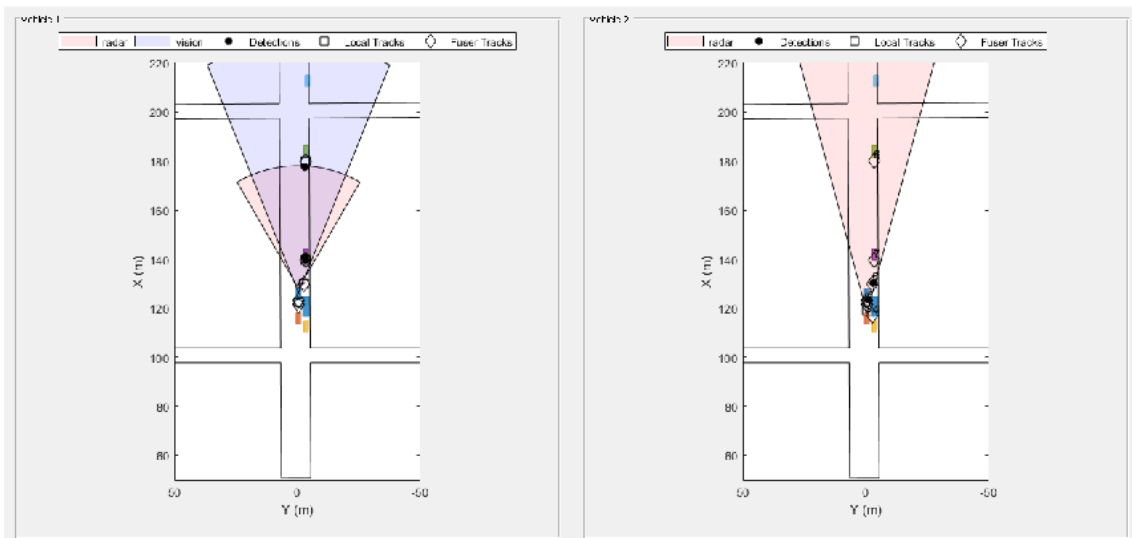

Avoiding Rumor Propagation

Finally, note how as the vehicles pass the objects, and these objects go out of their field of view, the fused tracks associated with these objects are dropped by both trackers (see snapshot 4). Dropping the tracks demonstrates that the fused tracks broadcast back and forth between the two vehicles are not used to propagate rumors.

```
showsnap(snaps, 4)
```
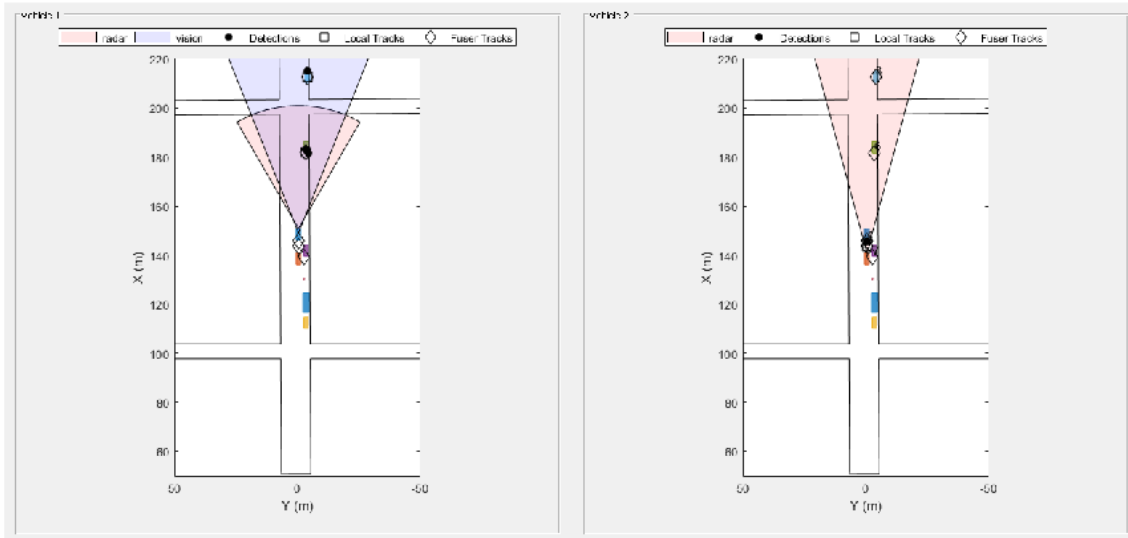
```
% Restart the driving scenario to return the actors to their initial positions.
restart(scenario);

% Release all the sensor objects so they can be used again.
for sensorIndex = 1:numSensors
    release(sensors{sensorIndex});
end

% Return the random seed to its previous value
rng(s)
```

Summary

In this example, you saw how track-to-track fusion can enhance the situational awareness and increase the safety in automotive applications. You saw how to set up a `trackFuser` to perform track-to-track fusion and how to define sources as either internal or external by using the `fuserSourceConfiguration` object. By doing so, you avoid rumor propagation and keep only the fused tracks that are really observed by each vehicle to be maintained.

References

[1] Bharanidhar Duraisamy, Tilo Schwartz, and Christian Wohler, "Track level fusion algorithms for automotive safety applications", 2013 International Conference on Signal Processing, Image Processing & Pattern Recognition, IEEE, 2013.

[2] Federal Communications Commission, "Dedicated Short Range Communications Service", https://www.fcc.gov/wireless/bureau-divisions/mobility-division/dedicated-short-range-communications-dsrc-service.

Supporting Functions

createDrivingScenario

```
function [scenario, egoVehicle, secondVehicle] = createDrivingScenario
% createDrivingScenario Returns the drivingScenario defined in the Designer

% Construct a drivingScenario object.
scenario = drivingScenario('SampleTime', 0.05);

% Add all road segments
roadCenters = [50.8 0.5 0; 253.4 1.5 0];
roadWidth = 12;
road(scenario, roadCenters, roadWidth);

roadCenters = [100.7 -100.6 0; 100.7 103.7 0];
road(scenario, roadCenters);

roadCenters = [201.1 -99.2 0; 199.7 99.5 0];
road(scenario, roadCenters);

% Add the ego vehicle
egoVehicle = vehicle(scenario, 'ClassID', 1, 'Position', [65.1 -0.9 0]);
waypoints = [71 -0.5 0; 148.7 -0.5 0];
speed = 12;
trajectory(egoVehicle, waypoints, speed);

% Add the second vehicle
secondVehicle = vehicle(scenario, 'ClassID', 1, 'Position', [55.1 -0.9 0]);
waypoints = [61 -0.5 0; 138.7 -0.5 0];
speed = 12;
trajectory(secondVehicle, waypoints, speed);

% Add the parked cars
vehicle(scenario, 'ClassID', 1, 'Position', [111.0 -3.6 0]);
vehicle(scenario, 'ClassID', 1, 'Position', [140.6 -3.6 0]);
vehicle(scenario, 'ClassID', 1, 'Position', [182.6 -3.6 0]);
vehicle(scenario, 'ClassID', 1, 'Position', [211.3 -4.1 0]);

% Add pedestrian
actor(scenario, 'ClassID', 4, 'Length', 0.5, 'Width', 0.5, ...
    'Height', 1.7, 'Position', [130.3 -2.7 0], 'RCSPattern', [-8 -8;-8 -8]);

% Add parked truck
vehicle(scenario, 'ClassID', 2, 'Length', 8.2, 'Width', 2.5, ...
    'Height', 3.5, 'Position', [117.5 -3.5 0]);
end
```

createSensors

```
function [sensors, numSensors, attachedVehicle] = createSensors(scenario)
% createSensors Returns all sensor objects to generate detections
% Units used in createSensors and createDrivingScenario
% Distance/Position - meters
% Speed             - meters/second
% Angles            - degrees
% RCS Pattern       - dBsm
```

5-433

```
% Assign into each sensor the physical and radar profiles for all actors
profiles = actorProfiles(scenario);
sensors{1} = radarDetectionGenerator('SensorIndex', 1, ...
    'SensorLocation', [3.7 0], 'MaxRange', 50, 'FieldOfView', [60 5], ...
    'ActorProfiles', profiles, 'HasOcclusion', true, 'HasFalseAlarms', false);
sensors{2} = visionDetectionGenerator('SensorIndex', 2, ...
    'MaxRange', 100, 'SensorLocation', [1.9 0], 'DetectorOutput', 'Objects only', ...
    'ActorProfiles', profiles);
sensors{3} = radarDetectionGenerator('SensorIndex', 3, ...
    'SensorLocation', [3.7 0], 'MaxRange', 120, 'FieldOfView', [30 5], ...
    'ActorProfiles', profiles, 'HasOcclusion', true, 'HasFalseAlarms', false);
attachedVehicle = [1;1;2];
numSensors = numel(sensors);
end
```

scenarioToEgo

```
function trackInEgo = scenarioToEgo(trackInScenario)
% Performs coordinate transformation from scenario to ego coordinates
% trackInScenario has StateParameters defined to transform it from scenario
% coordinates to ego coordinates
% We assume a constant velocity model with state [x;vx;y;vy;z;vz]
egoPosInScenario = trackInScenario.StateParameters.Position;
egoVelInScenario = trackInScenario.StateParameters.Velocity;
stateInScenario = trackInScenario.State;
stateShift = [egoPosInScenario(1);egoVelInScenario(1);egoPosInScenario(2);egoVelInScenario(2);ego
stateInEgo = stateInScenario - stateShift;
trackInEgo = objectTrack('UpdateTime',trackInScenario.UpdateTime,'State',stateInEgo,'StateCovaria
end
```

egoToScenario

```
function trackInScenario = egoToScenario(trackInEgo)
% Performs coordinate transformation from ego to scenario coordinates
% trackInEgo has StateParameters defined to transform it from ego
% coordinates to scenario coordinates
% We assume a constant velocity model with state [x;vx;y;vy;z;vz]
egoPosInScenario = trackInEgo.StateParameters.Position;
egoVelInScenario = trackInEgo.StateParameters.Velocity;
stateInScenario = trackInEgo.State;
stateShift = [egoPosInScenario(1);egoVelInScenario(1);egoPosInScenario(2);egoVelInScenario(2);ego
stateInEgo = stateInScenario + stateShift;
trackInScenario = objectTrack('UpdateTime',trackInEgo.UpdateTime,'State',stateInEgo,'StateCovaria
end
```

detectAndTrack

```
function [tracks,wasTrackerUpdated] = detectAndTrack(agent,time,posSelector)
% Create detections from the vehicle
poses = targetPoses(agent.Actor);
[detections,isValid] = vehicleDetections(agent.Actor.Position,agent.Sensors,poses,time,agent.DetI

% Update tracks for the vehicle
if isValid
    agent.Tracker.StateParameters = struct(...
        'Frame','Rectangular', ...
        'Position', agent.Actor.Position, ...
        'Velocity', agent.Actor.Velocity);
    tracks = agent.Tracker(detections,time);
```

```matlab
        tracksInScenario = tracks;
        for i = 1:numel(tracks)
            tracksInScenario(i) = egoToScenario(tracks(i));
        end
        pos = getTrackPositions(tracksInScenario,posSelector);
        plotTrack(agent.TrkPlotter,pos)
        wasTrackerUpdated = true;
    else
        tracks = objectTrack.empty(0,1);
        wasTrackerUpdated = false;
    end
end

function [objectDetections,isValid] = vehicleDetections(position, sensors, poses, time, plotter)
% Provides the detections for each vehicle.

numSensors = numel(sensors);
objectDetections = {};
isValidTime      = false(1, numSensors);

% Generate detections for each sensor
for sensorIndex = 1:numSensors
    sensor = sensors{sensorIndex};
    [objectDets, ~, isValidTime(sensorIndex)] = sensor(poses, time);
    objectDets = cellfun(@(d) setAtt(d), objectDets, 'UniformOutput', false);

    if isa(sensors{sensorIndex},'radarDetectionGenerator')
        objectDets = helperClusterDetections(objectDets, 5);
    end
    numObjects = numel(objectDets);
    objectDetections = [objectDetections; objectDets(1:numObjects)]; %#ok<AGROW>
end
isValid = any(isValidTime);

% Plot detections
if numel(objectDetections)>0
    detPos = cellfun(@(d)d.Measurement(1:2), objectDetections, 'UniformOutput', false);
    detPos = cell2mat(detPos')' + position(1:2);
    plotDetection(plotter, detPos);
end
end

function d = setAtt(d)
% Set the attributes to be struct
d.ObjectAttributes = struct;
end

function detectionClusters = helperClusterDetections(detections, vehicleSize)
% helperClusterDetections  Helper to cluster detections in the example
N = numel(detections);
distances = zeros(N);
for i = 1:N
    for j = i+1:N
        if detections{i}.SensorIndex == detections{j}.SensorIndex
            distances(i,j) = norm(detections{i}.Measurement(1:2) - detections{j}.Measurement(1:2)
        else
            distances(i,j) = inf;
        end
    end
```

```
        end
    end
    leftToCheck = 1:N;
    i = 0;
    detectionClusters = cell(N,1);
    while ~isempty(leftToCheck)
        % Remove the detections that are in the same cluster as the one under
        % consideration
        underConsideration = leftToCheck(1);
        clusterInds = (distances(underConsideration, leftToCheck) < vehicleSize);
        detInds = leftToCheck(clusterInds);
        clusterDets = [detections{detInds}];
        clusterMeas = [clusterDets.Measurement];
        meas = mean(clusterMeas, 2);
        i = i + 1;
        detectionClusters{i} = detections{detInds(1)};
        detectionClusters{i}.Measurement = meas;
        leftToCheck(clusterInds) = [];
    end
    detectionClusters(i+1:end) = [];

    % Since the detections are now for clusters, modify the noise to represent
    % that they are of the whole car
    for i = 1:numel(detectionClusters)
        measNoise = eye(6);
        measNoise(1:2,1:2) = vehicleSize^2 * eye(2);
        measNoise(4:5,4:5) = eye(2) * vehicleSize^2;
        detectionClusters{i}.MeasurementNoise = measNoise;
    end
end
```

Design and Simulate Tracking Scenario with Tracking Scenario Designer

This example shows how to use `trackingScenarioDesigner` with an existing session file. Using the application, you can add, modify, or remove platforms, monostatic radar sensors, and trajectories of all objects in the scenario. You can also export the scenario as a MATLAB script for further analysis.

Introduction

The goal of object tracking is to determine the state of an object in the presence of infrequent or uncertain measurements. In the case of multiple objects, tracking algorithms must also address the problem of data association. To test these algorithms, a *tracking scenario* can be used to synthetically generate realistic detections of objects in a three-dimensional scene.

This example models a scenario where a radar tower equipped with a monostatic radar sensor scans the sky. At a distance far from the sensor, two airplanes fly in close proximity other for a short duration of time. The close spacing of the aircraft trajectories and their distances from the radar tower challenge the radar's capability to properly resolve the two objects. This scenario is described in further detail in the Tracking Closely Spaced Targets Under Ambiguity example.

Launch Tracking Scenario Designer

The matfile `TSD_TrackingCloselySpacedTargets` was previously saved with a tracking scenario session. To launch the application and load the session file, use the command:

```
trackingScenarioDesigner('TSD_TrackingCloselySpacedTargets.mat')
```
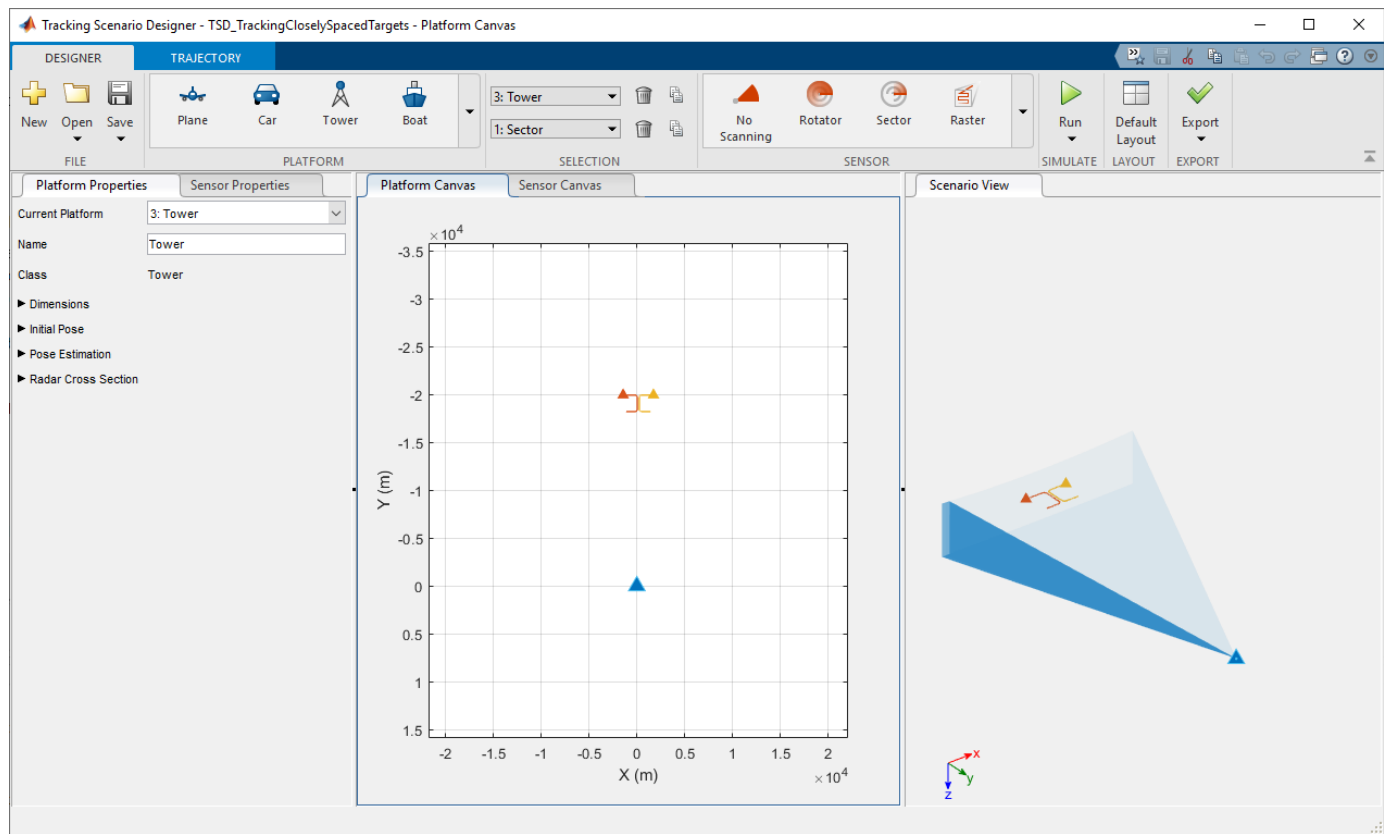
The application opens and loads the scenario. The Tracking Scenario Designer App consists of a toolstrip and three document groups:

- Platform Properties and Sensor Properties panels on the left
- Platform Canvas and Sensor Canvas in the center
- Scenario View showing a three-dimensional world view on the right

Edit and modify the scenario

Red plane

Select the red colored plane by clicking on it in the Platform Canvas.
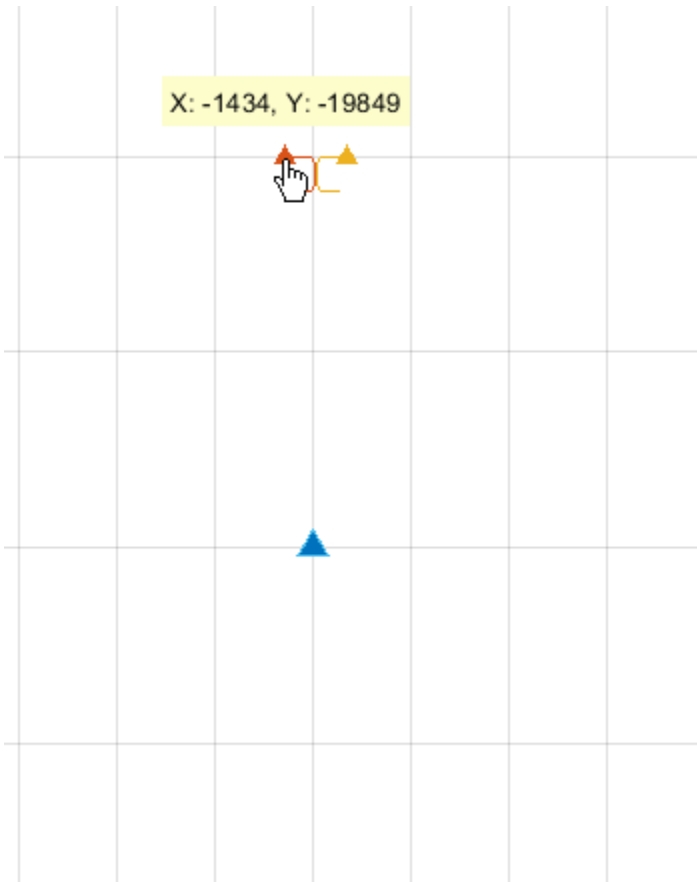
Inspect its trajectory using the Trajectory Table and the Time-Altitude plot
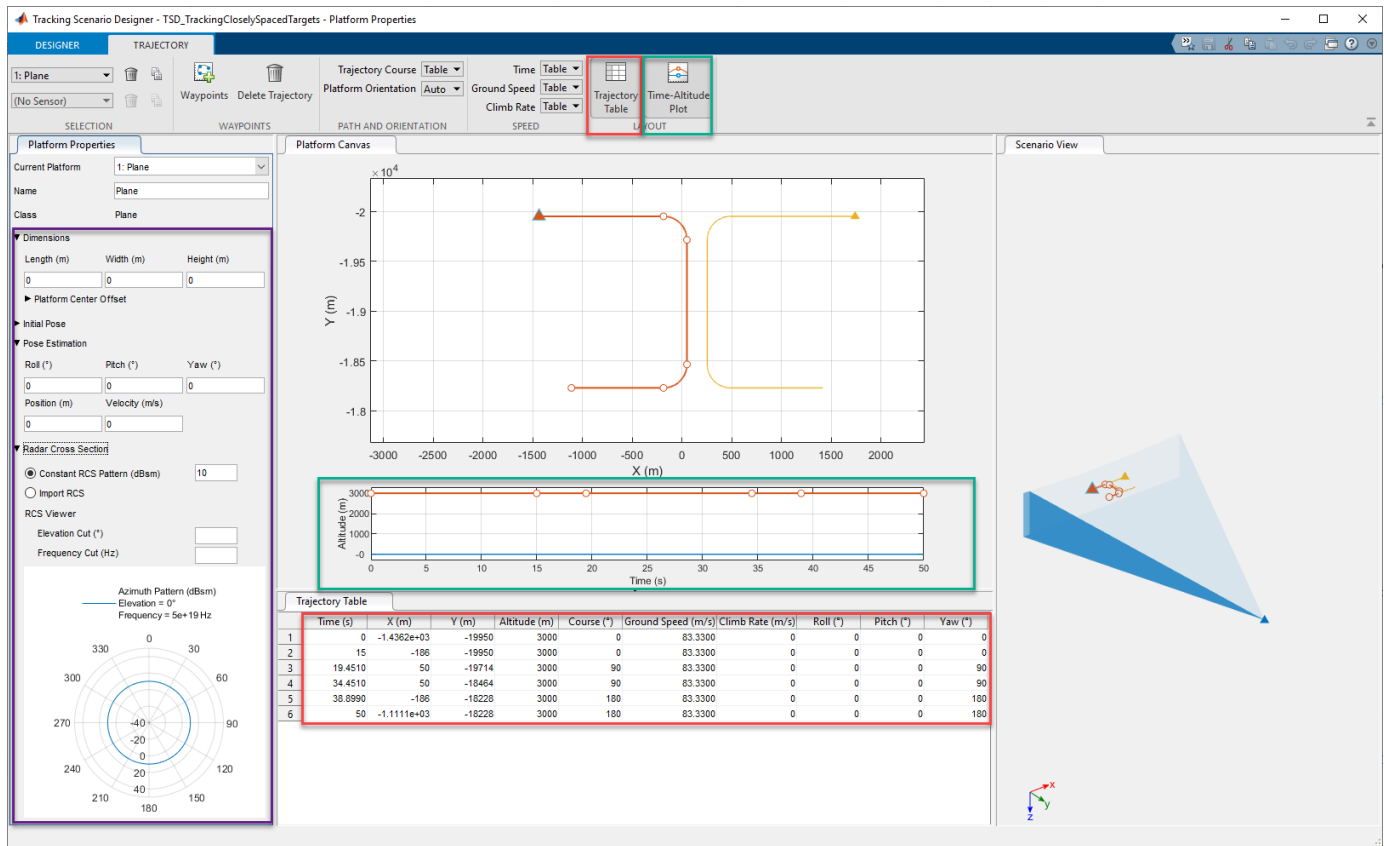
In the property panel (purple), note that the plane is modeled as a point object whose length, width, and height dimensions are zero. By default, its radar cross section is 10 dBsm.

Click on the Trajectory Table button (red) in the toolstrip to display the list of waypoints for this plane. The trajectory table appears at the bottom center of the application.

Click on the Time-Altitude Plot button (green) in the toolstrip to display the elevation profile of this plane. The plot appears within the same tab as the Platform Canvas.

Observe that the airplane is flying level at an altitude of 3000 m with a ground speed of 83.33 m/s. The duration of the trajectory is 50s.

You can further edit the trajectory by changing the parameter values in the table or by dragging waypoints in the Platform Canvas as shown in the GIF below.
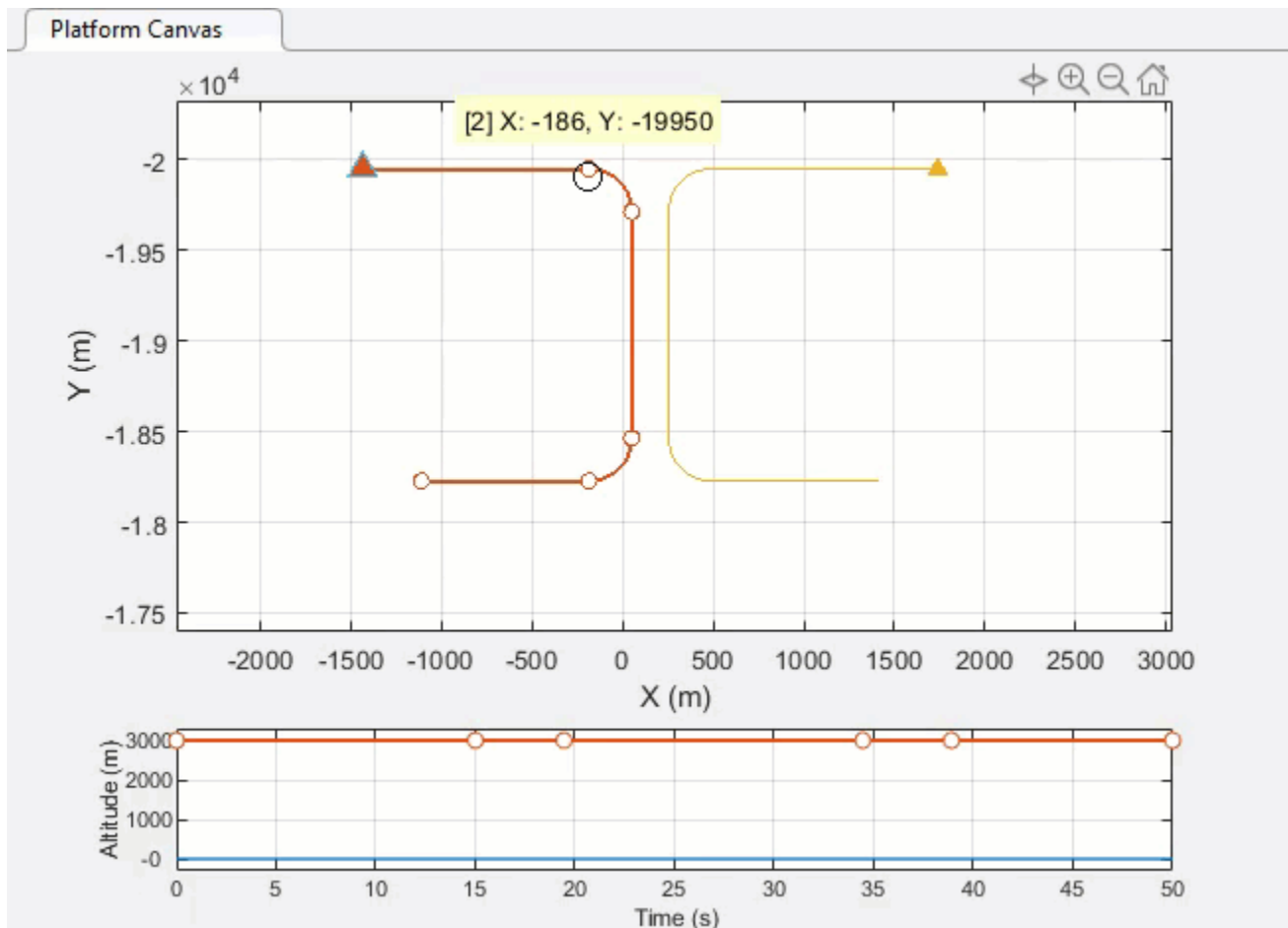
For the remainder of this example, we use the predefined trajectory in the session file.

Tower

Select the tower platform by clicking on the platform on the Platform Canvas or by choosing the tower platform from the Current Platform list in the Platform Properties tab.

Upon selecting the platform, the Sensor Properties and the Sensor Canvas for the tower platform become active.
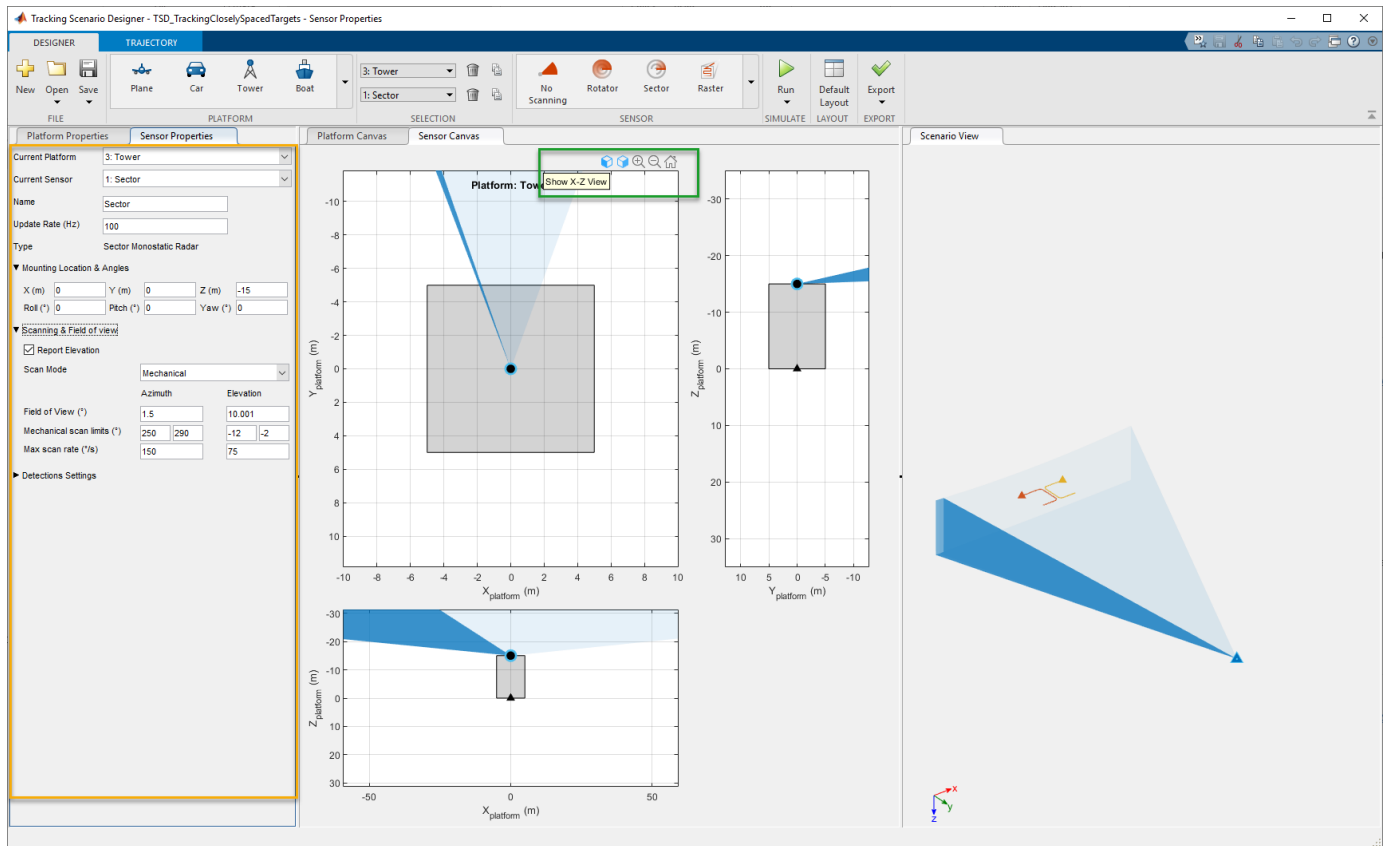
Observe the monostatic radar sensor properties in the panel (yellow). The radar has an update rate of 100 Hz, and a scan range of 40 degrees in azimuth and 10 degrees in elevation. Its field of view is 1.5 deg in azimuth and 10 deg in elevation. Note that we use 10.001 def to ensure the beam is not exactly the size of the elevation scan range, in which case the radar would sweep twice in elevation.

Note that the default platform frame is NED (North-East-Down). Also the Mounting Location of the sensor is defined as X = 0, Y = 0, Z = -15 to ensure that the radar is located at the top of the 15m high tower.

On the Sensor Canvas, you can use the toolbar (green) to display the side views of the tower. The axes are platform centric coordinates.

The sensor can be reconfigured using the property panel and its mounting location can be dragged on the Sensor Canvas for coarse editing.

Additionally, you can select a new sensor in the toolstrip gallery and mount it on the tower, or add a new platform in the scene using the platform gallery and then mount a new sensor on it. This is illustrated in the GIF below.
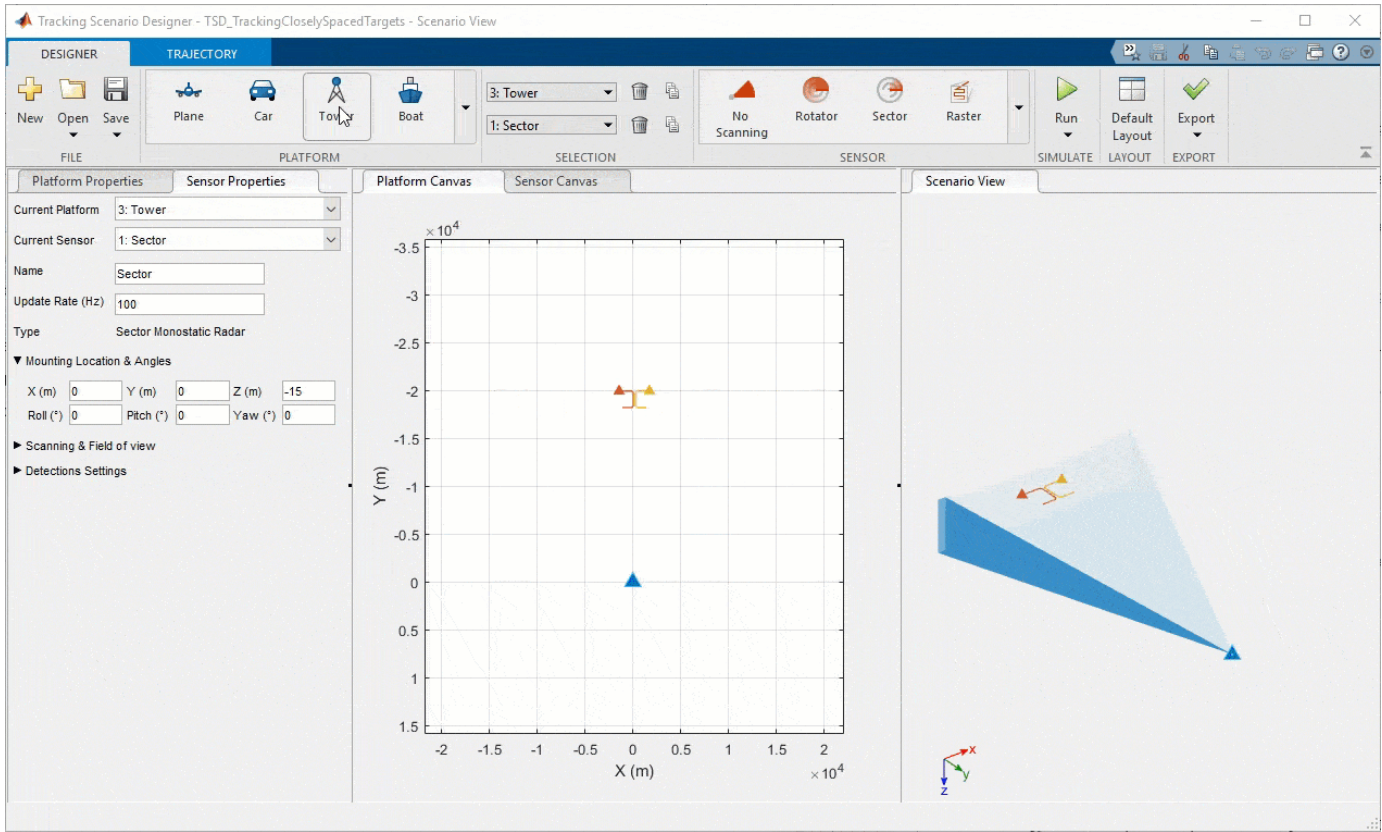
Simulate the scenario

As this scenario is already set up, the next step is to run the simulation to generate the synthetic radar detections.

Click **Run** to run the simulation. Alternatively, use the drop-down to select **Run Without Detections** to simulate the ground truth only.
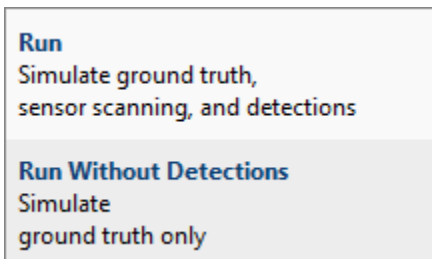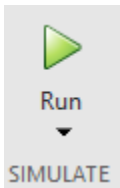
The application goes into simulation mode and the simulation begins automatically. You will see the two planes moving along their trajectories. When an aircraft is detected by the tower, you can see a dark purple marker show at the instant of detection.

To better observe the scene, you can use the axes toolbar to quickly navigate between X-Y, X-Z, or Y-Z view. Additionally, you can turn on and off the sensor coverage plot, the trajectory lines, the ground plane projection, and the orientation indicator.
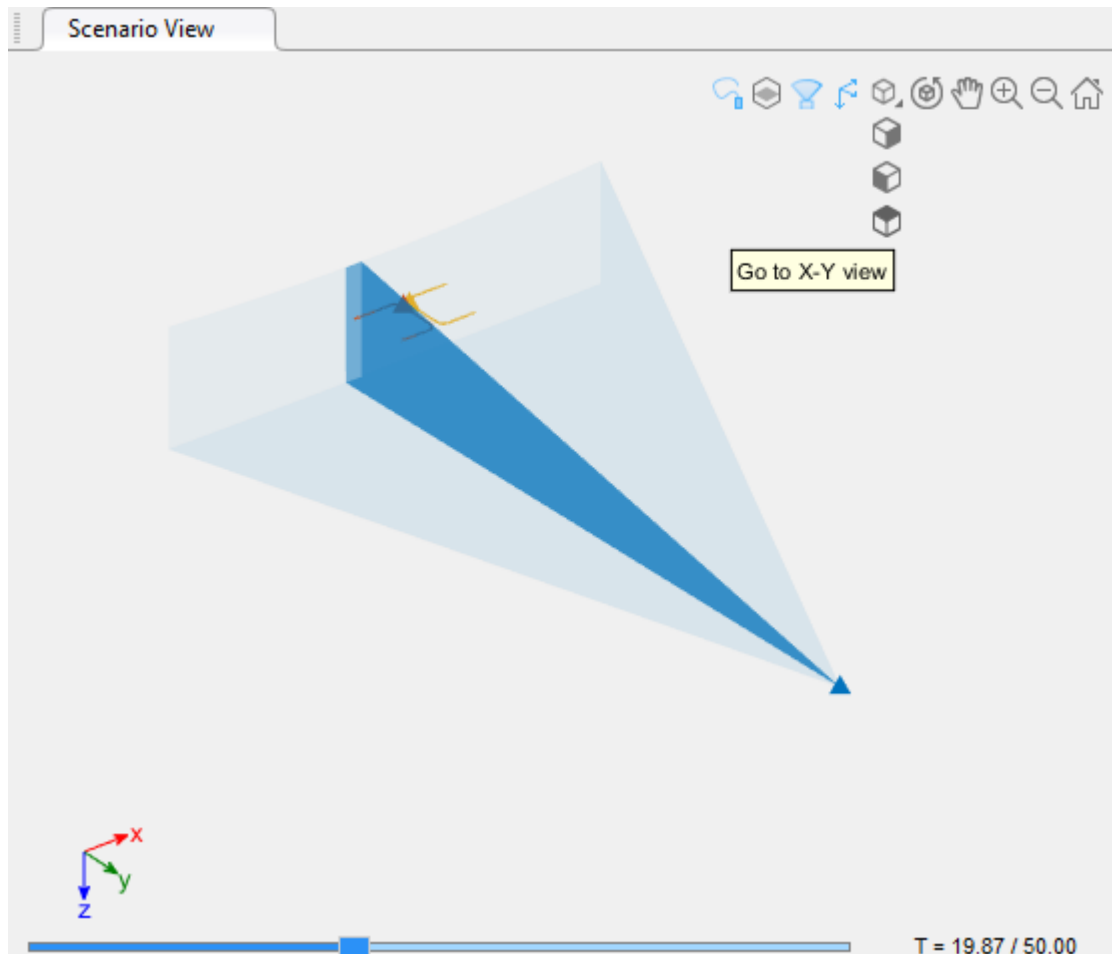
Playback Controls and Time Scroll Bar

To observe the detection creation as the beam scans a target, use the playback control which allow you to pause, step back, and step forward the simulation.
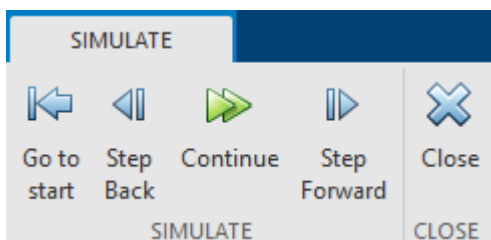
The time scroll bar is located at the bottom of the Scenario View. It represents the current status of the simulation as well as the current time of display.

The light blue progress bar shows progress in the data simulation of the scenario. The darker blue rectangle slider shows the progress of the animation of the scenario.

The rectangle slider can be dragged backward (left) or forward in time (right) within the time range that has been simulated.
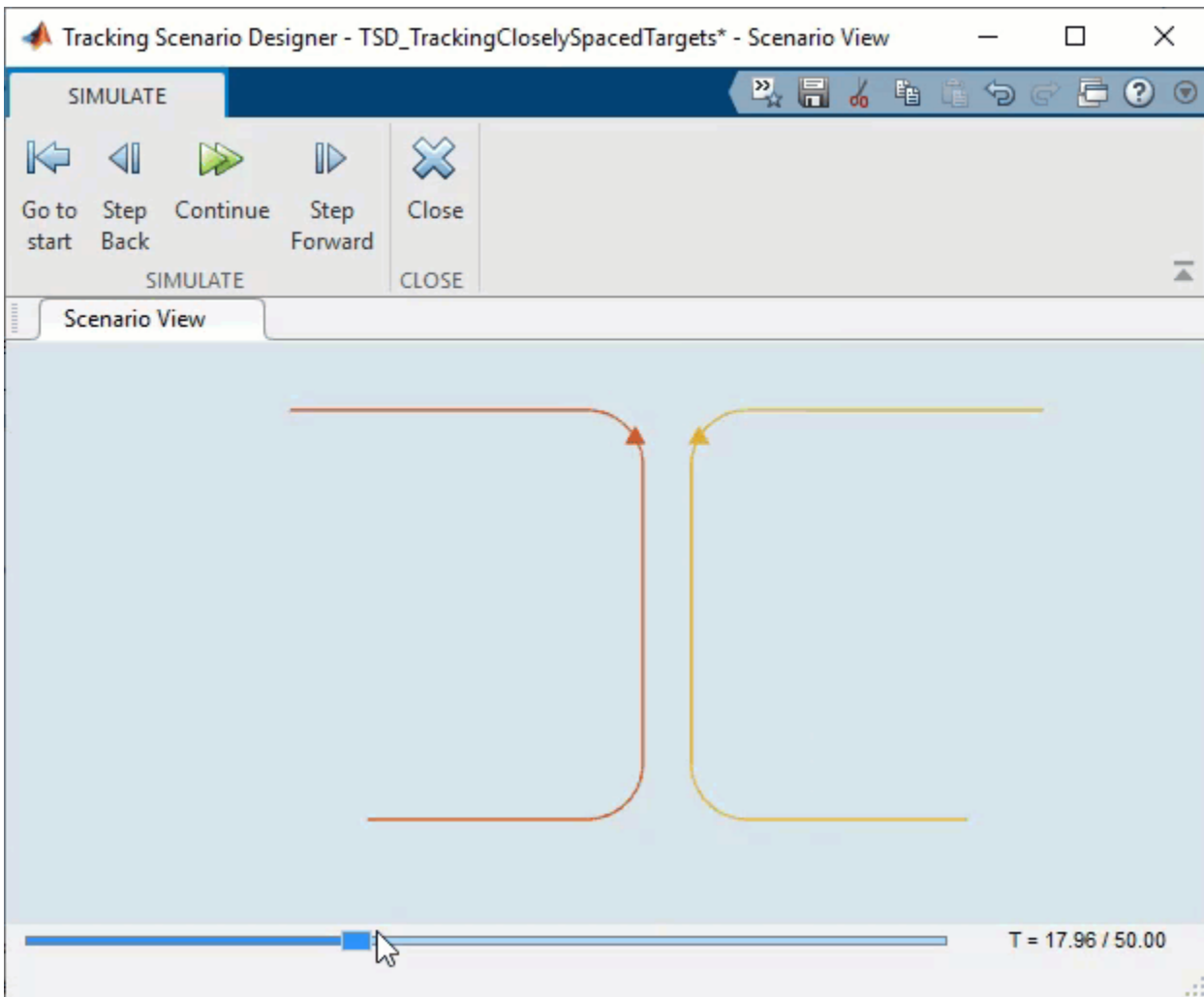
Step through the simulation, and observe that the radar generates a single detection per sweep between `t = 19.30 s` and `t = 20.50 s`. This is the region of ambiguity where the two planes cannot be resolved by the radar.
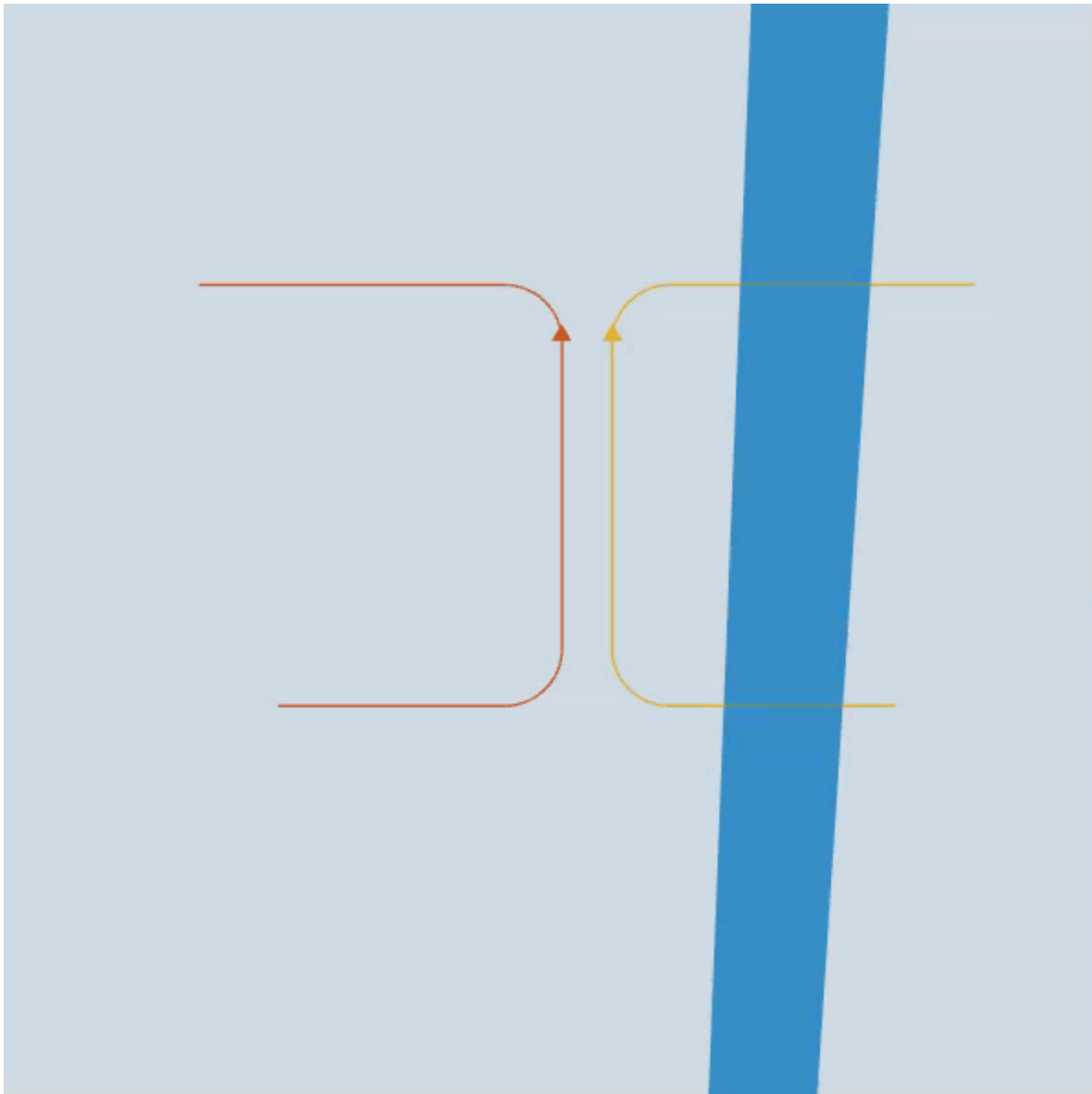
Export to MATLAB

To view and edit the code used to design the scenario, you can export the scenario as a MATLAB script. Use the script to add trackers and interact with the scenario programmatically.

Click **Export** to view the equivalent script that creates the scenario.
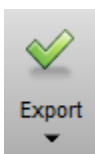
After clicking **Export**, you will see a script that can be saved to a file name of your choice. The current 3D view angles of the Scenario View are also exported to the script.

Run Generated Script

Visualize Targets and Detections

Once the script is saved, you can run it and observe the animated theater plot in MATLAB.
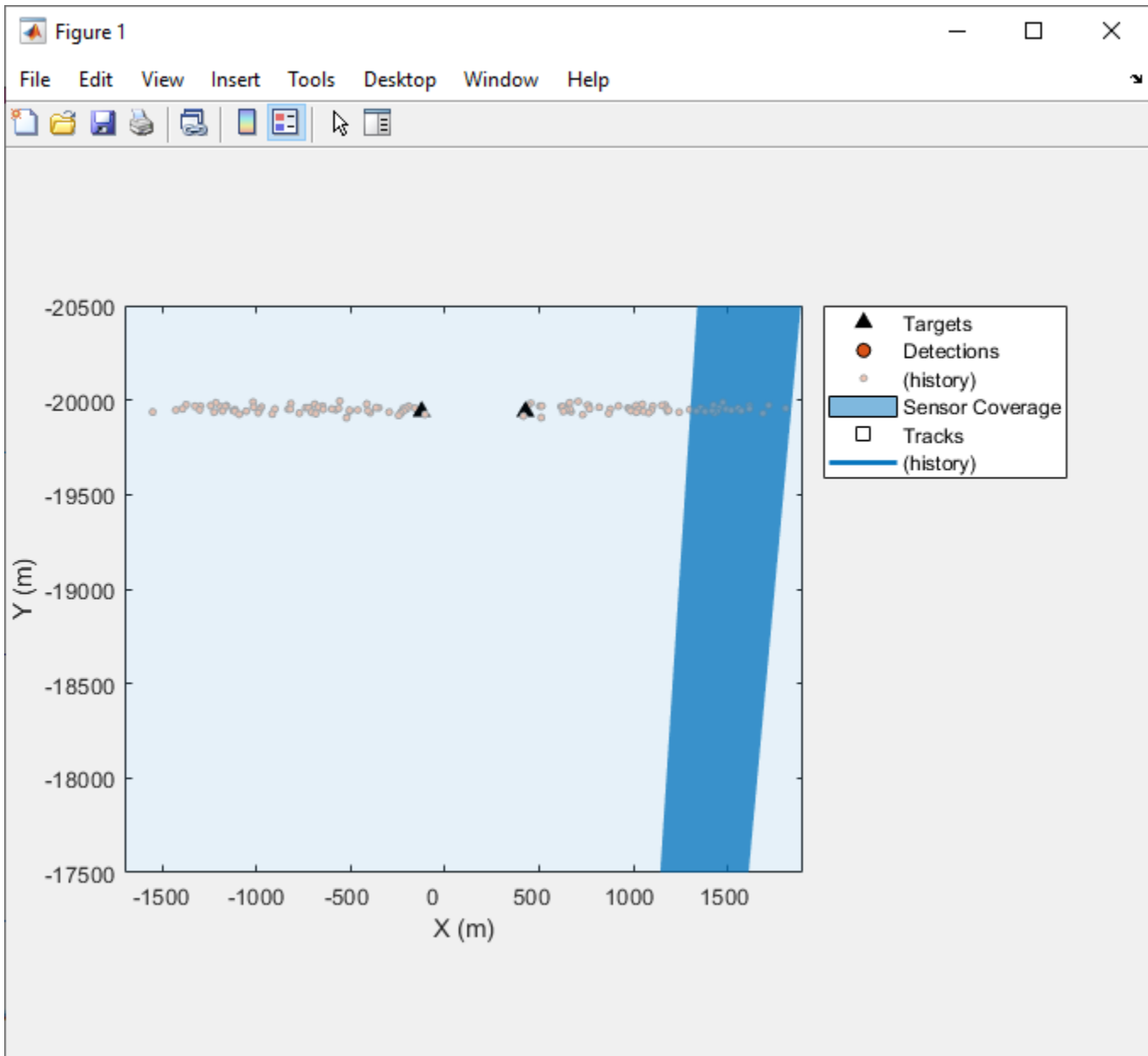
Edit the generated script for tracking

You can add the following commands to define a tracker as shown in the Tracking Closely Spaced Targets Under Ambiguity example.

Note: This code may need to be modified if you edit or use a different scenario.

The generated script from Tracking Scenario Designer has several comments that indicate where to add more code.

1) Configure a trackerJPDA

```matlab
% Configure your tracker here:
numTracks = 20;
gate = 45;
vol = 1e9;
beta = 1e-14;
pd = 0.8;
far = 1e-6;

tracker = trackerJPDA(...
    'FilterInitializationFcn',@initCVFilter,...
    'MaxNumTracks', numTracks, ...
    'MaxNumSensors', 1, ...
    'AssignmentThreshold',gate, ...
    'TrackLogic','Integrated',...
    'DetectionProbability', pd, ...
    'ClutterDensity', far/vol, ...
    'NewTargetDensity', beta,...
    'TimeTolerance',0.05);
```

2) Define a track plotter

```matlab
% Add a trackPlotter here:
trackp = trackPlotter(tp,'DisplayName','Tracks','ConnectHistory','on','ColorizeHistory','on');
```

3) Define a buffer for detections before the loop

```matlab
% Main simulation loop
detBuffer = {};
```

4) Update the tracker within the loop

```matlab
% Update your tracker here:
detBuffer = [detBuffer; dets]; %#ok<AGROW>
if configs.IsScanDone
    tracks = tracker(detBuffer,scenario.SimulationTime);
    pos = getTrackPositions(tracks,[1 0 0 0 0; 0 0 1 0 0 ; 0 0 0 0 1 0]);
    labels = string([tracks.TrackID]);
    detBuffer = {};
end
```

4) Update track plotter

```matlab
% Update the trackPlotter here:
if configs.IsScanDone
    trackp.plotTrack(pos,labels);
end
```

Visualize Tracks

Observe the tracking animation by rerunning the script with the above additions. You will now visualize the tracks of the two planes.
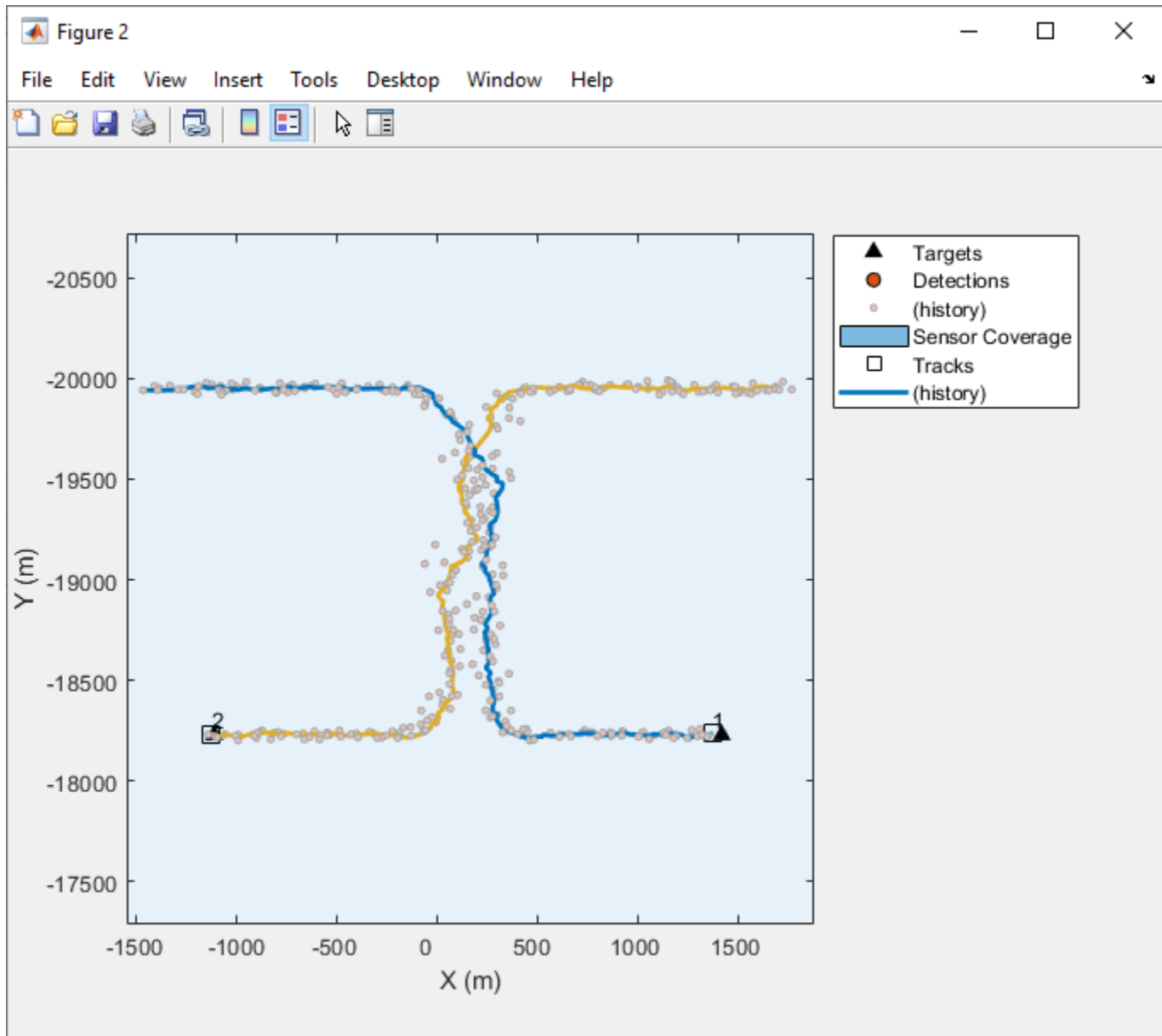
Summary

In this example, you used the Tracking Scenario Designer application to load a tracking scenario session file. You also learned how to navigate the application and how to simulate the scenario. In the scenario, two aircraft are detected by a single radar. For a portion of the time, the two aircraft are so close that the radar cannot resolve them. You learned how to export the scenario to a MATLAB script to rerun the simulation and how to modify the script to add a JPDA tracker.

Estimate Orientation with a Complementary Filter and IMU Data

This example shows how to stream IMU data from an Arduino and estimate orientation using a complementary filter.

Connect Hardware

Connect the SDA, SCL, GND, and VCC pins of the MPU-9250 sensor to the corresponding pins of the Arduino® hardware. This example uses an Arduino® Uno board with the following connections:

- SDA - A4
- SCL - A5
- VCC - +3.3V
- GND - GND
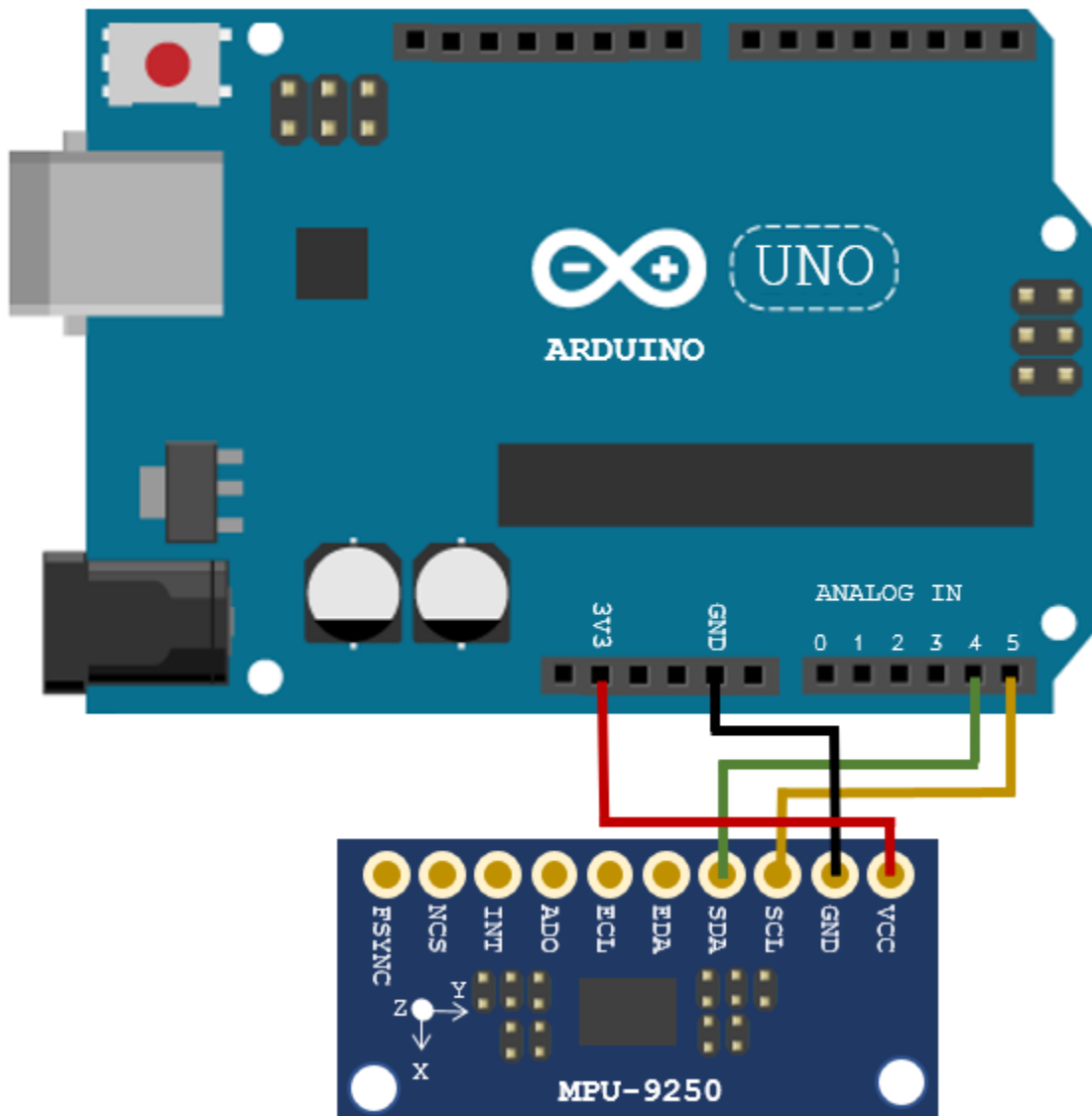
Ensure that the connections to the sensors are intact. It is recommended to attach/connect the sensor to a prototype shield to avoid loose connections while the sensor is in motion. Refer the Troubleshooting Sensors page to debug the sensor related issues.

Create Sensor Object

Create an `arduino` object and an `mpu9250` object. Specify the sensor sampling rate `Fs` and the amount of time to run the loops. Optionally, enable the `isVerbose` flag to check if any samples are overrun. By disabling the `useHW` flag, you can also run the example with sensor data saved in the MAT-file `loggedMPU9250Data.mat`.

The data in `loggedMPU9250Data.mat` was logged while the IMU was generally facing due South, then rotated:

- +90 degrees around the z-axis

5-451

- -180 degrees around the z-axis
- +90 degrees around the z-axis
- +90 degrees around the y-axis
- -180 degrees around the y-axis
- +90 degrees around the y-axis
- +90 degrees around the x-axis
- -270 degrees around the x-axis
- +180 degrees around the x-axis

Notice that the last two rotations around the x-axis are an additional 90 degrees. This was done to flip the device upside-down. The final orientation of the IMU is the same as the initial orientation, due South.

```
Fs = 100;
samplesPerRead = 10;
runTime = 20;
isVerbose = false;
useHW = true;

if useHW
    a = arduino;
    imu = mpu9250(a, 'SampleRate', Fs, 'OutputFormat', 'matrix', ...
        'SamplesPerRead', samplesPerRead, 'Streaming', true);
else
    load('loggedMPU9250Data.mat', 'allAccel', 'allGyro', 'allMag', ...
        'allT', 'allOverrun', ...
        'numSamplesAccelGyro', 'numSamplesAccelGyroMag')
end
```

Align Axes of MPU-9250 Sensor with NED Coordinates

The axes of the accelerometer, gyroscope, and magnetometer in the MPU-9250 are not aligned with each other. Specify the index and sign x-, y-, and z-axis of each sensor so that the sensor is aligned with the North-East-Down (NED) coordinate system when it is at rest. In this example, the magnetometer axes are changed while the accelerometer and gyroscope axes remain fixed. For your own applications, change the following parameters as necessary.

```
% Accelerometer axes parameters.
accelXAxisIndex = 1;
accelXAxisSign = 1;
accelYAxisIndex = 2;
accelYAxisSign = 1;
accelZAxisIndex = 3;
accelZAxisSign = 1;

% Gyroscope axes parameters.
gyroXAxisIndex = 1;
gyroXAxisSign = 1;
gyroYAxisIndex = 2;
gyroYAxisSign = 1;
gyroZAxisIndex = 3;
gyroZAxisSign = 1;

% Magnetometer axes parameters.
```

```
magXAxisIndex = 2;
magXAxisSign = 1;
magYAxisIndex = 1;
magYAxisSign = 1;
magZAxisIndex = 3;
magZAxisSign = -1;

% Helper functions used to align sensor data axes.

alignAccelAxes = @(in) [accelXAxisSign, accelYAxisSign, accelZAxisSign] ...
    .* in(:, [accelXAxisIndex, accelYAxisIndex, accelZAxisIndex]);

alignGyroAxes = @(in) [gyroXAxisSign, gyroYAxisSign, gyroZAxisSign] ...
    .* in(:, [gyroXAxisIndex, gyroYAxisIndex, gyroZAxisIndex]);

alignMagAxes = @(in) [magXAxisSign, magYAxisSign, magZAxisSign] ...
    .* in(:, [magXAxisIndex, magYAxisIndex, magZAxisIndex]);
```

Perform Additional Sensor Calibration

If necessary, you may calibrate the magnetometer to compensate for magnetic distortions. For more details, see the Compensating for Hard Iron Distortions section of the "Estimating Orientation Using Inertial Sensor Fusion and MPU-9250" on page 5-478 example.

Specify Complementary filter Parameters

The `complementaryFilter` has two tunable parameters. The `AccelerometerGain` parameter determines how much the accelerometer measurement is trusted over the gyroscope measurement. The `MagnetometerGain` parameter determines how much the magnetometer measurement is trusted over the gyroscope measurement.

```
compFilt = complementaryFilter('SampleRate', Fs)


compFilt =

  complementaryFilter with properties:

          SampleRate: 100
    AccelerometerGain: 0.0100
     MagnetometerGain: 0.0100
       HasMagnetometer: 1
    OrientationFormat: 'quaternion'
```

Estimate Orientation with Accelerometer and Gyroscope

Set the `HasMagnetometer` property to `false` to disable the magnetometer measurement input. In this mode, the filter only takes accelerometer and gyroscope measurements as inputs. Also, the filter assumes the initial orientation of the IMU is aligned with the parent navigation frame. If the IMU is not aligned with the navigation frame initially, there will be a constant offset in the orientation estimation.

```
compFilt = complementaryFilter('HasMagnetometer', false);

tuner = HelperOrientationFilterTuner(compFilt);

if useHW
```

```matlab
        tic
    else
        idx = 1:samplesPerRead;
        overrunIdx = 1;
    end
    while true
        if useHW
            [accel, gyro, mag, t, overrun] = imu();
            accel = alignAccelAxes(accel);
            gyro = alignGyroAxes(gyro);
        else
            accel = allAccel(idx,:);
            gyro = allGyro(idx,:);
            mag = allMag(idx,:);
            t = allT(idx,:);
            overrun = allOverrun(overrunIdx,:);

            idx = idx + samplesPerRead;
            overrunIdx = overrunIdx + 1;
            pause(samplesPerRead/Fs)
        end

        if (isVerbose && overrun > 0)
            fprintf('%d samples overrun ...\n', overrun);
        end

        q = compFilt(accel, gyro);
        update(tuner, q);

        if useHW
            if toc >= runTime
                break;
            end
        else
            if idx(end) > numSamplesAccelGyro
                break;
            end
        end
    end
```
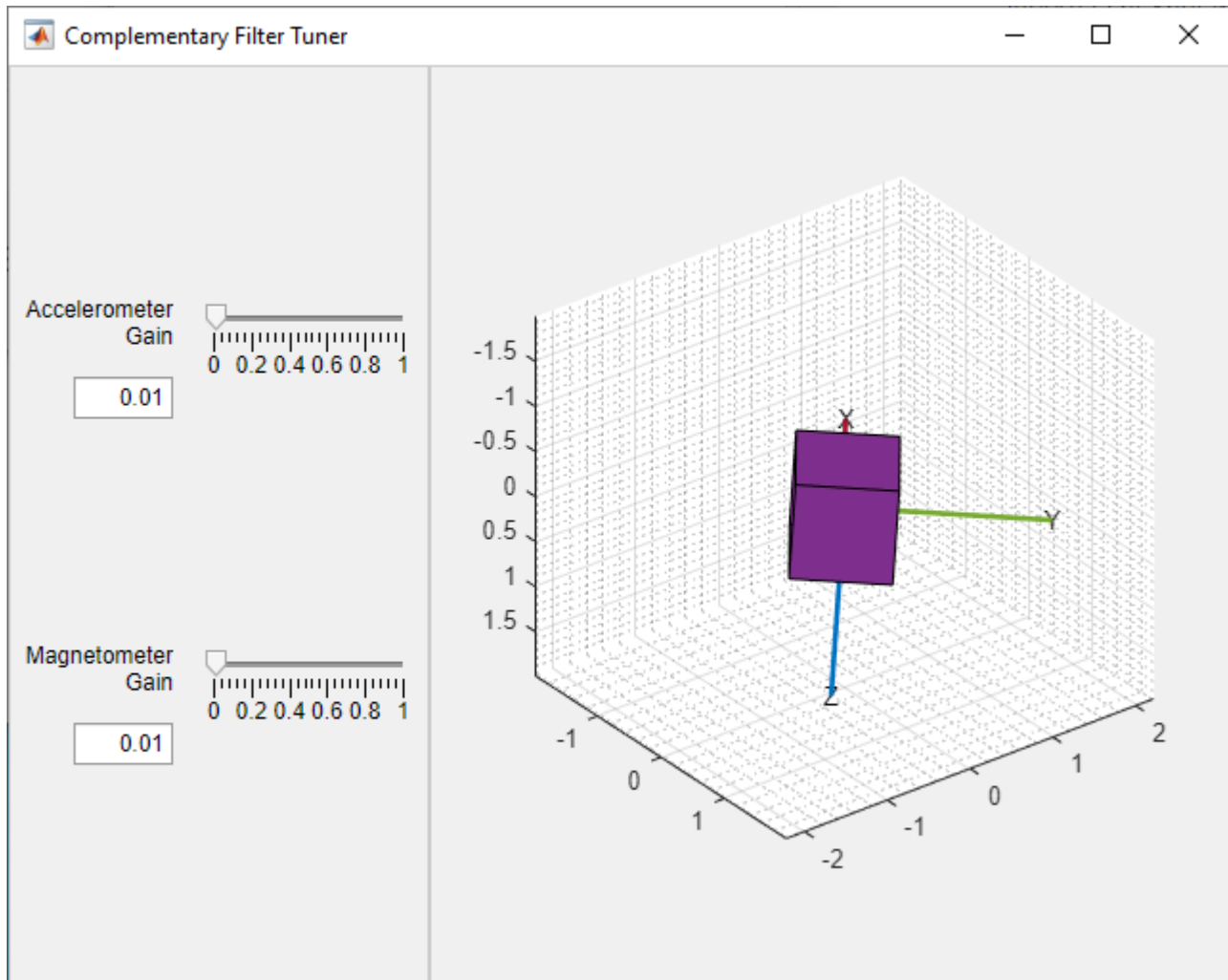
Estimate Orientation with Accelerometer, Gyroscope, and Magnetometer

With the default values of `AccelerometerGain` and `MagnetometerGain`, the filter trusts more on the gyroscope measurements in the short-term, but trusts more on the accelerometer and magnetometer measurements in the long-term. This allows the filter to be more reactive to quick orientation changes and prevents the orientation estimates from drifting over longer periods of time. For specific IMU sensors and application purposes, you may want to tune the parameters of the filter to improve the orientation estimation accuracy.

```
compFilt = complementaryFilter('SampleRate', Fs);

tuner = HelperOrientationFilterTuner(compFilt);

if useHW
    tic
end
while true
    if useHW
        [accel, gyro, mag, t, overrun] = imu();
        accel = alignAccelAxes(accel);
        gyro = alignGyroAxes(gyro);
```

```
            mag = alignMagAxes(mag);
        else
            accel = allAccel(idx,:);
            gyro = allGyro(idx,:);
            mag = allMag(idx,:);
            t = allT(idx,:);
            overrun = allOverrun(overrunIdx,:);

            idx = idx + samplesPerRead;
            overrunIdx = overrunIdx + 1;
            pause(samplesPerRead/Fs)
        end

        if (isVerbose && overrun > 0)
            fprintf('%d samples overrun ...\n', overrun);
        end

        q = compFilt(accel, gyro, mag);
        update(tuner, q);

        if useHW
            if toc >= runTime
                break;
            end
        else
            if idx(end) > numSamplesAccelGyroMag
                break;
            end
        end
    end
```
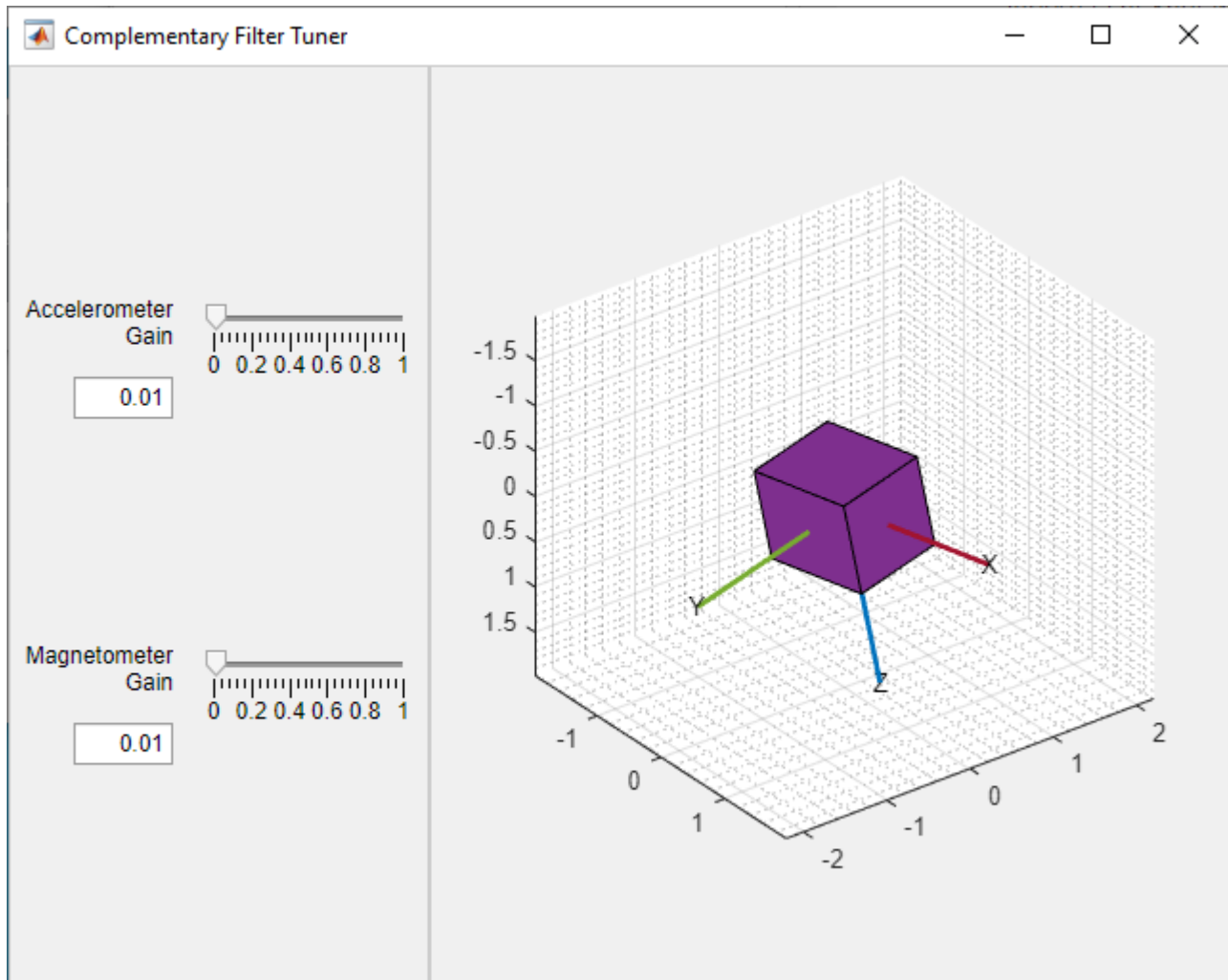
Summary

This example showed how to estimate the orientation of an IMU using data from an Arduino and a complementary filter. This example also showed how to configure the IMU and discussed the effects of tuning the complementary filter parameters.

Logged Sensor Data Alignment for Orientation Estimation

This example shows how to align and preprocess logged sensor data. This allows the fusion filters to perform orientation estimation as expected. The logged data was collected from an accelerometer and a gyroscope mounted on a ground vehicle.

Load Logged Sensor Data

Load logged inertial measurement unit (IMU) data and extract individual sensor data and timestamps.

```
load('imuData', 'imuTT')

time = imuTT.Time;
accel = imuTT.LinearAcceleration;
gyro = imuTT.AngularVelocity;
orient = imuTT.Orientation;
```

Inspect the Gyroscope Data

From the range of angular velocity readings, the logged gyroscope data is in radians per second instead of degrees per second. Also, the larger z-axis values and small x- and y-axis values indicate that the device rotated around the z-axis only.

```
figure
plot(time, gyro)
title('Gyroscope')
ylabel('rad/s')
legend('x-axis', 'y-axis', 'z-axis')
```
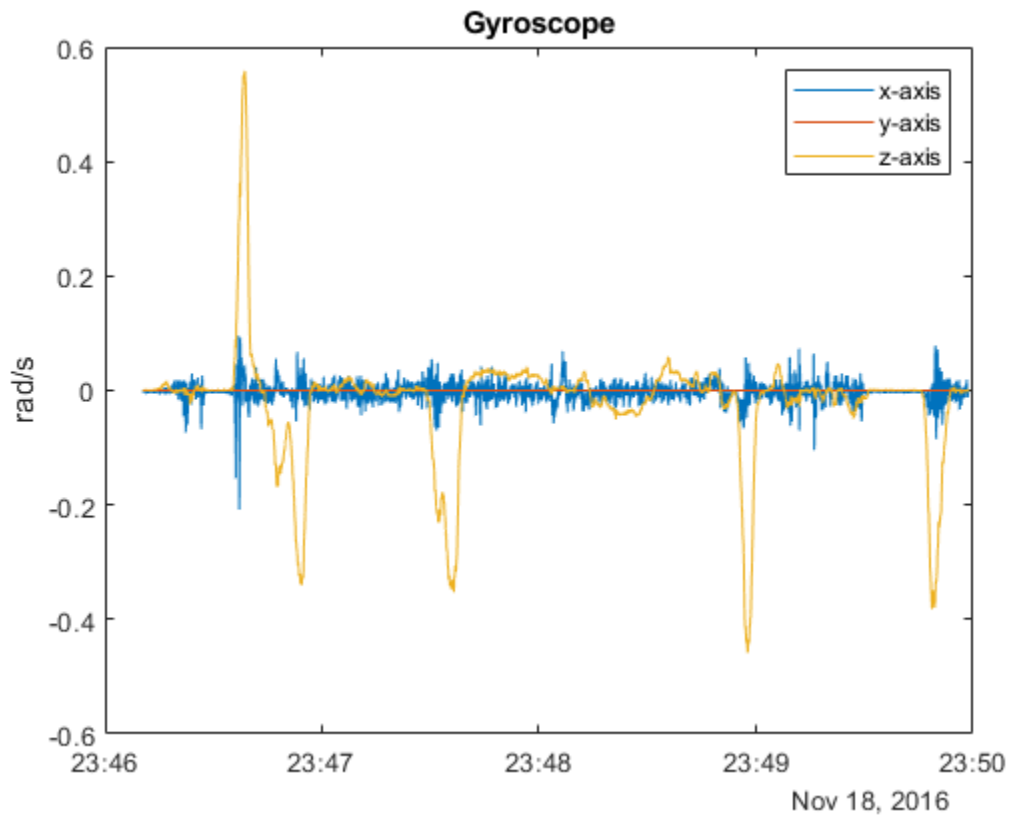
Inspect the Accelerometer Data

Since the z-axis reading of the accelerometer is around 10, the logged data is in meters per second squared instead of g's.

```
figure
plot(time, accel)
title('Accelerometer')
ylabel('m/s^2')
legend('x-axis', 'y-axis', 'z-axis')
```
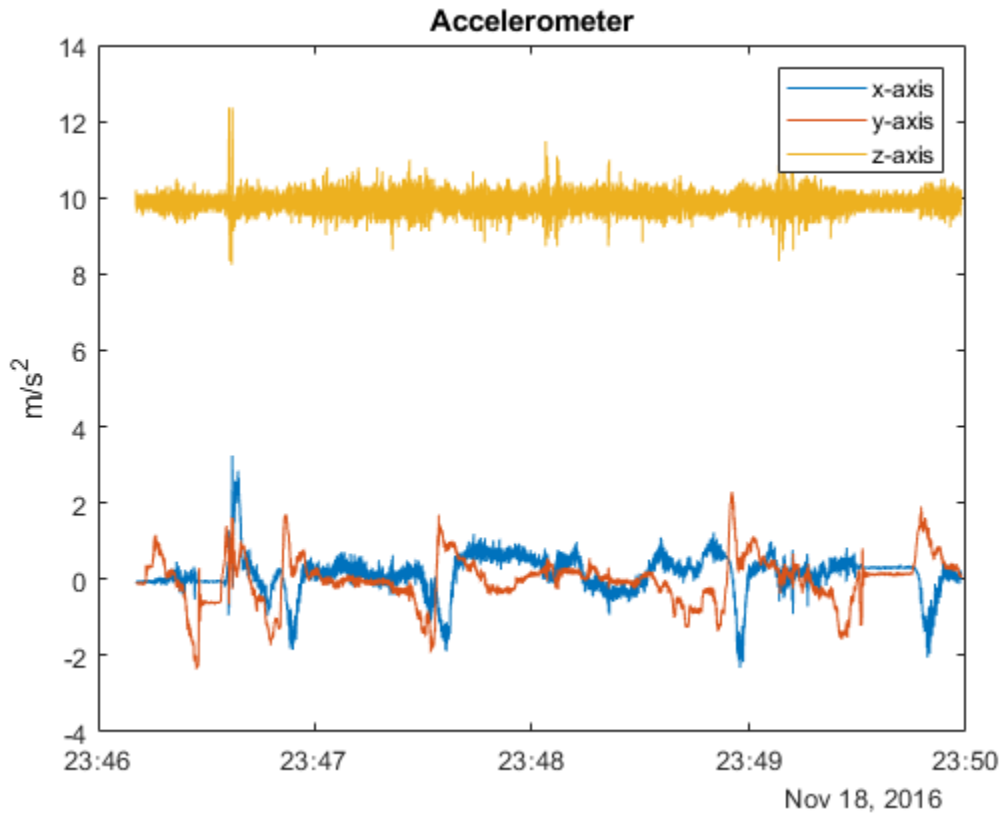
Inspect the Orientation Data

Convert the logged orientation quaternion data to Euler angles in degrees. The z-axis is changing while the x- and y-axis are relatively fixed. This matches the gyroscope and accelerometer readings. Therefore, no axis negating or rotating is required. However, the z-axis Euler angle is decreasing while the gyroscope reading is positive. This means that the logged orientation quaternion is expected to be applied as a point rotation operator ($v' = qvq^*$). In order to have the orientation quaternion match the orientations filters, such as `imufilter`, the quaternion needs to be applied as a frame rotation operator ($v' = q^*vq$). This can be done by conjugating the logged orientation quaternion.

```
figure
plot(time, eulerd(orient, 'ZYX', 'frame'))
title('Euler Angles')
ylabel('\circ') % Degrees symbol.
legend('z-axis', 'y-axis', 'x-axis')
```
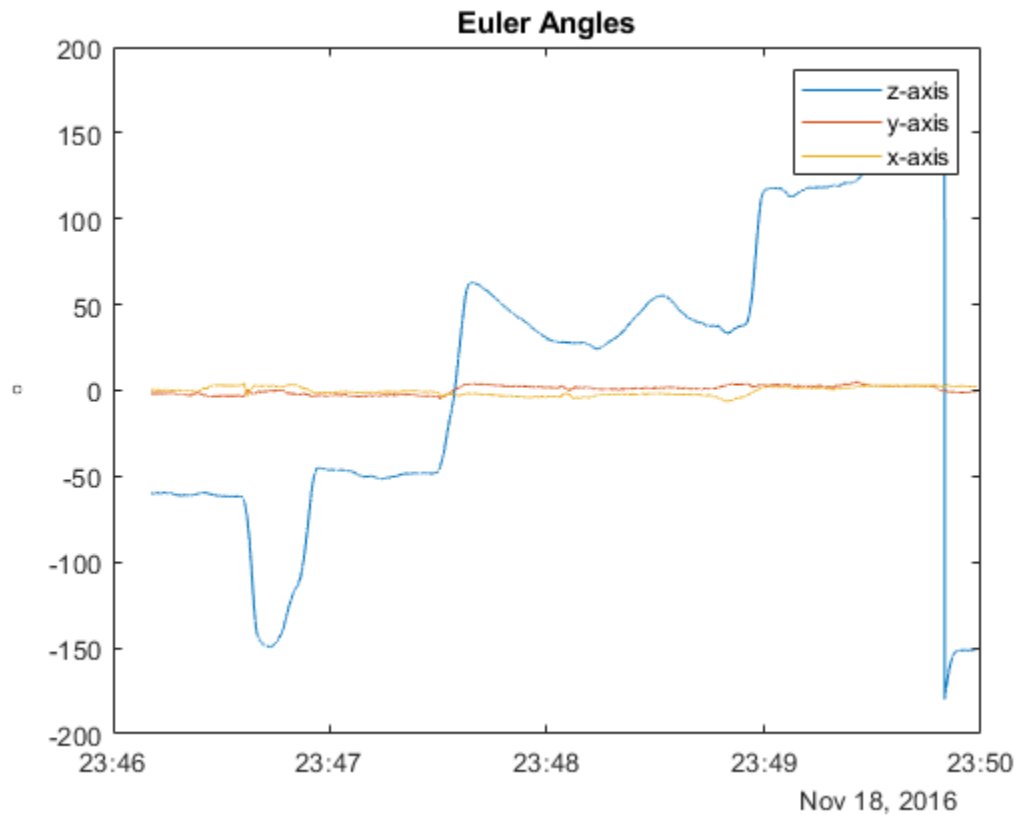
Find the Sampling Rate of the Logged Data

An estimate of the sampling rate can be obtained by taking the mean of the difference between the timestamps. Notice that there are some variances in the time differences. Since the variances are small for this logged data, the mean of the time differences can be used. Alternatively, the sensor data could be interpolated using the timestamps and equally spaced timestamps as query points.

```
deltaTimes = seconds(diff(time));
sampleRate = 1/mean(deltaTimes);

figure
plot([deltaTimes, repmat(mean(deltaTimes), numel(deltaTimes), 1)])
title('Time Differences')
ylabel('s')
legend('differences', 'mean')
```
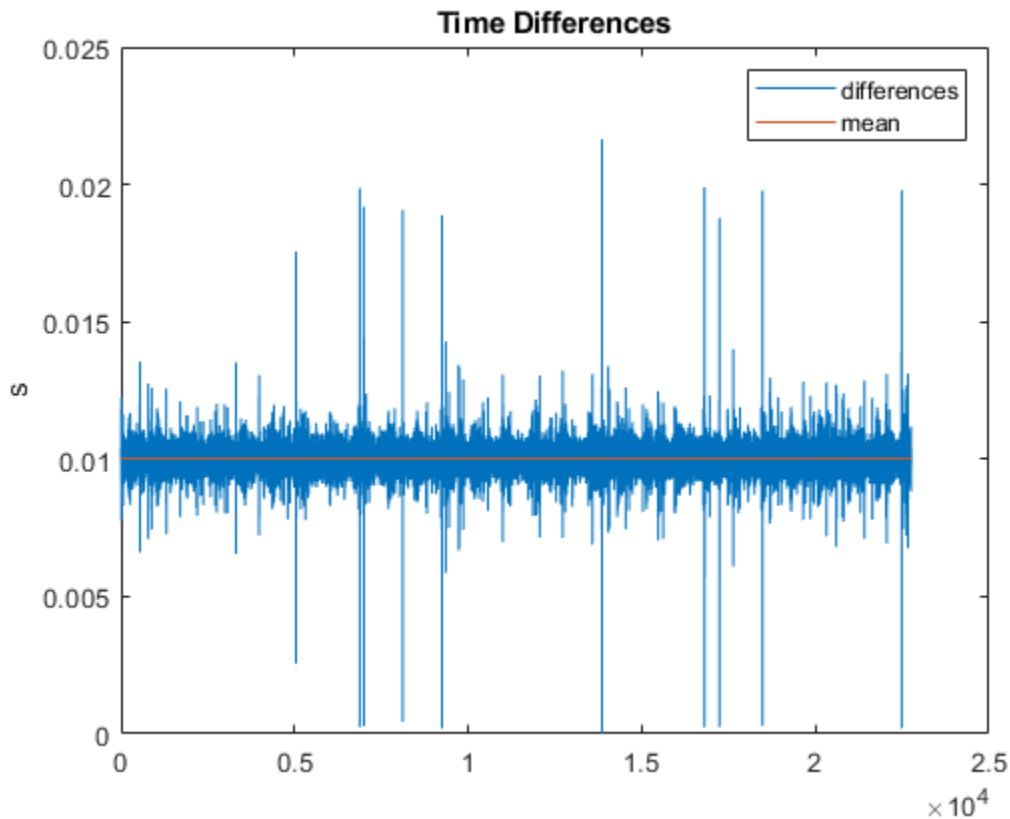
Compare the Transformed Logged Quaternion to the `imufilter` Quaternion

Conjugate the logged orientation quaternion before comparing it to the estimated orientation quaternion from `imufilter`. From the plot below, there is still a constant offset in the z-axis Euler angle estimate. This is because the `imufilter` assumes the initial orientation of the device is aligned with the navigation frame.

```
loggedOrient = conj(orient);

filt = imufilter('SampleRate', sampleRate);
estOrient = filt(accel, gyro);


figure
subplot(2, 1, 1)
plot(time, eulerd(loggedOrient, 'ZYX', 'frame'), '--')
title('Logged Euler Angles')
ylabel('\circ') % Degrees symbol.
legend('z-axis', 'y-axis', 'x-axis')
subplot(2, 1, 2)
plot(time, eulerd(estOrient, 'ZYX', 'frame'))
title('|imufilter| Euler Angles')
ylabel('\circ') % Degrees symbol.
legend('z-axis', 'y-axis', 'x-axis')
```
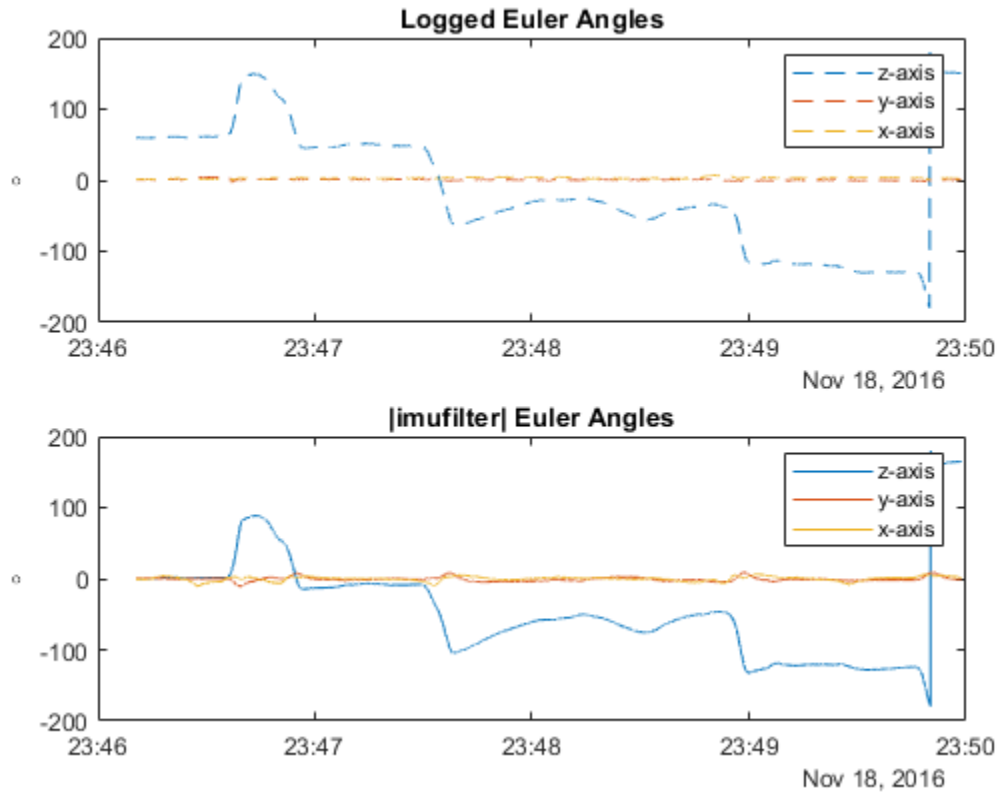
Align the Logged Orientation and `imufilter` Orientation

Align the `imufilter` orientation quaternion with the logged orientation quaternion by applying a constant bias using the first logged orientation quaternion. For quaternions, a constant rotation bias can be applied by pre-multiplying frame rotations or post-multiplying point rotations. Since `imufilter` reports quaternions as frame rotation operators, the estimated orientation quaternions are pre-multiplied by the first logged orientation quaternion.

```
alignedEstOrient = loggedOrient(1) .* estOrient;

figure
subplot(2, 1, 1)
plot(time, eulerd(loggedOrient, 'ZYX', 'frame'), '--')
title('Logged Euler Angles')
ylabel('\circ') % Degrees symbol.
legend('z-axis', 'y-axis', 'x-axis')
subplot(2, 1, 2)
plot(time, eulerd(alignedEstOrient, 'ZYX', 'frame'))
title('Aligned |imufilter| Euler Angles')
ylabel('\circ') % Degrees symbol.
legend('z-axis', 'y-axis', 'x-axis')
```
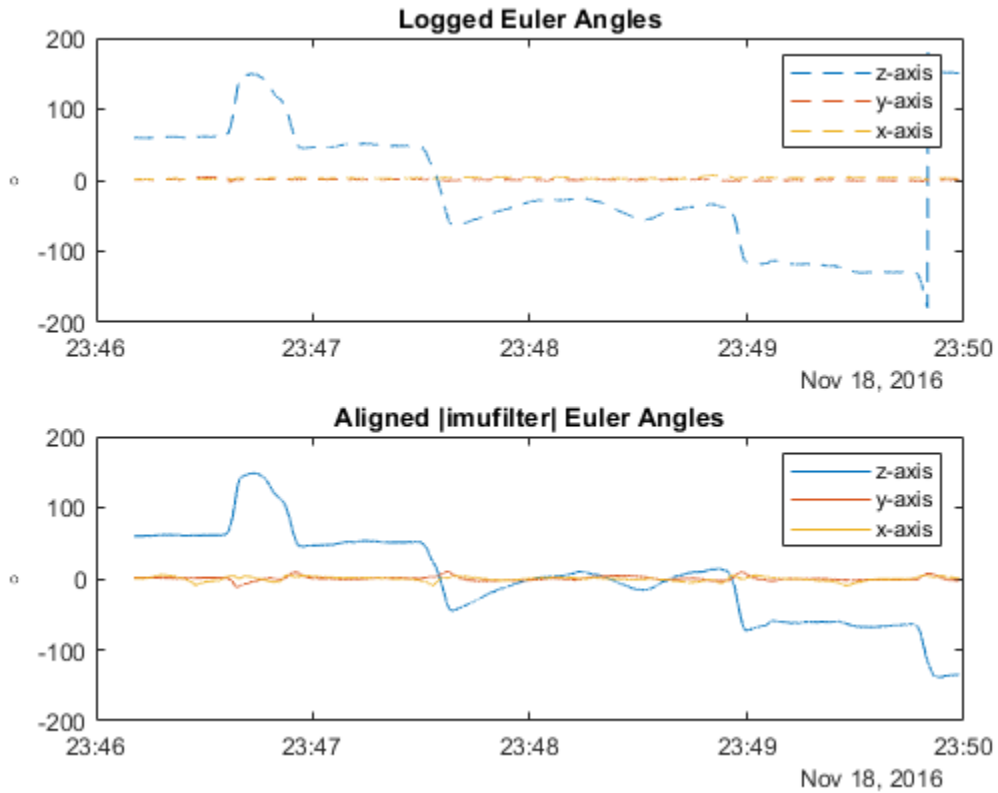
Conclusion

For the MAT-file in this example, you checked the following aspects for alignment:

- Units for accelerometer and gyroscope.
- Axes alignments of accelerometer and gyroscope.
- Orientation quaternion rotation operator (point: $v' = qvq^*$ or frame: $v' = q^*vq$)

Different unit conversions, axes alignments, and quaternion transformations may need to be applied depending on the format of the logged data.

Remove Bias from Angular Velocity Measurement

This example shows how to remove gyroscope bias from an IMU using `imufilter`.

Use `kinematicTrajectory` to create a trajectory with two parts. The first part has a constant angular velocity about the *y*- and *z*-axes. The second part has a varying angular velocity in all three axes.

```
duration = 60*8;
fs = 20;
numSamples = duration * fs;
rng('default') % Seed the RNG to reproduce noisy sensor measurements.

initialAngVel = [0,0.5,0.25];
finalAngVel = [-0.2,0.6,0.5];
constantAngVel = repmat(initialAngVel,floor(numSamples/2),1);
varyingAngVel = [linspace(initialAngVel(1), finalAngVel(1), ceil(numSamples/2)).', ...
    linspace(initialAngVel(2), finalAngVel(2), ceil(numSamples/2)).', ...
    linspace(initialAngVel(3), finalAngVel(3), ceil(numSamples/2)).'];

angVelBody = [constantAngVel; varyingAngVel];
accBody = zeros(numSamples,3);

traj = kinematicTrajectory('SampleRate',fs);

[~,qNED,~,accNED,angVelNED] = traj(accBody,angVelBody);
```

Create an `imuSensor` System object™, `IMU`, with a nonideal gyroscope. Call `IMU` with the ground-truth acceleration, angular velocity, and orientation.

```
IMU = imuSensor('accel-gyro', ...
    'Gyroscope',gyroparams('RandomWalk',0.003,'ConstantBias',0.3), ...
    'SampleRate',fs);

[accelReadings, gyroReadingsBody] = IMU(accNED,angVelNED,qNED);
```

Create an `imufilter` System object, `fuse`. Call `fuse` with the modeled accelerometer readings and gyroscope readings.

```
fuse = imufilter('SampleRate',fs, 'GyroscopeDriftNoise', 1e-6);

[~,angVelBodyRecovered] = fuse(accelReadings,gyroReadingsBody);
```

Plot the ground-truth angular velocity, the gyroscope readings, and the recovered angular velocity for each axis.

The angular velocity returned from the `imufilter` compensates for the effect of the gyroscope bias over time and converges to the true angular velocity.

```
time = (0:numSamples-1)'/fs;

figure(1)
plot(time,angVelBody(:,1), ...
    time,gyroReadingsBody(:,1), ...
    time,angVelBodyRecovered(:,1))
title('X-axis')
legend('True Angular Velocity', ...
```

```
        'Gyroscope Readings', ...
        'Recovered Angular Velocity')
ylabel('Angular Velocity (rad/s)')
```
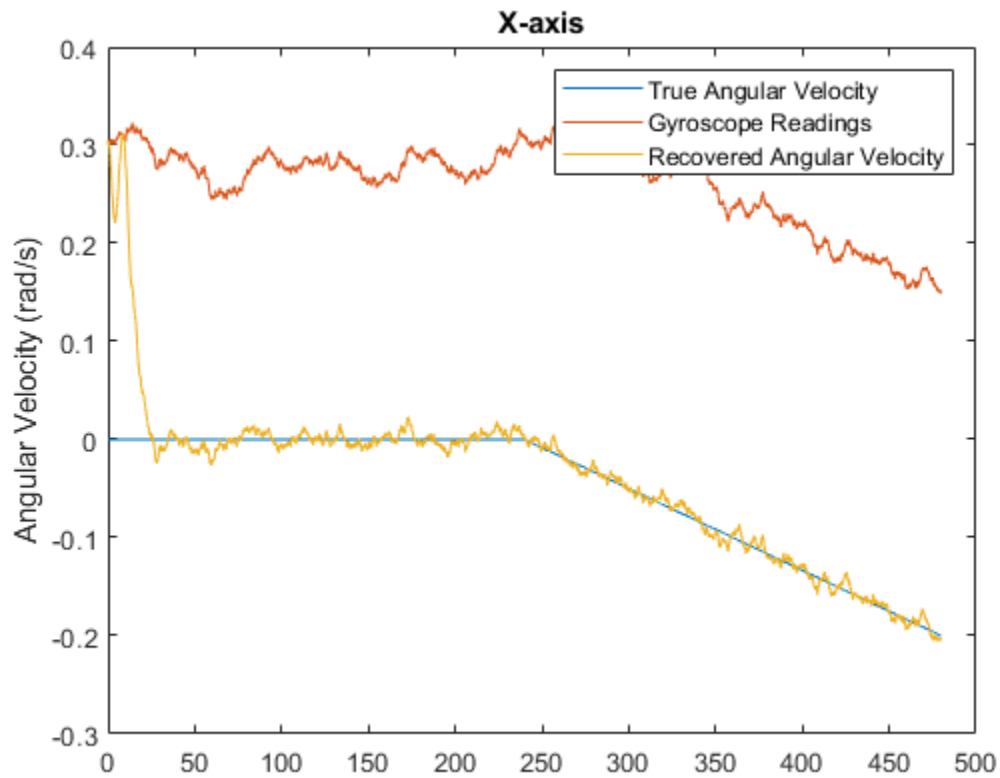


```
figure(2)
plot(time,angVelBody(:,2), ...
     time,gyroReadingsBody(:,2), ...
     time,angVelBodyRecovered(:,2))
title('Y-axis')
ylabel('Angular Velocity (rad/s)')
```
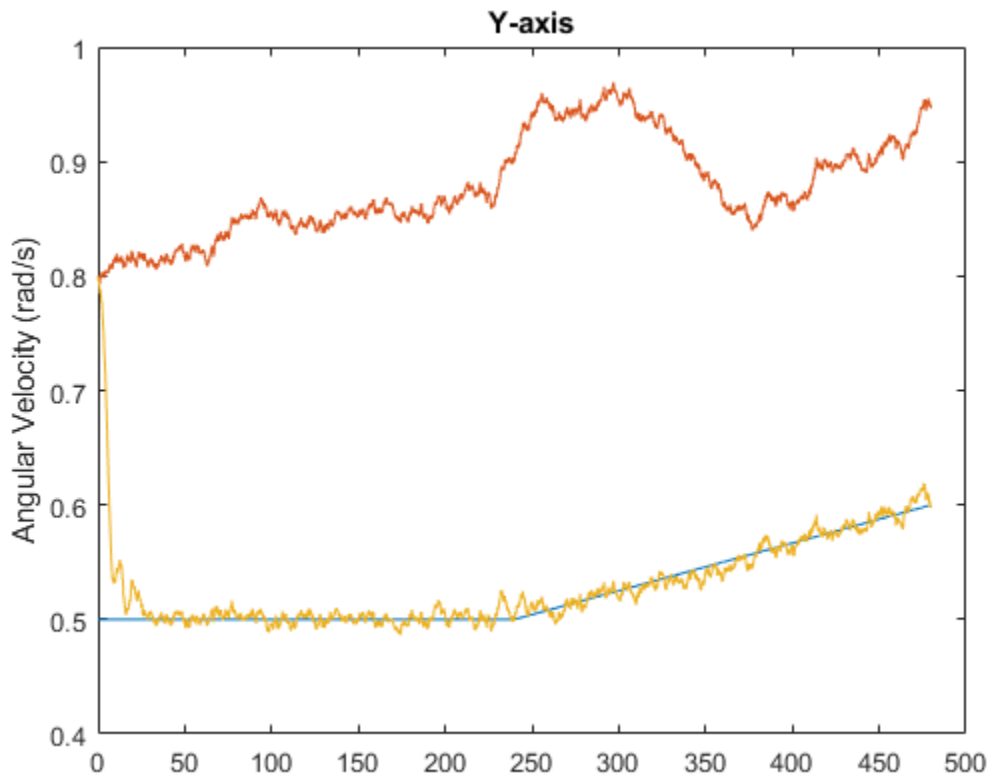
```
figure(3)
plot(time,angVelBody(:,3), ...
     time,gyroReadingsBody(:,3), ...
     time,angVelBodyRecovered(:,3))
title('Z-axis')
ylabel('Angular Velocity (rad/s)')
xlabel('Time (s)')
```
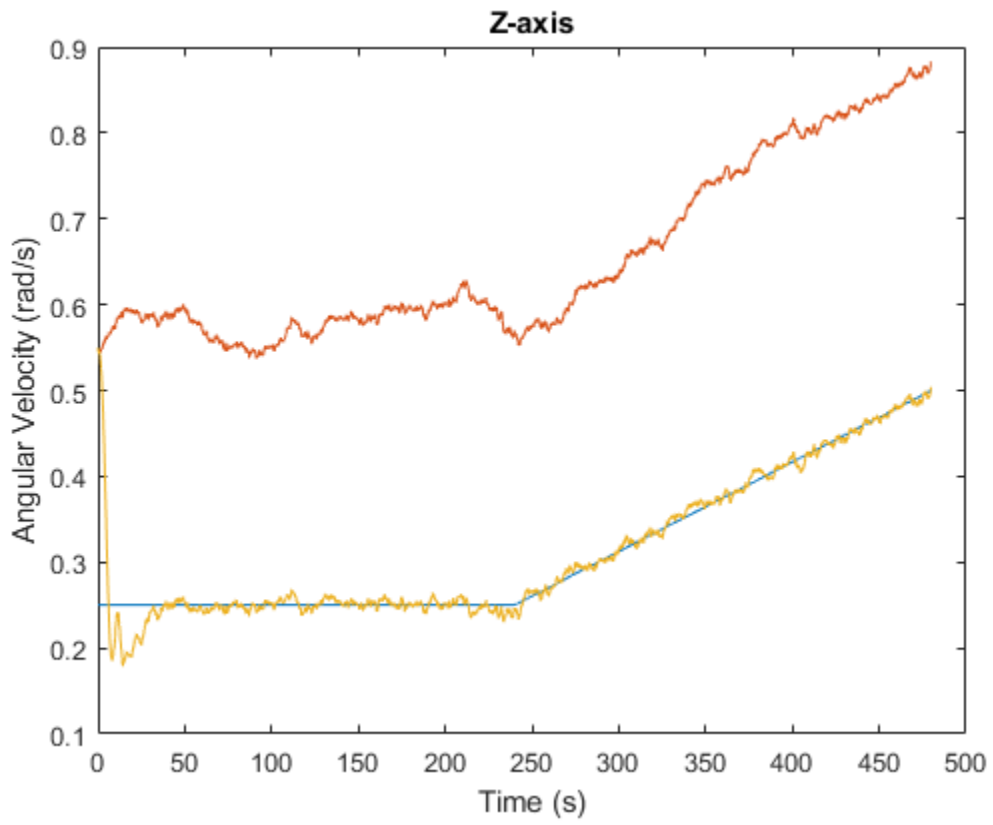
Convert Detections to objectDetection Format

These examples show how to convert actual detections in the native format of the sensor into `objectDetection` objects. `objectDetection` is the standard input format for most filters and trackers in Sensor Fusion and Tracking toolbox. The five examples progressively show how to set up `objectDetection` with varied tracking scenarios.

- Example 1 on page 5-0   configures the detection in a stationary rectangular frame.
- Example 2 on page 5-0   configures the detection in a moving rectangular frame.
- Example 3 on page 5-0   configures the detection in a moving spherical frame.
- Example 4 on page 5-0   shows how to express detections obtained by consecutive rotations.
- Example 5 on page 5-0   shows how to configure 3-D detections.

An `objectDetection` report must contain the basic detection information: `Time` and `Measurement`. It can also contain other key properties, including `MeasurementNoise`, `SensorIndex`, `ObjectClassID`, `ObjectAttributes`, and `MeasurementParameters`. Setting up `MeasurementParameters` correctly so that a filter or tracker can interpret the measurement is crucial in creating `objectDetection`. The first example shows the basic setup of an `objectDetection`. The remaining examples focus on how to correctly set up `MeasurementParameters`.

Example 1: Convert Detections in Stationary Rectangular Frame

Consider a 2-D tracking scenario with a stationary tower and a truck. The tower located at the origin of the scenario frame is equipped with a radar sensor. At $t = 0$ seconds, the truck at the position of (10,20,0) meters is traveling in the positive X direction at a speed of 5 m/s.
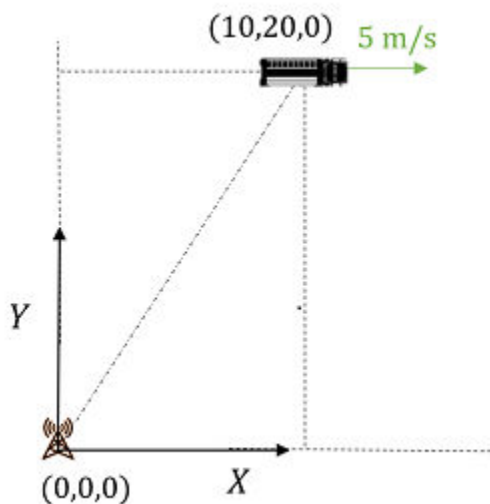
The radar sensor outputs 3-D position and velocity measurements in the scenario frame, so the measurement can be written as follows:

```
measurement1 = [10;20;0;5;0;0]; % [x;y;z;vx;vy;vz]
```

You can specify additional properties such as `MeasurmentNoise`, `SensorIndex`, `ObjectClassID`, and `ObjectAttributes` for the `objectDetection` object. For example, assuming the standard

5-469

deviation of the position and velocity measurement noise is 10 m and 1 m/s, respectively, you can define the measurement error covariance matrix as:

```
measurementNoise1 = diag([10*ones(3,1);ones(3,1)]);
```

Create an `objectDetection` using these values.

```
time1 = 0; % detection time
detect1 = objectDetection(time1,measurement1,'MeasurementNoise',measurementNoise1)
```

```
detect1 =
  objectDetection with properties:

                   Time: 0
            Measurement: [6x1 double]
       MeasurementNoise: [6x6 double]
            SensorIndex: 1
          ObjectClassID: 0
   MeasurementParameters: {}
        ObjectAttributes: {}
```

Example 2: Convert Detections in Moving Rectangular Frame

Consider a 2-D tracking scenario with an ego car and a truck. At $t = 0$ seconds, the car is located at (20,10,0) meters with respect to the scenario frame. The car is moving with a speed of 5 m/s in the Y direction of the scenario frame. The local (forward) frame of the ego car, $\{x,y\}$, rotates from the scenario frame by an angle of 90 degrees. As in the previous example, a truck at the position of (10,20,0) meters is traveling in the positive X direction at a speed of 5 m/s.
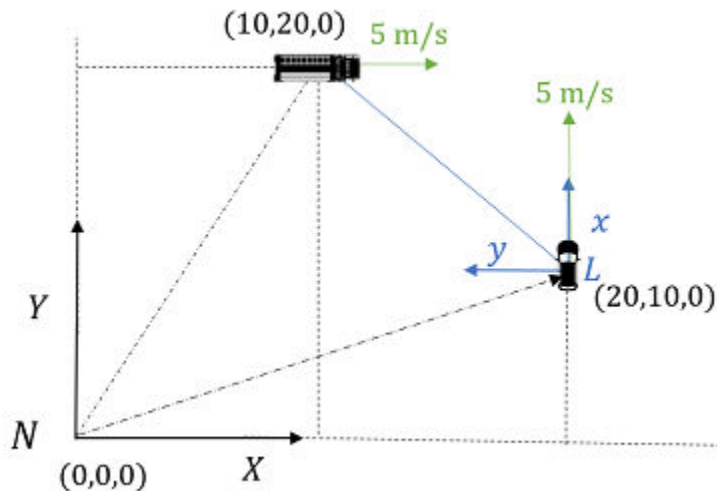
Meanwhile, the ego car is observing the truck in its own local frame, $\{x,y\}$. In practice, you can obtain the measurement directly from the sensor system of the ego car. From the figure, the measurements of the truck are [10; 10; 0 -5; -5; 0] with respect to the $\{x,y\}$ frame in the order of [x;y;z;vx;vy;vz].

```
measurement2 = [10; 10; 0; -5;  -5; 0]; % [x;y;z;vx;vy;vz]
```

To specify the object detection, you need to specify the coordinate transformation from the scenario rectangular frame {*X,Y*} to the local rectangular frame {*x,y*}. You can use the MeasurementParameters property of objectDetection to specify these transformation parameters. In the transformation, the scenario frame is the *parent* frame, and the ego car local frame is the *child* frame.

- The Frame property sets the child frame type to 'rectangular' (in this example) or 'spherical'.
- The OriginPosition property sets the position of the origin of the child frame with respect to the parent frame.
- The OriginVelocity property sets the velocity of the origin of the child frame with respect to the parent frame.

```
MP2 = struct();

MP2.Frame = 'rectangular';
MP2.OriginPosition =[20; 10; 0];
MP2.OriginVelocity = [0; 5; 0];
```

Specify rotation using the rotation matrix converted from Euler angles. Set IsParentToChild to true to indicate rotation from the parent frame to the child frame.

```
rotAngle2 = [90 0 0]; % [yaw,pitch,row]
rotQuat2 = quaternion(rotAngle2,'Eulerd','ZYX','frame');
rotMatrix2 = rotmat(rotQuat2,'frame');
MP2.Orientation = rotMatrix2;
MP2.IsParentToChild = true;
```

Specify measurements.

- Set HasElevation and HasAzimuth both to false, since the child frame is rectangular.
- Set HasRange to true to enable position measurement.
- Set HasVelocity to true to enable velocity measurement.

```
MP2.HasElevation = false;
MP2.HasAzimuth = false;
MP2.HasRange = true;
MP2.HasVelocity = true;
```

Create the objectDetection object and specify the MeasurementParameters property.

```
time2 = 0;
detection2 = objectDetection(time2,measurement2,'MeasurementParameters',MP2)

detection2 =
  objectDetection with properties:

                   Time: 0
            Measurement: [6x1 double]
       MeasurementNoise: [6x6 double]
            SensorIndex: 1
          ObjectClassID: 0
   MeasurementParameters: [1x1 struct]
       ObjectAttributes: {}
```

To verify the object detection, you can use the `cvmeas` measurement function to regenerate the measurement. The `cvmeas` function can take the actual state of the target and measurement parameters as input. The state input of `cvmeas` is in the order of [x;vx;y;vy;z;vz]. As shown in the following output, the results agree with `measurement2`.

```
state2 =[10;5;20;0;0;0]; % [x;vx;y;vy;z;vz]
cvmeas2 = cvmeas(state2,MP2)% [x;y;z;vx;vy;vz]
```

cvmeas2 = 6×1

```
    10.0000
    10.0000
         0
    -5.0000
    -5.0000
         0
```

Example 3: Convert Detections in Moving Spherical Frame

Consider the previous tracking scenario, only now the measurement is obtained by a scanning radar with a spherical output frame. The boresight direction of the radar is aligned with the Y direction (same as x direction) at t = 0 seconds.
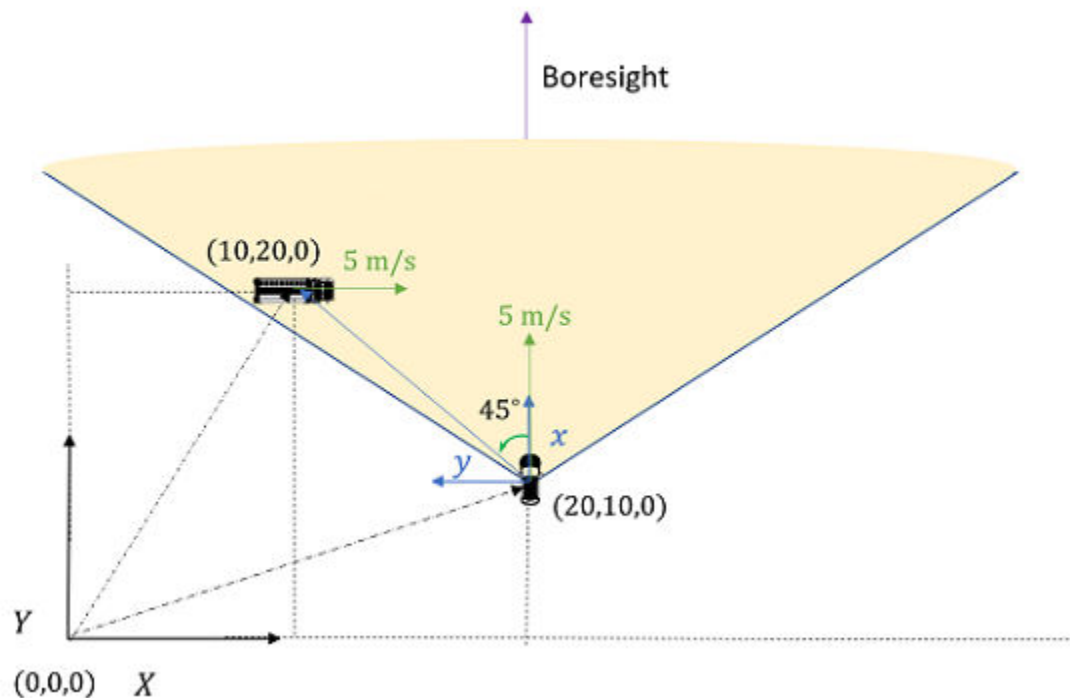
Since the relative velocity between the truck and the car is in the line-of-sight direction, the measurement, which is in the order of [azimuth; elevation; range; range-rate], can be obtained as follows:

```
measurement3 =[45; 0; 10/sind(45); -5/sind(45)]; % [az;el;rng;rr]. Units in degrees.
```

Specify the measurement parameters.

```
MP3 = struct();

MP3.Frame = 'spherical'; % The child frame is spherical.
MP3.OriginPosition = [20; 10; 0];
MP3.OriginVelocity = [0; 5; 0];

% Specify rotation.
rotAngle3 = [90 0 0];
rotQuat3 = quaternion(rotAngle3,'Eulerd','ZYX','frame');
rotMatrix3 = rotmat(rotQuat3,'frame');
MP3.Orientation = rotMatrix3;
MP3.IsParentToChild = true;
```

Set `HasElevation` and `HasAzimuth` to `true` to output azimuth and elevation angles in the spherical child frame. Set `HasRange` and `HasVelocity` both to `true` to output range and range-rate, respectively.

```
MP3.HasElevation = true;
MP3.HasAzimuth = true;
MP3.HasRange = true;
MP3.HasVelocity = true;
```

Create the `objectDetection` object.

```
time3 = 0;
detection3 = objectDetection(time3,measurement3,'MeasurementParameters',MP3)

detection3 =
  objectDetection with properties:

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

Verify the results using `cvmeas`. The results agree with `measurement3`.

```
state3 = [10;5;20;0;0;0]; % [x;vx;y;vy;z;vz]
cvmeas3 = cvmeas(state3,MP3) % [az;el;rng;rr]

cvmeas3 = 4×1

   45.0000
        0
   14.1421
   -7.0711
```

Example 4: Convert Detections Between Three Frames

Consider the previous tracking scenario, only now the boresight direction of the radar rotates 45 degrees from the *x* direction of the car's local frame.
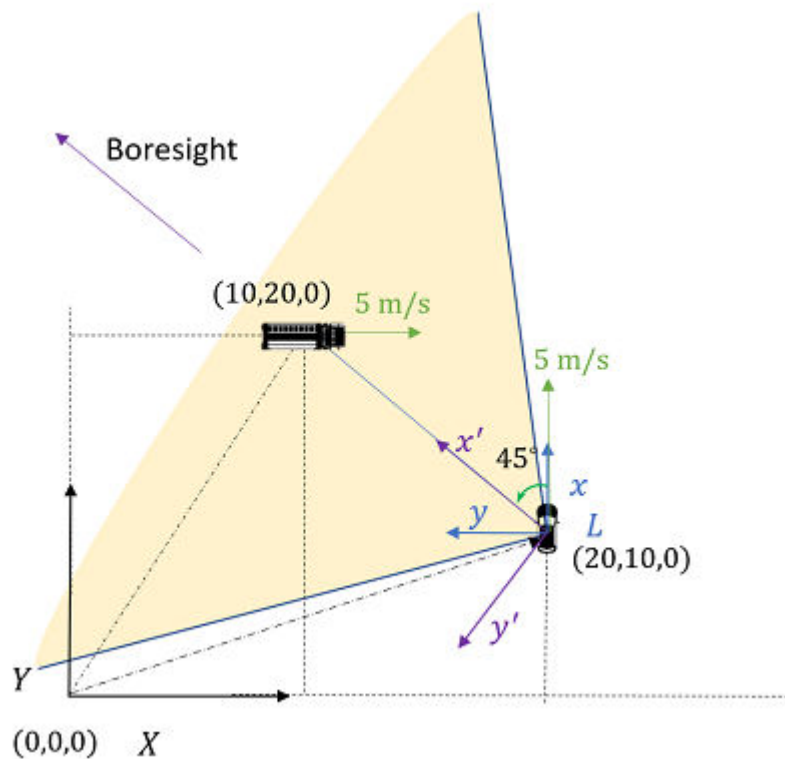
The new measurements, expressed in the new spherical frame $\{x',y'\}$, are:

```
measurement4 = [0; 0; 10/sind(45); -5/sind(45)]; % [az;el;rng;rr]
```

For the measurement parameters, you can specify the rotation as a 135-degree rotation from the scenario frame to the new spherical frame. Alternately, you can specify it as two consecutive rotations: rectangular $\{X,Y\}$ to rectangular $\{x,y\}$ and rectangular $\{x,y\}$ to spherical $\{x',y'\}$. To illustrate the multiple frame transformation feature supported by the `MeasurementParameters` property, this example uses the latter approach.

The first set of measurement parameters is exactly the same as `MP2` used in Example 2 on page 5-0   . `MP2` accounts for the rotation from the rectangular $\{X,Y\}$ to the rectangular $\{x,y\}$. For the second set of measurement parameters, `MP4`, you need to specify only a 45-degree rotation from the rectangular $\{x,y\}$ to the spherical $\{x',y'\}$.

```
MP4 = struct();

MP4.Frame = 'spherical';
MP4.OriginPosition =[0; 0; 0]; % Colocated positions.
MP4.OriginVelocity = [0; 0; 0]; % Same origin velocities.

% Specify rotation.
rotAngle4 = [45 0 0];
rotQuat4 = quaternion(rotAngle4,'Eulerd','ZYX','frame');
rotMatrix4 = rotmat(rotQuat4,'frame');
MP4.Orientation = rotMatrix4;
MP4.IsParentToChild = true;

% Specify outputs in the spherical child frame.
```

```
MP4.HasElevation = true;
MP4.HasAzimuth = true;
MP4.HasRange = true;
MP4.HasVelocity = true;
```

Create the combined `MeasurementParameters` input, MPc.

```
MPc =[MP4 MP2];
```

Create the `objectDetection` object.

```
time4 = 0;
detection4 = objectDetection(time4,measurement4,'MeasurementParameters',MPc)
```

```
detection4 =
  objectDetection with properties:

                    Time: 0
             Measurement: [4x1 double]
        MeasurementNoise: [4x4 double]
             SensorIndex: 1
           ObjectClassID: 0
    MeasurementParameters: [1x2 struct]
        ObjectAttributes: {}
```

Verify the results using `cvmeas`. The result agrees with `measurement4`.

```
state4 = [10;5;20;0;0;0]; % [x;vx;y;vy;z;vz]
cvmeas4 = cvmeas(state4,MPc) % [az;el;rr;rrate]
```

```
cvmeas4 = 4×1

    0.0000
         0
   14.1421
   -7.0711
```

Example 5: Convert 3D Detections

Consider an unmanned aerial vehicle (UAV) monitoring a region. At $t = 0$ seconds, the UAV is at the position of (5,5,-1) km with respect to the global north-east-down (NED) frame. The velocity of the UAV is (-50,-100,5) m/s. The orientation of the UAV body frame $\{x,y,z\}$ with respect to the global NED frame is given as (-120,2,2) degrees in yaw, pitch, and roll. At the same time, a car at the position of (1,1,0) km is moving east with a speed of 30 m/s. The UAV measures the car using a radar system aligned with its own body axis.
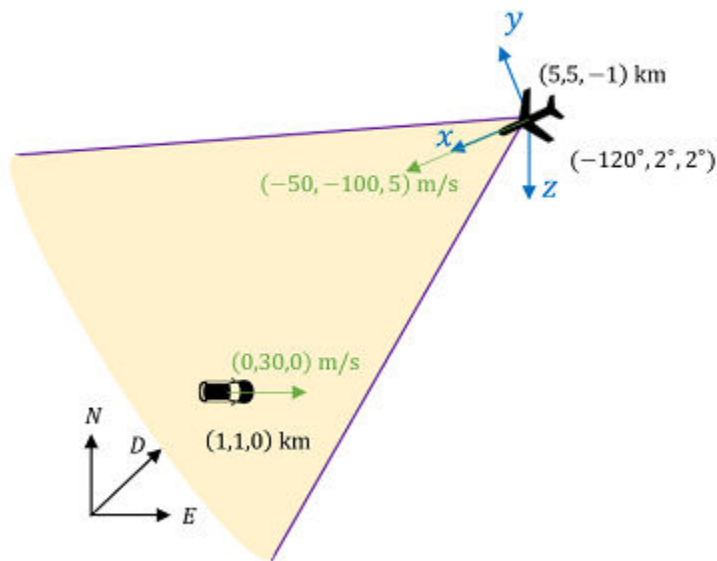
Based on this information, specify the kinematic parameters for the measurement transformation.

Specify the frame type, origin position, and origin velocity of the UAV body frame.

```
MP5 = struct();
```

```
MP5.Frame = 'spherical';
MP5.OriginPosition = [5000; 5000; -1000];
MP5.OriginVelocity = [-50; -100; 5];
```

Specify the rotation from the NED frame to the UAV body frame.

```
Rot_angle5 = [-120 2 2]; % [yaw,pitch,roll]
Rot_quat5 = quaternion(Rot_angle5,'Eulerd','ZYX','frame');
Rot_matrix5 = rotmat(Rot_quat5,'frame');
MP5.Orientation = Rot_matrix5;
MP5.IsParentToChild = true;
```

Specify the output measurements in a spherical frame.

```
MP5.HasElevation = true;
MP5.HasAzimuth = true;
MP5.HasRange = true;
MP5.HasVelocity = true;
```

You can obtain the measurement directly from the radar system on the UAV. Use the `cvmeas` function is to obtain the measurement. The measurement is in the order of [azimuth;elevation;range;range-rate].

```
car_state5 = [1000;0;1000;30;0;0]; % [x;vx;y;vy;z;vz].
measurement5 = cvmeas(car_state5,MP5);
meas_az5 = measurement5(1)
```

```
meas_az5 = -14.6825
```

```
meas_el5 = measurement5(2)

meas_el5 = 12.4704

meas_rng5 = measurement5(3)

meas_rng5 = 5.7446e+03

meas_rr5 = measurement5(4)

meas_rr5 = -126.2063
```

The elevation angle is defined as an angle from the xy-plane to the z direction. That is why the elevation angle is positive for a target on the ground relative to the UAV. This convention is used throughout the toolbox.

The measurement noise for azimuth, elevation, range, and range-rate is [1,1,20,2], respectively. Also, the index of the radar is 2, and the radar can classify the detected object as 1 for the type of 'car'.

```
index5 = 2;
covariance5 = diag([1;1;20;2]);
classID5 = 1;
```

Create an objectDetection object for the detection.

```
time5 = 0;
detection = objectDetection(time5,measurement5,'SensorIndex',index5,...
            'MeasurementNoise',covariance5,'ObjectClassID',classID5,'MeasurementParameters',MP5)

detection =
  objectDetection with properties:

                    Time: 0
             Measurement: [4x1 double]
        MeasurementNoise: [4x4 double]
             SensorIndex: 2
           ObjectClassID: 1
    MeasurementParameters: [1x1 struct]
        ObjectAttributes: {}
```

Estimating Orientation Using Inertial Sensor Fusion and MPU-9250

This example shows how to get data from an InvenSense MPU-9250 IMU sensor and to use the 6-axis and 9-axis fusion algorithms in the sensor data to compute orientation of the device.

MPU-9250 is a 9-axis sensor with accelerometer,gyroscope, and magnetometer. The accelerometer measures acceleration, the gyroscope measures angular velocity, and the magnetometer measures magnetic field in x-, y- and z- axis. The axis of the sensor depends on the make of the sensor.

Required MathWorks® Products

- MATLAB®
- MATLAB Support Package for Arduino® Hardware
- Sensor Fusion and Tracking Toolbox™

Required Hardware

- Arduino Uno
- InvenSense MPU-9250

Hardware Connection

Connect the SDA, SCL, GND, and the VCC pins of the MPU-9250 sensor to the corresponding pins on the Arduino® Hardware. This example uses the Arduino Uno board with the following connections:

- SDA - A4
- SCL - A5
- VCC - +3.3V
- GND - GND
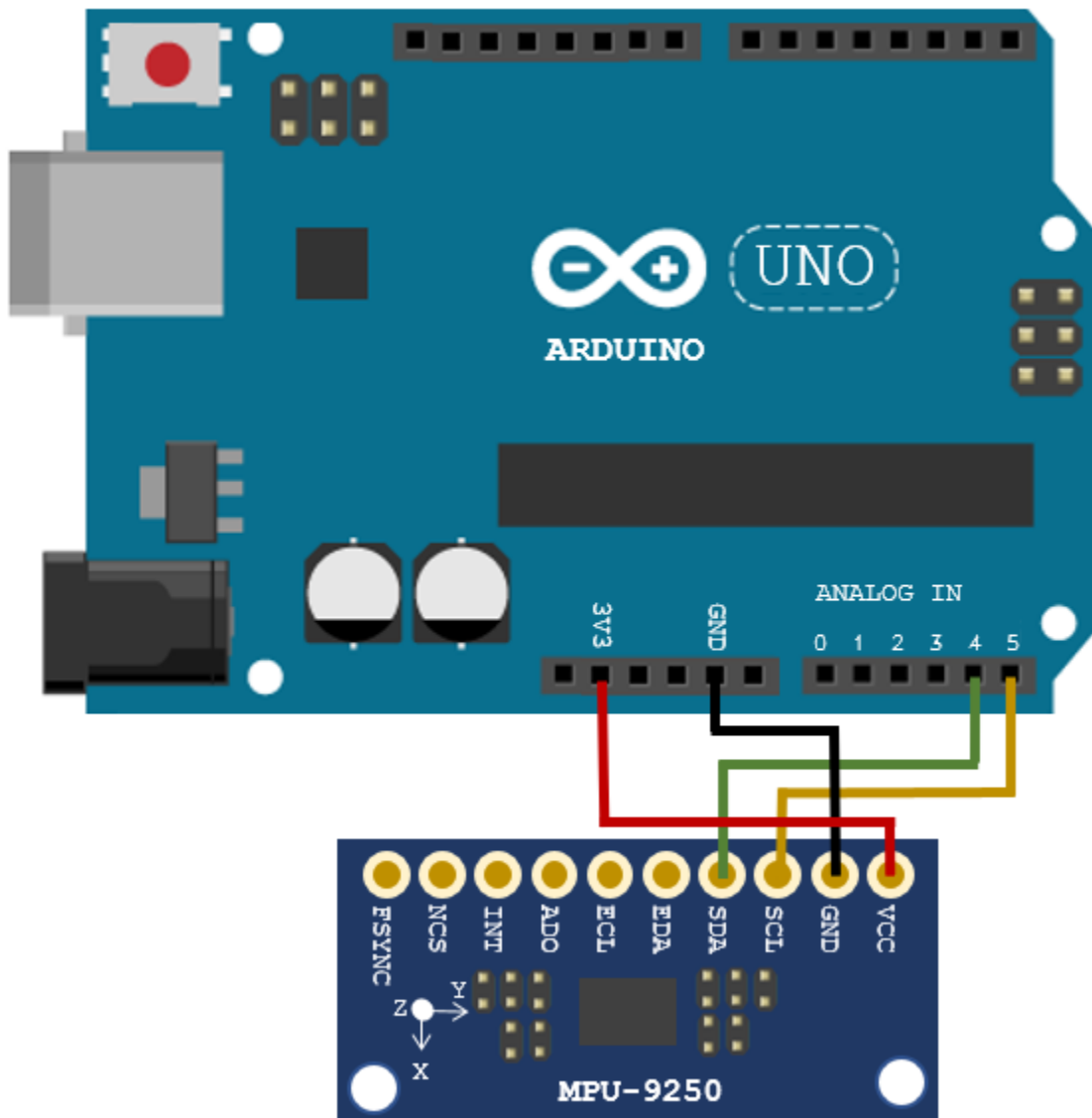
Ensure that the connections to the sensors are intact. It is recommended to use a prototype shield and solder the sensor to it to avoid loose connections while moving the sensor. Refer the "Troubleshooting Sensors" (MATLAB Support Package for Arduino Hardware) page for sensors to debug the sensor related issues.

Create Sensor Object

Create an `arduino` object and include the I2C library.

```
a = arduino('COM9', 'Uno', 'Libraries', 'I2C');
```

```
Updating server code on board Uno (COM9). This may take a few minutes.
```

Create the MPU-9250 sensor object.

```
fs = 100; % Sample Rate in Hz
imu = mpu9250(a,'SampleRate',fs,'OutputFormat','matrix');
```

Compensating For Hard Iron Distortions

Fusion algorithms use magnetometer readings which need to compensate for magnetic distortions such as hard iron distortion. Hard iron distortions are produced by materials which create a magnetic field, resulting in shifting the origin on the response surface. These distortions can be corrected by subtracting the correction values from the magnetometer readings for each axis. In order to find the correction values, do the following:

1. Rotate the sensor from 0 to 360 degree around each axis.

2. Obtain the minimum and maximum magnetometer readings.

3. Average the minimum and maximum readings to get the correction values for each axis.

The correction values change with the surroundings.

The following code snippets can be used to obtain bias values for x axis, similar procedure can be followed for other axes as well:

```
displayMessage(['Fusion algorithms use magnetometer readings which need to compensate for magneti
    'The given code snippet can be used to find the correction values for compensating Hard Iron
    'is executing, rotate the sensor around x axis from 0 to 360 degree. For other axes, modify
    'sensor along that axis'],'Compensating Hard Iron Distortions');
tic;
stopTimer = 100;
magReadings=[];
while(toc<stopTimer)
    % Rotate the sensor around x axis from 0 to 360 degree.
    % Take 2-3 rotations to improve accuracy.
    % For other axes, rotate around that axis.
    [accel,gyro,mag] = read(imu);
    magReadings = [magReadings;mag];
end

% For y axis, use magReadings (:,2) and for z axis use magReadings(:,3)
magx_min = min(magReadings(:,1));
magx_max = max(magReadings(:,1));
magx_correction = (magx_max+magx_min)/2;
```

For more accurate tracking, calibrate the magnetometer for other distortions as well. The `magcal` function (this function is available in the Sensor Fusion and Tracking Toolbox™) can be used to compensate soft iron distortions as well. Change the correction values calculated for your sensor in the `readSensorDataMPU9250` function in the example folder.

Aligning the axis of MPU-9250 sensor with NED Coordinates

Sensor Fusion algorithms used in this example use North-East-Down(NED) as a fixed, parent coordinate system. In the NED reference frame, the X-axis points north, the Y-axis points east, and the Z-axis points down. Depending on the algorithm, north may either be the magnetic north or true north. The algorithms in this example use the magnetic north. The algorithms used here expects all the sensors in the object to have their axis aligned and is in accordance with NED convention.

MPU-9250 has two devices, the magnetometer and the accelerometer-gyroscope, on the same board. The axes of these devices are different from each other. The magnetometer axis is aligned with the NED coordinates. The axis of the accelerometer-gyroscope is different from magnetometer in MPU-9250. The accelerometer and the gyroscope axis need to be swapped and/or inverted to match

the magnetometer axis. For more information refer to the section "Orientation of Axes" section in MPU-9250 datasheet.
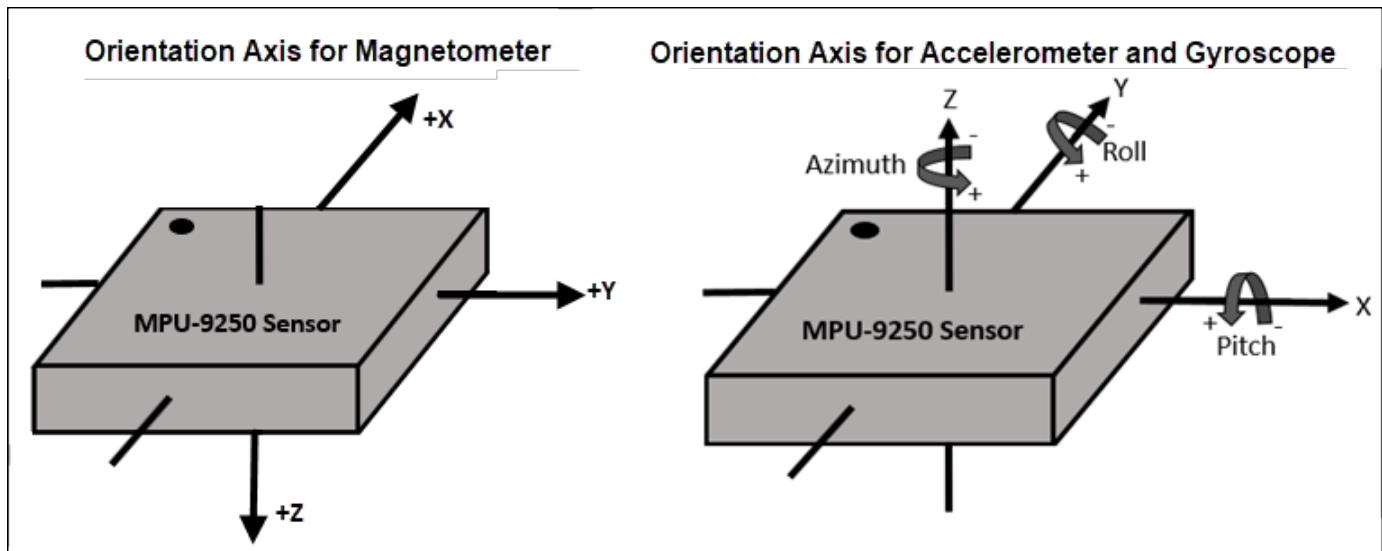
To align MPU-9250 accelerometer-gyroscope axes to NED coordinates, do the following:

1. Define device axes: Define the imaginary axis as the device axis on the sensor in accordance to NED coordinate system which may or may not be same as sensor axes. For MPU-9250, magnetometer axis can be considered as device axis.

2. Swap the x and y values of accelerometer and gyroscope readings, so that the accelerometer and gyroscope axis is aligned with magnetometer axis.

3. Determine polarity values for accelerometer, and gyroscope.

a. Accelerometer

- Place the sensor such that device X axis is pointing downwards, perpendicular to the surface at which sensor is kept. Accelerometer readings should read approximately [9.8 0 0]. If not negate the x-values of accelerometer.
- Place the sensor such that device Y axis is pointing downwards, perpendicular to the surface at which sensor is kept. Accelerometer readings should read approximately [0 9.8 0]. If not negate the y-values of accelerometer.
- Place the sensor such that device Z axis is pointing downwards, perpendicular to the surface at which sensor is kept. Accelerometer readings should read approximately [0 0 9.8]. If not negate the z-values of accelerometer.

b. Gyroscope

Rotate the sensor along each axis and capture the readings. Use the right hand screw rule to correct the polarity of rotation.

The above method is used to set the axis of the sensor in this example.

Tuning Filter Parameters

The algorithms used in this example, when properly tuned, enable estimation of orientation and are robust against environmental noise sources. You must consider the situations in which the sensors

are used and tune the filters accordingly. Refer Tuning Filter Parameters for more details related to tuning filter parameters.

The example demonstrates three algorithms to determine orientation, namely `ahrsfilter`, `imufilter`, and `ecompass`. Refer Determine Orientation Using Inertial Sensors for more details related to inertial fusion algorithms.

Accelerometer-Gyroscope-Magnetometer Fusion

An attitude and heading reference system (AHRS) consist of a 9-axis system that uses an accelerometer, gyroscope, and magnetometer to compute orientation of the device. The `ahrsfilter` produces a smoothly changing estimate of orientation of the device, while correctly estimating the north direction. The `ahrsfilter` has the ability to remove gyroscope bias and can also detect and reject mild magnetic jamming.

The following code snippets use `ahrsfilter` system object to determine orientation of the sensor and creates a figure which gets updated as you move the sensor. The sensor has to be stationary, before the start of this example.

```
% GyroscopeNoise and AccelerometerNoise is determined from datasheet.
GyroscopeNoiseMPU9250 = 3.0462e-06; % GyroscopeNoise (variance value) in units of rad/s
AccelerometerNoiseMPU9250 = 0.0061; % AccelerometerNoise(variance value)in units of m/s^2
viewer = HelperOrientationViewer('Title',{'AHRS Filter'});
FUSE = ahrsfilter('SampleRate',imu.SampleRate, 'GyroscopeNoise',GyroscopeNoiseMPU9250,'Accelerome
stopTimer = 100;
```

While the below code is getting executed, slowly move the sensor and check if the motion in the figure matches the motion of the sensor.

```
% Use ahrsfilter to estimate orientation and update the viewer as the
% sensor moves for time specified by stopTimer
displayMessage(['This section uses AHRS filter to determine orientation of the sensor by collecti
    'system object. Move the sensor to visualize orientation of the sensor in the figure window.
    'click OK'],...
    'Estimate Orientation using AHRS filter and MPU-9250')
tic;
while(toc < stopTimer)
    [accel,gyro,mag] = readSensorDataMPU9250(imu);
    rotators = FUSE(accel,gyro,mag);
    for j = numel(rotators)
        viewer(rotators(j));
    end
end
```

When the device X axis of sensor is pointing to north, the device Y-axis is pointing to east and device Z-axis is pointing down.
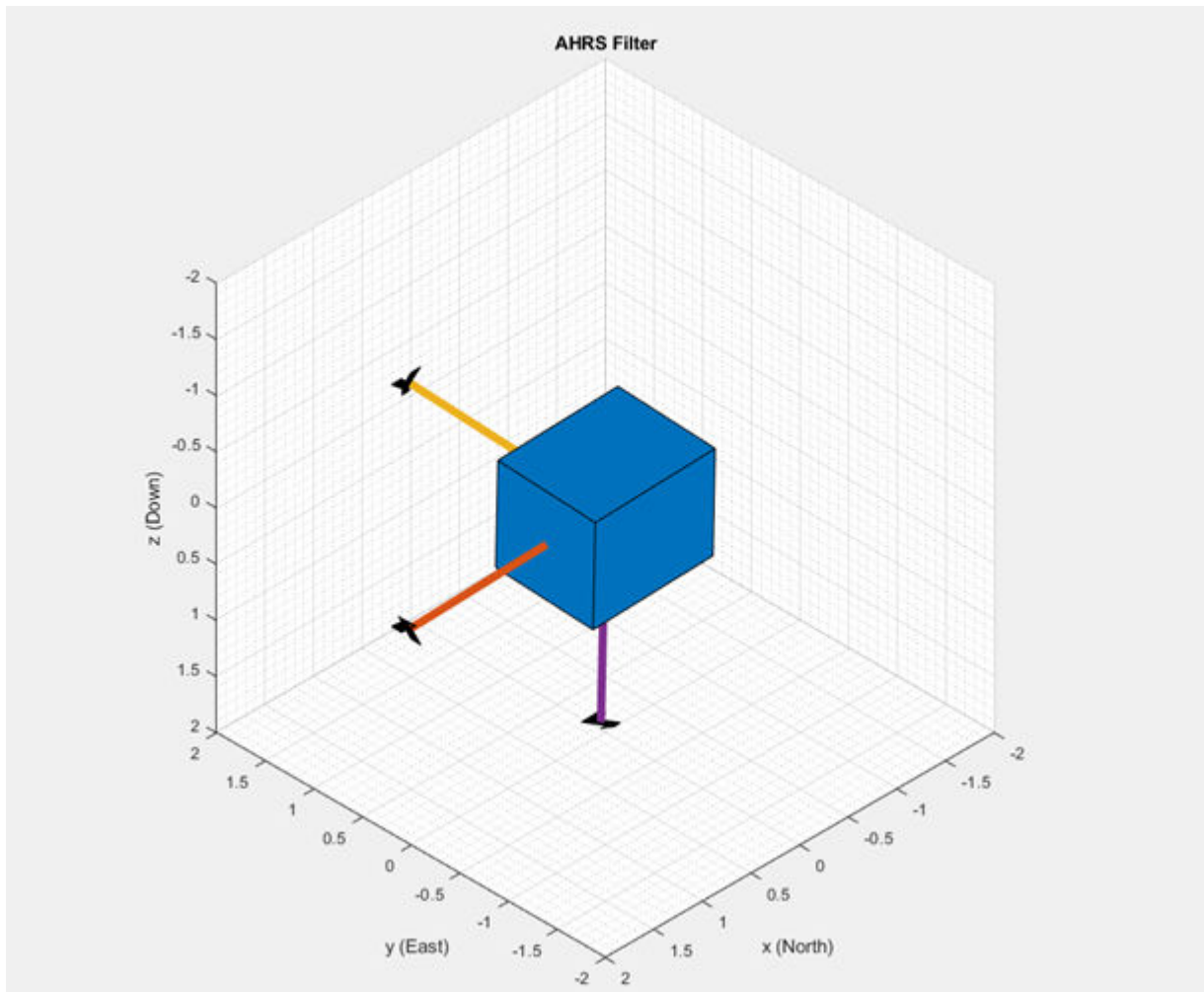
When the device X axis of sensor is pointing to north, device Y-axis is pointing to west and device Z-axis is pointing upwards.
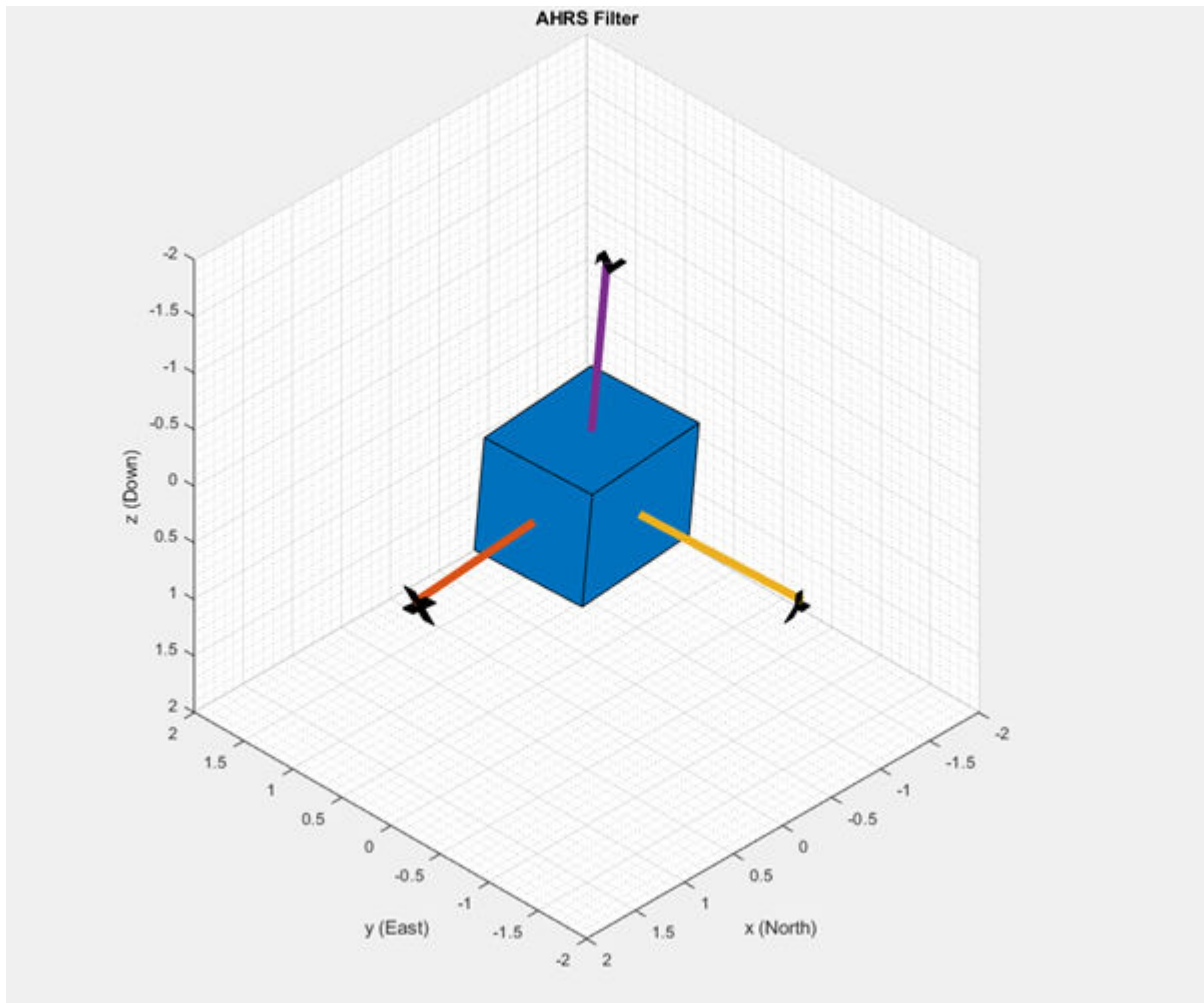
Accelerometer-Gyroscope Fusion

The `imufilter` system object fuses accelerometer and gyroscope data using an internal error-state Kalman filter. The filter is capable of removing the gyroscope bias noise, which drifts over time. The filter does not process magnetometer data, so it does not correctly estimate the direction of north. The algorithm assumes the initial position of the sensor is in such a way that device X-axis of the sensor is pointing towards magnetic north, the device Y-axis of the sensor is pointing to east and the device Z-axis of the sensor is pointing downwards. The sensor must be stationary, before the start of this example.

The following code snippets use `imufilter` object to determine orientation of the sensor and creates a figure which gets updated as you move the sensor.

```
displayMessage(['This section uses  IMU filter to determine orientation of the sensor by collect:
    'system object. Move the sensor to visualize orientation of the sensor in the figure window.
    'click OK'],...
    'Estimate Orientation using IMU filter and MPU-9250.')
% GyroscopeNoise and AccelerometerNoise is determined from datasheet.
GyroscopeNoiseMPU9250 = 3.0462e-06; % GyroscopeNoise (variance) in units of rad/s
AccelerometerNoiseMPU9250 = 0.0061; % AccelerometerNoise (variance) in units of m/s^2
viewer = HelperOrientationViewer('Title',{'IMU Filter'});
```

```
FUSE = imufilter('SampleRate',imu.SampleRate, 'GyroscopeNoise',GyroscopeNoiseMPU9250,'Accelerome
stopTimer=100;
```

While the below code is getting executed, slowly move the sensor and check if the motion in the
figure matches the motion of the sensor.

```
% Use imufilter to estimate orientation and update the viewer as the
% sensor moves for time specified by stopTimer
tic;
while(toc < stopTimer)
    [accel,gyro] = readSensorDataMPU9250(imu);
    rotators = FUSE(accel,gyro);
    for j = numel(rotators)
        viewer(rotators(j));
    end
end
```

The `imufilter` algorithm can also be used with MPU6050 as well, since it does not require
magnetometer values.

When the device X axis of sensor is pointing to north, device Z-axis is pointing downwards and device
Y-axis is pointing to east.
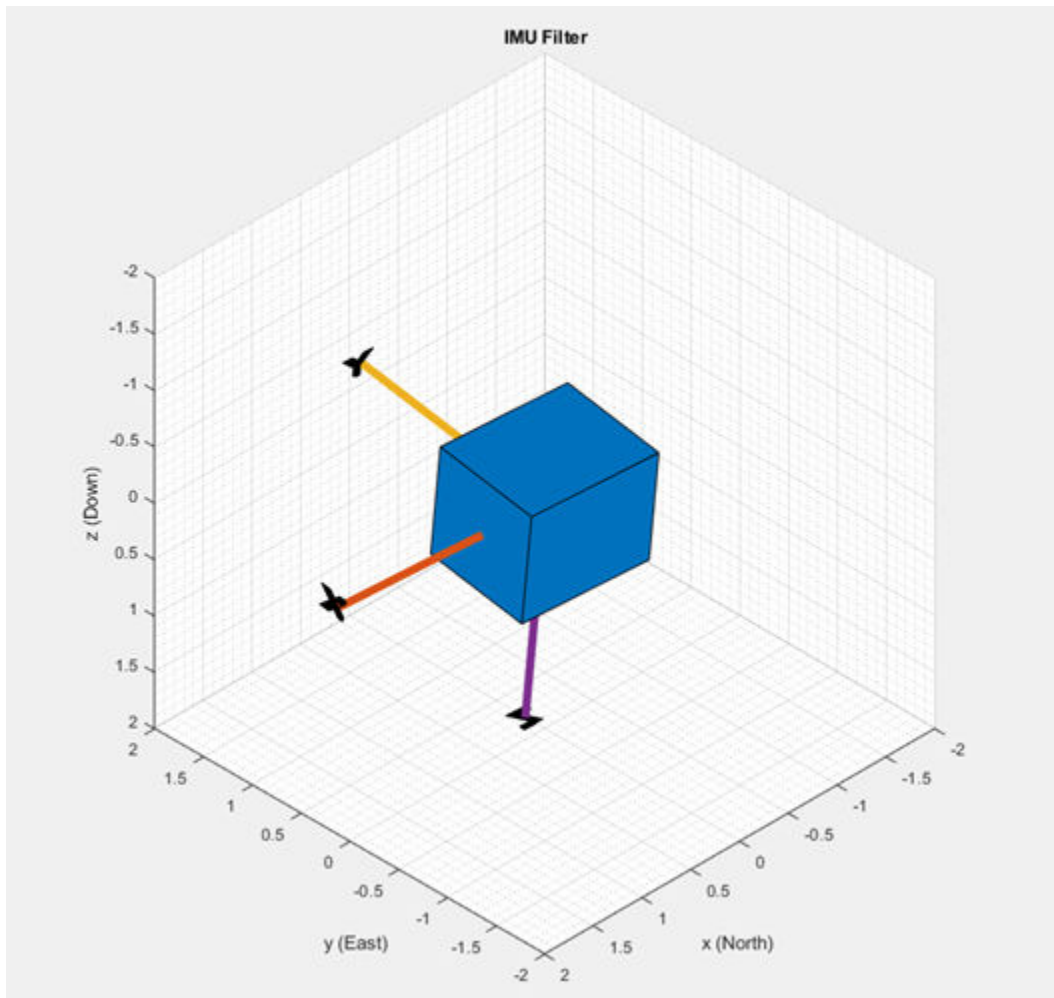
When the device X axis of sensor is pointing upwards, device Y-axis is points to west and device Z-axis points to south.
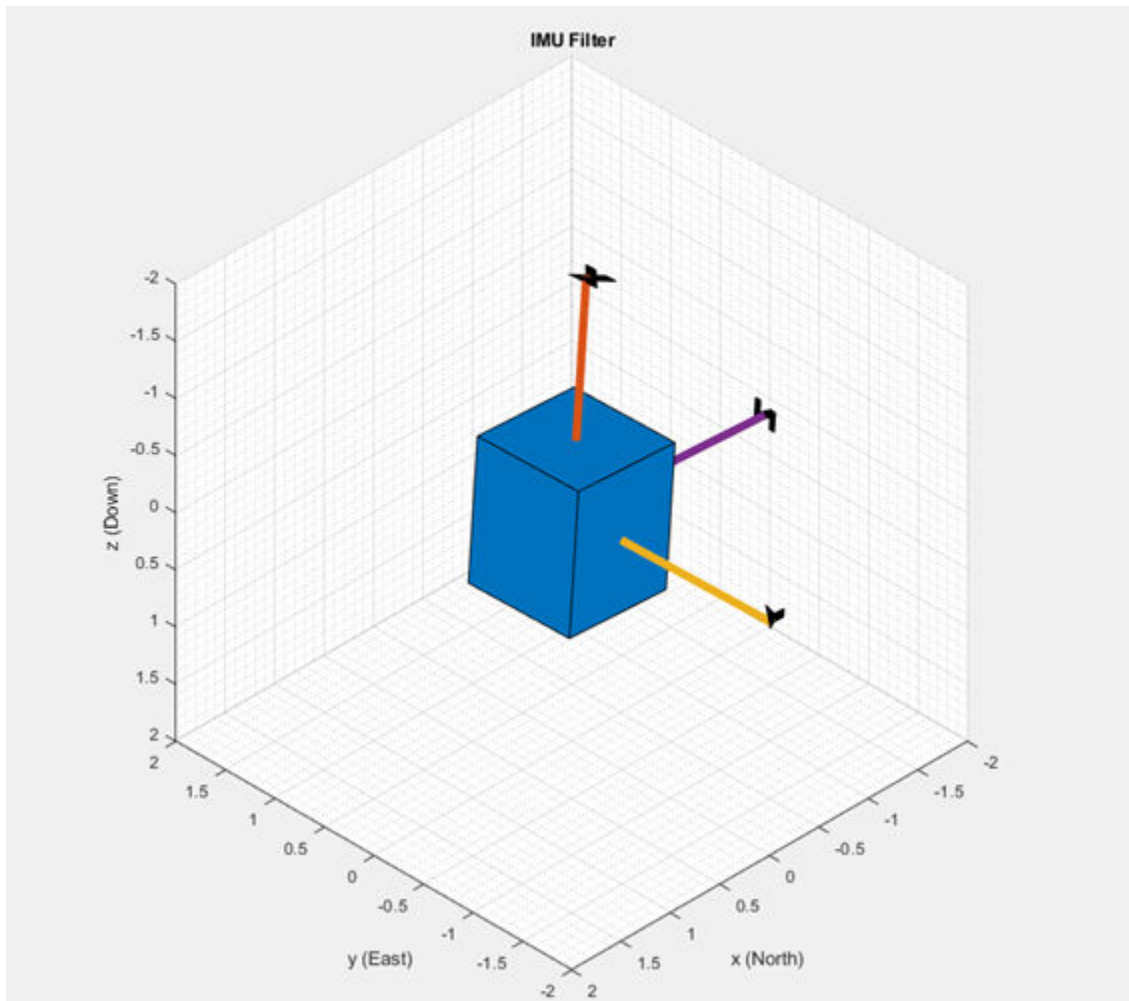
Accelerometer-Magnetometer Fusion

The `ecompass` system object fuses the accelerometer and magnetometer data. The Ecompass algorithm is a memoryless algorithm that requires no parameter tuning but is highly susceptible to sensor noise. You could use spherical linear interpolation (SLERP) to lowpass filter a noisy trajectory. Refer Lowpass Filter Orientation using Quaternion SLERP example for more details

```
displayMessage(['This section uses \slecompass \rmfunction to determine orientation of the senso
    '\rmsystem object. Move the sensor to visualize orientation of the sensor in the figure windo
    'Estimate Orientation using Ecompass algorithm.')
tic;
viewer = HelperOrientationViewer('Title',{'Ecompass Algorithm'});
stopTimer = 100;
tic;
% Use ecompass algorithm to estimate orientation and update the viewer as the
% sensor moves for time specified by stopTimer.
while(toc < stopTimer)
    [accel,gyro,mag] = readSensorDataMPU9250(imu);
    rotators = ecompass(accel,mag);
```

```
    for j = numel(rotators)
        viewer(rotators(j));
    end
end
```

Clean Up

When the connection is no longer needed, release and clear the objects

```
release(imu);
delete(imu);
clear;
```

Things to try

You can try this example with other sensors such as InvenSense MPU-6050 and STMicroelectronics LSM9DS1. Note that the MPU-6050 sensor can be used only with the `imufilter` system object.

Track Simulated Vehicles Using GNN and JPDA Trackers in Simulink

This example shows how to configure and utilize GNN and JPDA trackers in a simulated highway scenario in Simulink® with Sensor Fusion and Tracking Toolbox™. It closely follows the "Sensor Fusion Using Synthetic Radar and Vision Data in Simulink" (Automated Driving Toolbox). A main benefit of modeling the system in Simulink is the simplicity of performing "what-if" analysis and choosing a tracker that results in the best performance based on the requirements.

Introduction

Synthetic radar and vision data help us in evaluating the target tracking algorithms. This example primarily focuses on the tracking abilities of `trackerGNN` and `trackerJPDA` in Simulink.

Setup and Overview of the Model

The setup of this example is the same as the setup of "Sensor Fusion Using Synthetic Radar and Vision Data in Simulink" (Automated Driving Toolbox) example, except that the simulated detections are concatenated as the input to the `Trackers` block.

Trackers

The `Trackers` block is a variant subsystem, which you can use to switch between the GNN tracker and the JPDA tracker.

The first variant of the Trackers block, `trackerGNN`, assumes a constant velocity motion model and an extended Kalman filter by setting the Filter initialization function as the default `initcvekf`.
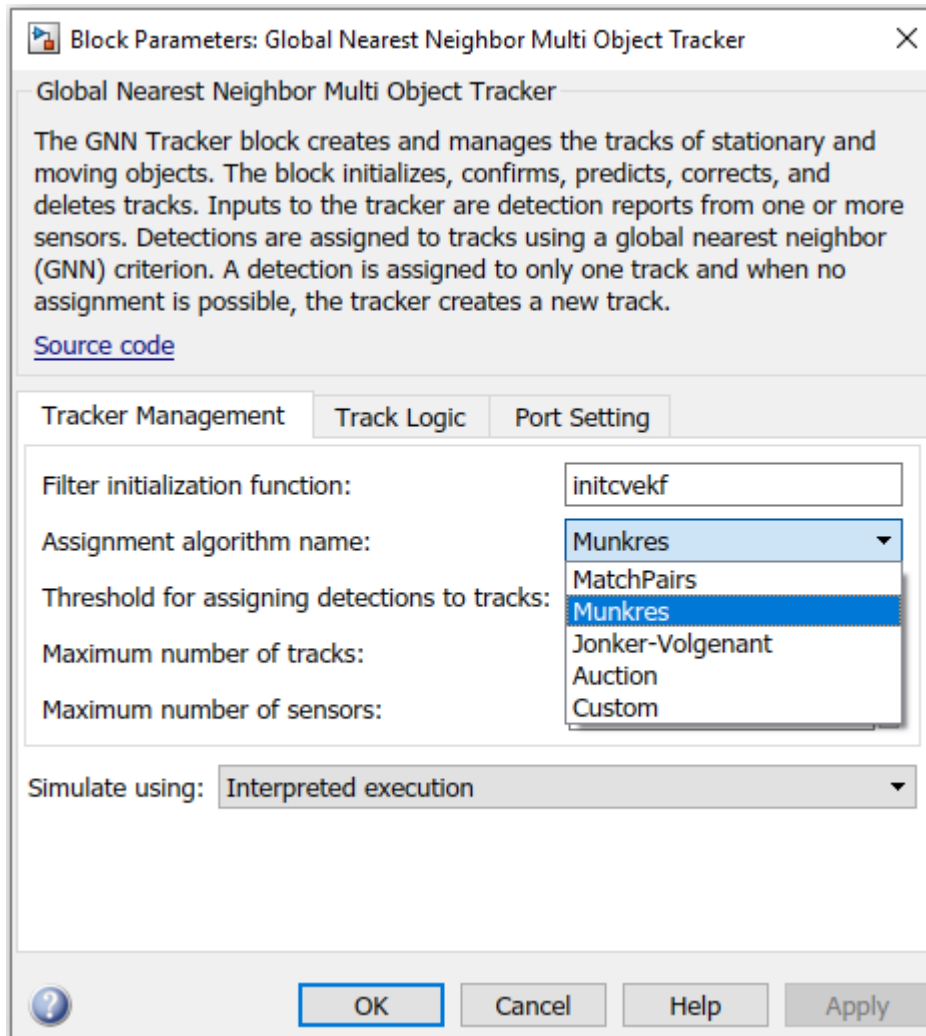
With GNN tracker, you can choose your own customized filter which fits the motion of the simulated objects. You can choose the assignment algorithm among MatchPairs, Munkres, Jonker-Volgenant, Auction and your own customized assignment algorithm. You can also specify the track maintenance logic as `History` or `Score`.

The second variant of the Trackers block, `trackerJPDA`, also assumes a constant velocity motion model and an extended Kalman filter by the default filter initialization function `initcvekf`.
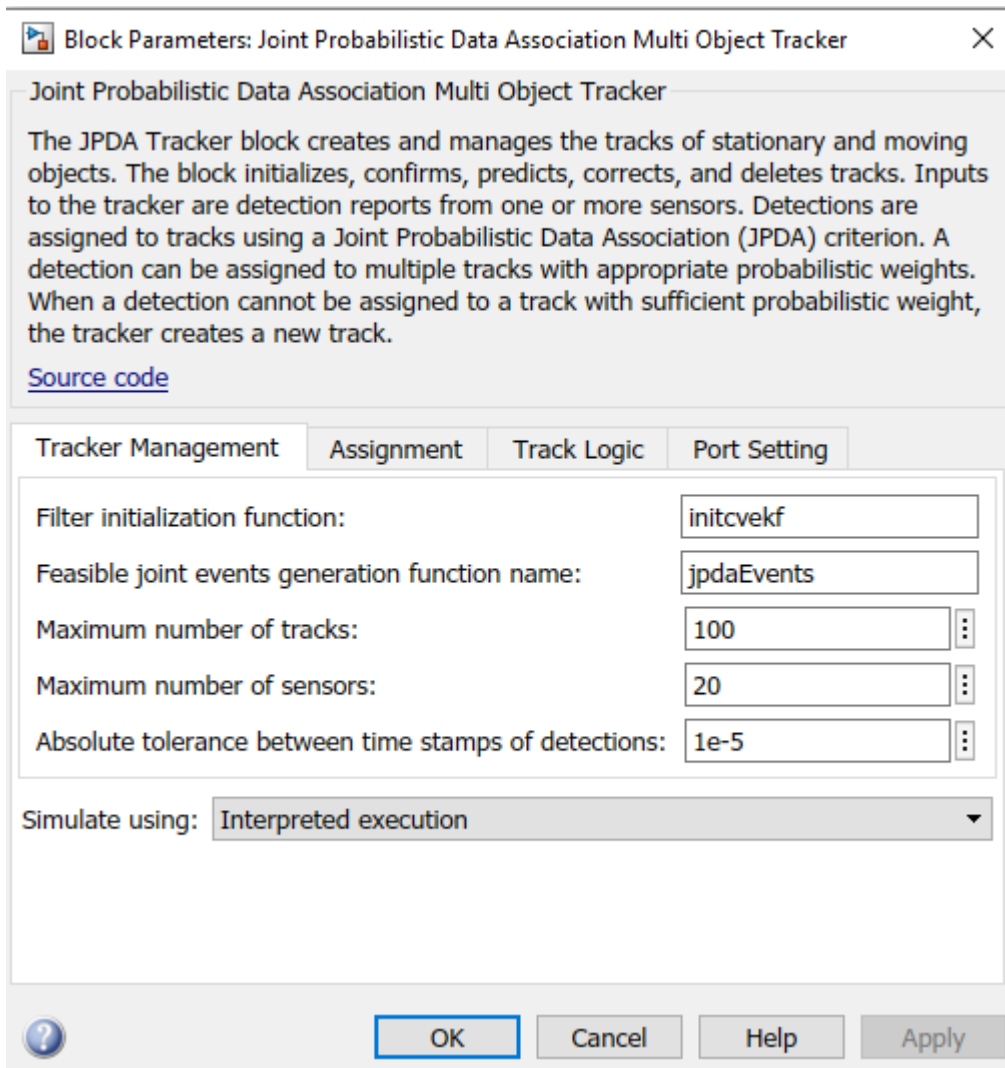
With `trackerJPDA`, you can customize your own filter initialization function and choose between the `History` and the `Integrated` track logic.

You can select your preferable subsystem by setting the value of conditional variable `Tracker` in the base workspace. The following table shows the `Tracker` values corresponding to their configurations.

Tracker	Selected Subsystem Configuration
1	GNN Tracker with extended Kalman filter
2	JPDA Tracker with extended Kalman filter

You can also use the "Edit and Manage Workspace Variables by Using Model Explorer" (Simulink) to change the value of `Tracker`.
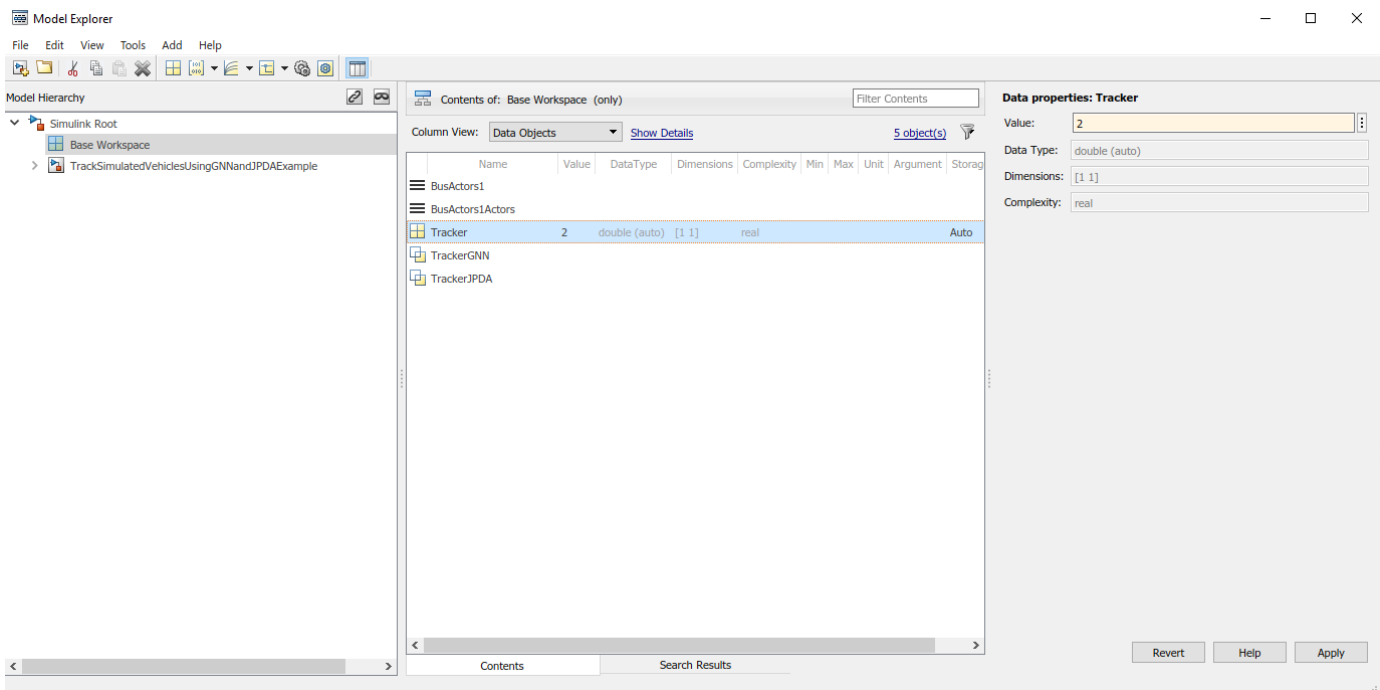
Track Metrics

The Track Metrics is implemented using a MATLAB System (Simulink) block. The code for the block is defined by a helper class, HelperTrackMetrics.

Results

To visualize the results, use the Bird's-Eye Scope. The Bird's-Eye Scope is a model-level visualization tool via a menu provided on the Simulink model toolbar. After opening the scope, click **Find Signals** to set up the signals. Then run the simulation to display the actors, vision and radar detections, tracks, and road boundaries. The following image shows the bird's-eye scope for this example.

In Simulink, you can run this example via interpreted execution or code generation. With interpreted execution, the model simulates the block using the MATLAB execution engine which allows faster startup time but longer execution time. With code generation, the model simulates the block using the subset of MATLAB code supported for code generation which allows better performance than the interpreted execution.

After running the model, you can visualize the results on the figures below.
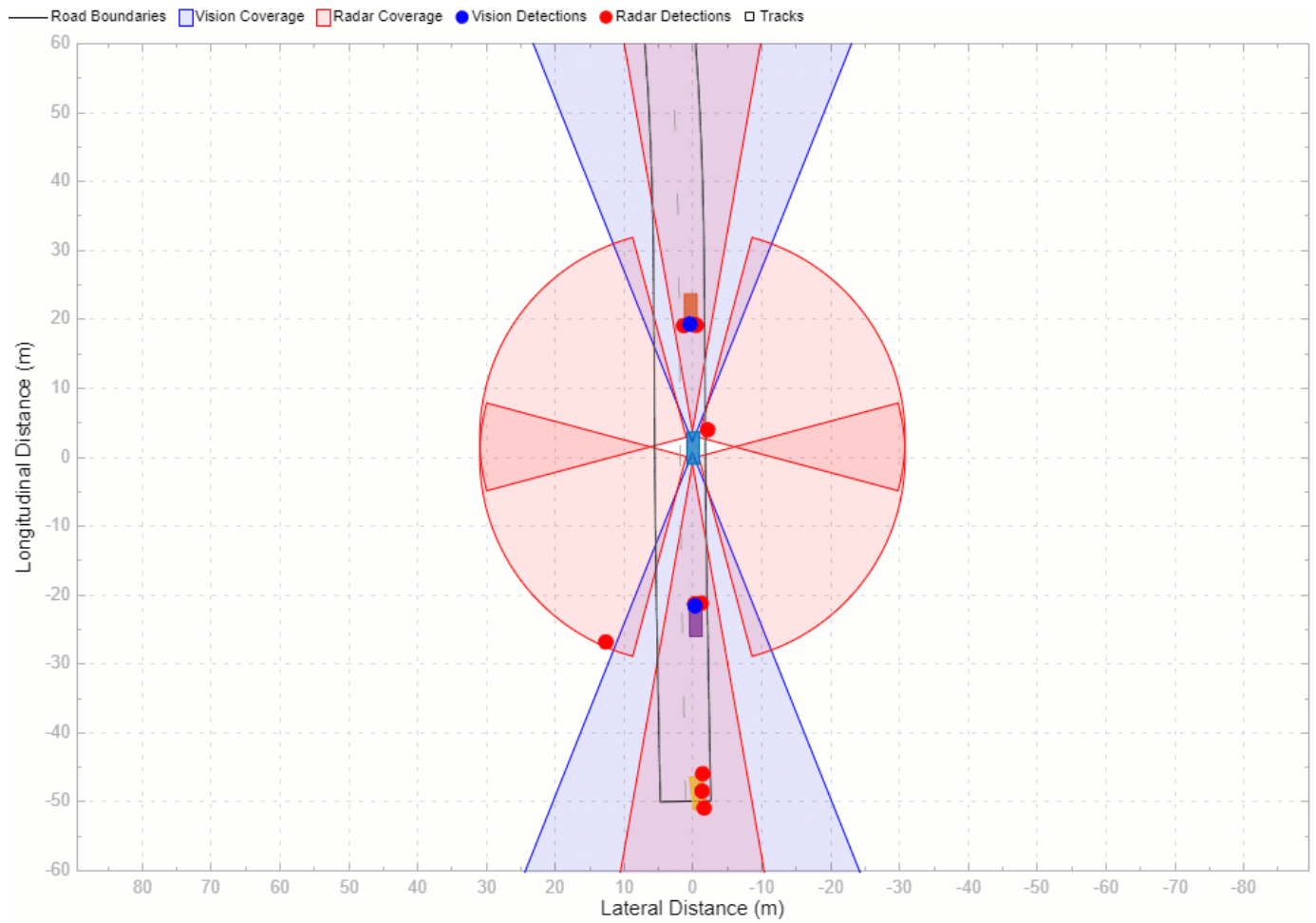
The above figure shows the tracking results using synthetic radar and vision data with `trackerGNN`. You can see that the tracker maintained tracks throughout the tracking region.
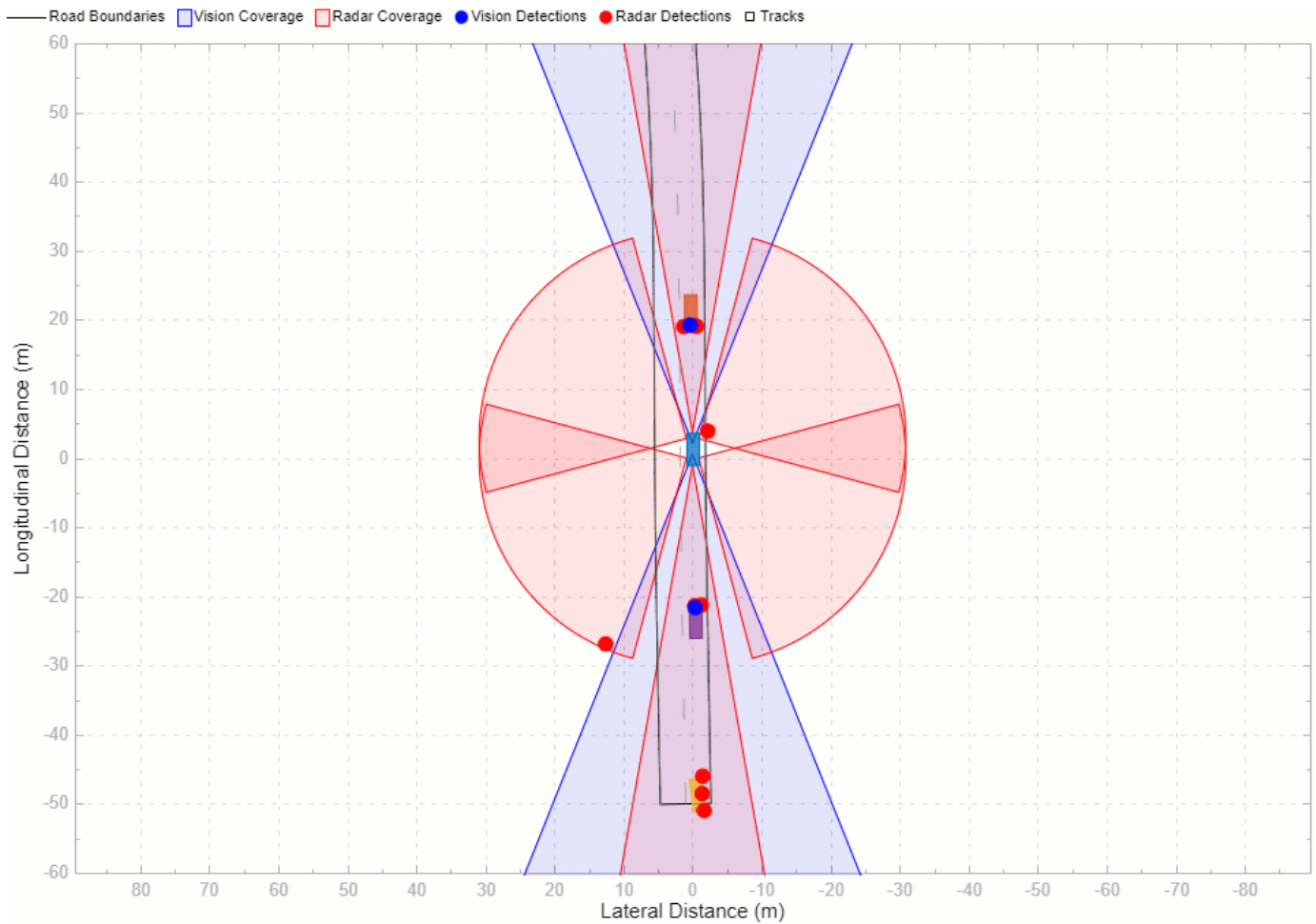
The above figure shows the tracking results using synthetic radar and vision data with `trackerJPDA`. You can see that the tracker maintained tracks throughout the tracking region and the tracking performance is better than that of `trackerGNN` as there are fewer false tracks generated.

Evaluate Tracking Performance

You can use the `Track Metrics` block to evaluate the tracking performance of each tracker using quantitative metrics. In this example, you view the number of target tracks, the number of redundant tracks, and the number of false tracks. A target track is a track that is associated with unique targets. A redundant track is a track that is associated with a ground truth object that has been associated to another track. A false track is a track that is not associated with any ground truth objects. Below the first figure shows the tracking performance results of GNN tracker and the second figure shows the results of JPDA tracker.
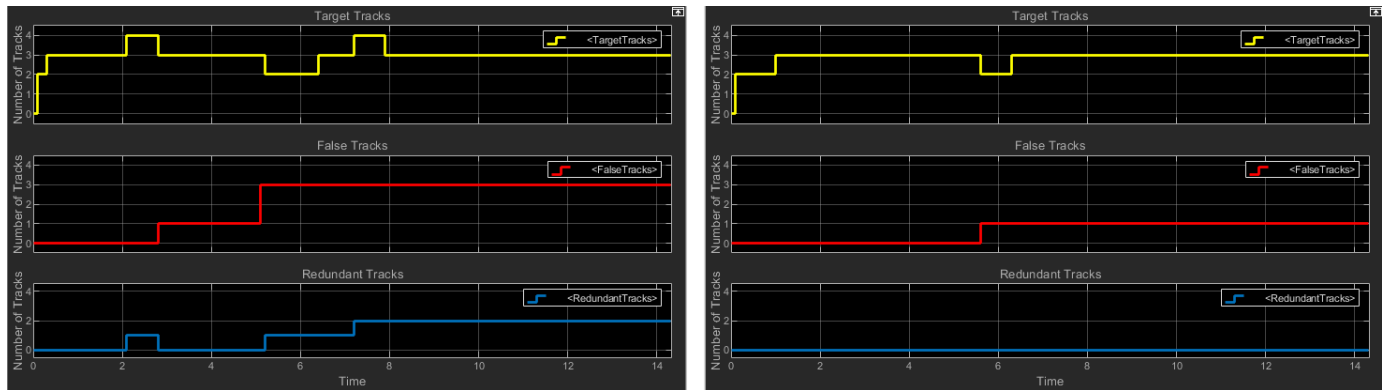
The assignment metrics illustrate that two redundant tracks were initialized and confirmed by `trackerGNN` whereas `trackerJPDA` does not create any redundant tracks. The redundant tracks were generated due to imperfect clustering, where detections belonging to the same target were clustered into more than one cluster. Also, the `trackerGNN` created and confirmed three false tracks whereas `trackerJPDA` confirmed only one false track. These metrics show that `trackerJPDA` provides better tracking performance than `trackerGNN`.

Summary

This example shows how to generate a scenario, simulate sensor detections, and use these detections to track moving vehicles around the ego vehicle using the `trackerGNN` and `trackerJPDA` blocks in Simulink. You have also seen the simplicity of exchanging between the two trackers, the flexibility of customizing these trackers to suit your own target tracking requirements, and the ability to evaluate the tracking results using track metrics.

Read and Parse NMEA Data Directly From GPS Receiver

This example shows you how to read the data from a GPS receiver connected to a computer, and parse the National Marine Electronics Association(NMEA) data.

NMEA Data from GPS Receiver

National Marine Electronics Association (NMEA) format is a specification that defines how data is transmitted between various marine electronic devices. Most of the GPS receiver output data in the NMEA format.

The data is transmitted in a sequence called a sentence. Each sentence is independent of the other sentences from the receiver. Sentences contain information such as latitude, longitude, speed, and time as ASCII characters. The data in a sentence are comma separated. The data is output on the serial port of the receiver at an interval called the `Update Rate`.

NMEA Sentence Format

StartChar	SentenceName	Separator	Data$_1$	Separator	Data$_2$	Separator	...	Data$_N$	CheckChar	Checksum	EndChar

NMEA Header	Value	Description
StartChar	$	ASCII for 36
SentenceName	For example, GPGGA	NMEA Sentence Identifier
Separator	,	ASCII for 44
Data_1 to Data_N	For example, 083445.00,1256.60109,N	Data fields, such as latitude and longitude
CheckChar	*	ASCII for 42
CheckSum	For example, 7E	Hexadecimal number representing 8-bit exclusive OR of all characters between $ and *
EndChar	CR LF	Carriage Return and Line Feed

The parser implemented in this example can be used to parse RMC, CGA, GSA, and VTG sentences. The message format for each of these sentences are explained below. The message formats are based on the protocol specification mentioned in the u-blox 6 Receiver Description.

RMC Message Format

$GPRMC,hhmmss,Status,Latitude,N,Longitude,E,SOG,COG,ddmmyy,MV,MVE,Mode*CS<CR><LF>

$GPRMC,083559.00,A,4717.11437,N,00833.91522,E,0.004,77.52,091202,,,A*57

Field No	Name	Example	Description
1	$GPRMC	$GPRMC	Message ID, RMC protocol header
2	hhmmss.ss	083559.00	UTC Time, Time of position fix, 08:35:59.00 UTC
3	status	A	Status, V – Navigation receiver warning, A – Data valid
4	Latitude	4717.11437, N	Latitude 47 deg 17.11437' North
5	Longitude	00833.91522, E	Longitude 8 deg 33.91522' East
6	SOG	0.004	Speed over Ground in Knots
7	COG	77.52	Course over ground in degrees
8	ddmmyy	091202	Date, 09th December 2002
9	MV, MVE	Receiver did not output these values	Magnetic Variation and Magnetic Variation E/W indicator
10	Mode	A	Mode indicator, (Only for NMEA version above 2.2) N – No Fix A – Autonomous GNSS Fix D – Differential GNSS Fix E – Estimated/Dead Reckoning Fix
11	CS	57	Checksum
12	CR LF		Carriage Return and Line Feed

GGA Message Format

$GPGGA,hhmmss.ss,Latitude,N,Longitude,E,FS,NoSV,HDOP,msl,m,Altref,m,DiffAge,DiffStation*cs<CR><LF>

$GPGGA,092725.00,4717.11399,N,00833.91590,E,1,8,1.01,499.6,M,48.0,M,,0*5B

Field No	Name	Example	Description
1	$GPGGA	$GPGGA	Message ID, GGA protocol header
2	hhmmss.ss	002725.00	UTC Time, Time of position fix, 09:27:25.00 UTC
3	Latitude	4717.11437, N	Latitude 47 deg 17.11437' North
4	Longitude	00833.91522, E	Longitude 8 deg 33.91522' East
5	FS (Fix Status)	1	Position fix status indicator 0 – Invalid 1 – GPS Fix 2 – DGPS Fix 6 – Estimated Fix
6	NoSV	8	Number of satellites used
7	HDOP	1.01	Horizontal Dilution of precision
8	Alt,m	499.6,M	MSL Altitude and its units (meters).The unit field is a fixed field
9	Altref,m	48.0,M	Height of geoid and its units(meters).The unit field is a fixed field
10	DiffAge	Receiver did not output these value	Age of Differential Corrections, Blank (Null) fields when DGPS is not used
11	DGPSstation	0	DGPS reference station ID
12	CS	5B	Checksum
13	CR LF		Carriage Return and Line Feed

GPVTG Message Format

$GPVTG,cogt,T,cogm,M,sog,N,kph,K,mode*cs <CR><LF>

$GPVTG,77.52,T,,M,0.004,N,0.008,K,A*06

Field No	Name	Example	Description
1	$GPVTG	$GPVTG	Message ID, VTG protocol header
2	COGt(True track made good)	77.52	Course over ground (true)
3	T	T	Fixed Field: True
4	COGm(Magnetic track made good)		Course over ground (magnetic)
5	M	M	Fixed Field: Magnetic
6	SOG	0.004	Speed over ground in knots
7	N	N	Fixed field: knots
8	kph	0.008	Speed over ground in kph
9	K	K	Fixed Field: kilometers per hour
10	Mode	A	Mode indicator, (Only for NMEA version above 2.2) N – No Fix A – Autonomous GNSS Fix D – Differential GNSS Fix E – Estimated/Dead Reckoning Fix
11	CS	06	Checksum
12	CR LF		Carriage Return and Line Feed

GSA Message Format

$GPGSA,Smode,FS{,sv},PDOP,HDOP,VDOP*cs<CR><LF>

$GPGSA,A,3,23,29,07,08,09,18,26,28,,,,,1.94,1.18,1.54*OD

Field No	Name	Example	Description
1	$GPGSA	$GPGSA	Message ID, GSA protocol header
2	Smode(Selection of Fix)	A	M – Manually forced to operate in 2D or 3D mode A – Allowed to automatically switch 2D/3D mode
3	Fix Status	3	1 – Fix not available 2 – 2D Fix available 3 – 3D Fix available
4	sv(PRNs)		Satellite number (12 fields)
5	PDOP	1.94	Position dilution of precision
6	HDOP	1.18	Horizontal dilution of precision
7	VDOP	1.54	Vertical Dilution of precision
8	CS	57	Checksum
9	CR LF		Carriage Return and Line Feed

Hardware Setup

Required Hardware

- Ublox NEO-6M GPS Module
- Serial to USB signal converters like FTDI (If the GPS module cannot be directly connected to computer using USB)
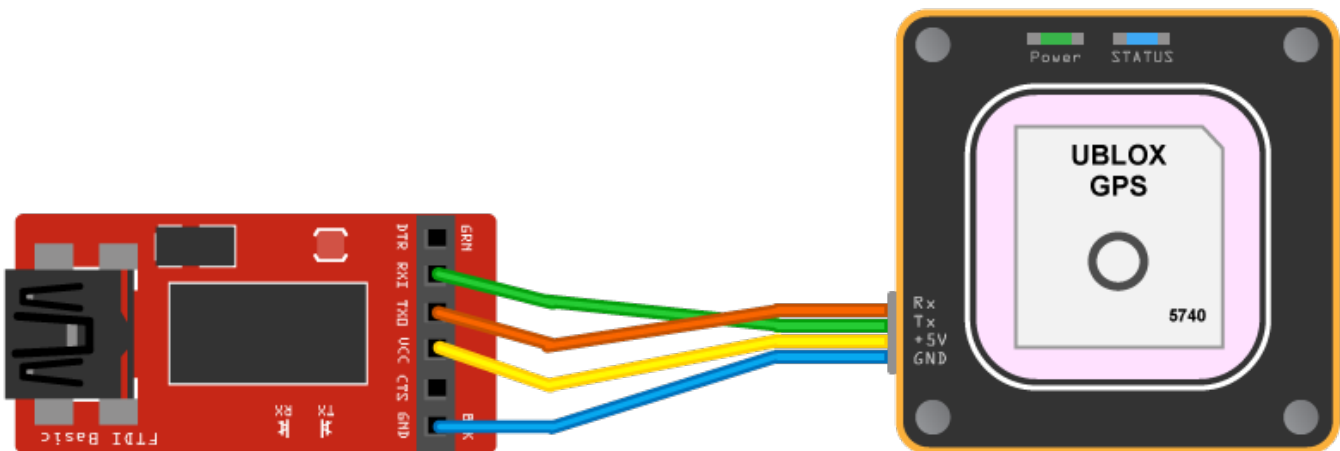
If a USB connector is available on the GPS receiver, connect the GPS module directly to the computer via USB Cable and install the required drivers. Otherwise, use serial to USB converters, to convert the serial transmissions to USB signals.

Create Serial Port Object

Create a `gpsReceiver` object and associate it with the serial port on which GPS module is connected. The serial data from the GPS module follows 9600 8N1 format.

```
ublox_gps = gpsReceiver('COM17');
```

```
ublox_gps =

  gpsReceiver with properties:

        Port: 'COM17'
    BaudRate: 9600
```

GPS outputs data in multiple sentences which follows NMEA format. The `read` method of the `gpsReceiver` class returns frames of raw GPS data. The start of each frame is assumed as an RMC sentence and the end of a frame is assumed to be the start of the next RMC sentence. The number of frames returned can be specified as an argument to the `read` function. By default, the function returns one frame of NMEA data. Make sure the Ublox-6M module is configured to its default state, where the `Baud Rate` is 9600, `Update Rate` is 1 hz and the module outputs the messages corresponding to RMC, GSA, GGA, and VTG IDs.

```
rawGPSData = read(ublox_gps);
```

For multiple number of frames, specify the number of NMEA frames as an argument in the `read` function.

```
NumOfFrames = 3;
rawGPSDataWithFrames = read(ublox_gps, NumOfFrames);
```

Parse GPS Data

The `HelperParseGPSData` function in the example folder can be used to parse the GPS Data. The implemented parser is capable of decoding RMC, GGA, GSA, and VTG Sentences. The parser function returns a structure whose fields correspond to the data decoded from the corresponding sentence. If the receiver does not output a value, the corresponding checksum field in the structure will be empty. The `ChecksumValidity` field determines the validity of the received data.

The below code can be used to parse for RMC, GGA, GSA, and VTG sentences.

```
[GPRMC, GPGGA, GPVTG, GPGSA] = HelperParseGPSData(rawGPSData);
```

```
GPRMC =

  struct with fields:

             MessageID: 'RMC'
                Status: 'A,Data Valid'
           DateandTime: '18:17:34.00 22/05/19 UTC'
              Latitude: '12.9604°,N'
             Longitude: '77.7001°,E'
                   SOG: '0.177 knots'
                   COG: ''
     MagneticVariation: ''
                  Mode: 'Autonomous GNSS Fix'
              Checksum: '72'
       ChecksumValidity: 'Correct Checksum'


GPGGA =

  struct with fields:

             MessageID: 'GGA'
                  Time: '18:17:34.00 UTC'
              Latitude: '12.9604°,N'
             Longitude: '77.7001°,E'
             FixStatus: '1 GPS Fix'
          NumSatelites: 9
                  HDOP: 0.9000
              Altitude: '913.0 m'
         HeightofGeoid: '-86.4 m'
               DiffAge: ''
          DGPSStationID: ''
              Checksum: '7E'
       ChecksumValidity: 'Correct Checksum'
```

```
GPVTG =

  struct with fields:

                MessageID: 'VTG'
        TrueTrackMadeGood: ''
     MagneticTrackMadeGood: ''
        GroundSpeedinKnots: '0.177 knots'
        GroundSpeedinKmph: '0.327 km/hr'
                     Mode: 'A,Autonomous GNSS Fix'
                 Checksum: '24'
         ChecksumValidity: 'Correct Checksum'

    '

GPGSA =

  struct with fields:

              MessageID: 'GSA'
          SelectionofFix: 'A Automatic'
                FixType: '3,3D Fix'
                    PRNs: [20 27 32 25 21 24 31 10 8]
                    PDOP: 1.5200
                    HDOP: 0.9000
                    VDOP: 1.2200
                Checksum: '03'
        ChecksumValidity: 'Correct Checksum'
```

If there are multiple NMEA sentences available in the input raw GPS Data, an array of structures is returned as output. The size of the array will be the number of available NMEA sentences corresponding to the message IDs.

If an NMEA sentence corresponding to a message ID is not available in the input data, the output structure corresponding to that message ID will return null fields except for MessageID field in the structure. These outputs can be suppressed using tilde operator.

```
GPSData = '$GPGGA,135015.00,1257.62410,N,07742.00316,E,1,06,1.56,903.3,M,-86.4,M,,*77';
[~,GPGGA,~,~] = HelperParseGPSData(GPSData);
```

```
GPGGA =

  struct with fields:

            MessageID: 'GGA'
                 Time: '13:50:15.00 UTC'
             Latitude: '12.9604°,N'
            Longitude: '77.7001°,E'
            FixStatus: '1 GPS Fix'
        NumSatelites: 6
                 HDOP: 1.5600
             Altitude: '903.3 m'
        HeightofGeoid: '-86.4 m'
              DiffAge: ''
        DGPSStationID: ''
             Checksum: '77'
     ChecksumValidity: 'Correct Checksum'
```

The `flush` method of the object can be used to flush the input buffers associated with the `gpsReceiver` object.

```
flush(ublox_gps);
```

Clean Up

When the connection is no longer required, disconnect and delete the object.

```
delete(ublox_gps);
```

To get help on parser function, use

```
help HelperParseGPSData;
```

To get help on gpsReceiver object, use

```
help gpsReceiver;
```

Track-to-Track Fusion for Automotive Safety Applications in Simulink

This example shows how to perform track-to-track fusion in Simulink® with Sensor Fusion and Tracking Toolbox™. In the context of autonomous driving, the example illustrates how to build a decentralized tracking architecture using a track fuser block. In the example, each vehicle perform tracking independently as well as fuse tracking information received from other vehicles. This example closely follows the "Track-to-Track Fusion for Automotive Safety Applications" on page 5-424 MATLAB® example.

Introduction

Automotive safety applications largely rely on the situational awareness of the vehicle. A better situational awareness provides the basis to a successful decision-making for different situations. To achieve this, vehicles can benefit from intervehicle data fusion. This example illustrates the workflow in Simulink for fusing data from two vehicles to enhance situational awareness of the vehicle.

Setup and Overview of the Model
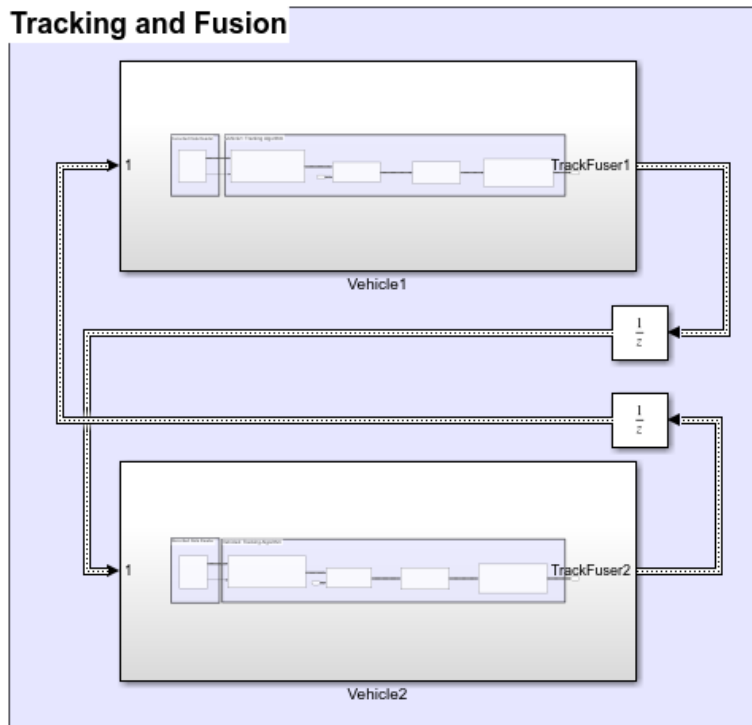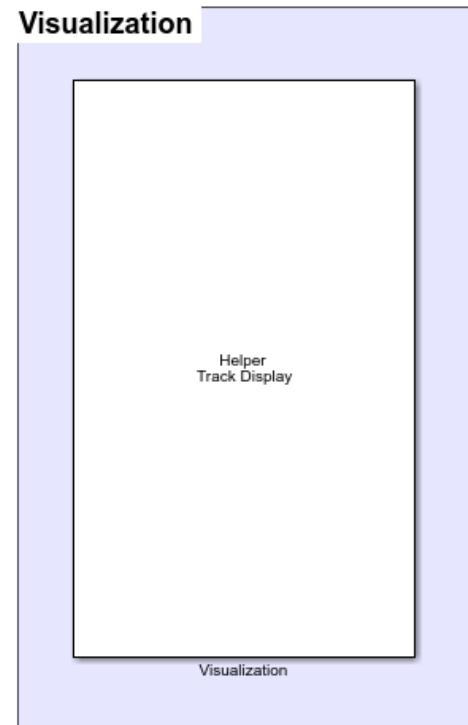
Prior to running this example, the `drivingScenario` object was used to create the same scenario defined in "Track-to-Track Fusion for Automotive Safety Applications" on page 5-424. The detections and time data of objects detected from the sensors of `Vehicle1` and `Vehicle2` in the scenario were then saved to the data files `v1Data.mat` and `v2Data.mat`, respectively. Also, the pose information of vehicles were saved in files `v1Pose.mat` and `v2Pose.mat`.

Tracking and Fusion

In Tracking and Fusion section of the model there are two subsystems which implements the target tracking and fusion capabilities of `Vehicle1` and `Vehicle2` in this scenario.

Vehicle1
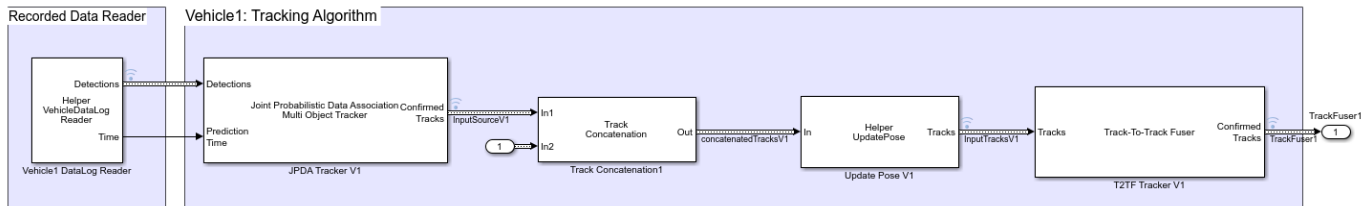
This subsystem includes `Vehicle1 DataLog Reader` block that reads the prerecorded detection and time data form `v1Data.mat` file for `Vehicle1`. This data is then passed to the `JPDA Tracker V1` block which processes the detections to generate a list of tracks. The tracks are then passed into a `Track Concatenation1` block, which concatenates these input tracks. The first input to the `Track Concatenation1` block is the local tracks from the JPDA tracker and the second input is the tracks received from the other vehicle's track fuser. To transform local tracks to central tracks, the track fuser needs the parameter information about the local tracks. However, this information is not available from the direct outputs of the JPDA tracker. Therefore, a helper Update Pose block is used to supply these information by reading the data from the v1Pose.mat file. The updated tracks are then broadcasted to `T2TF Tracker V1` block as an input. Finally, the `trackFuser T2TF Tracker V1` block fuses the local vehicle tracks with the tracks received from the other vehicle's track fuser. After each update, the track fuser on each vehicle broadcasts its fused tracks to be fed into the update of the other vehicle's track fuser in the next time stamp.

Vehicle2
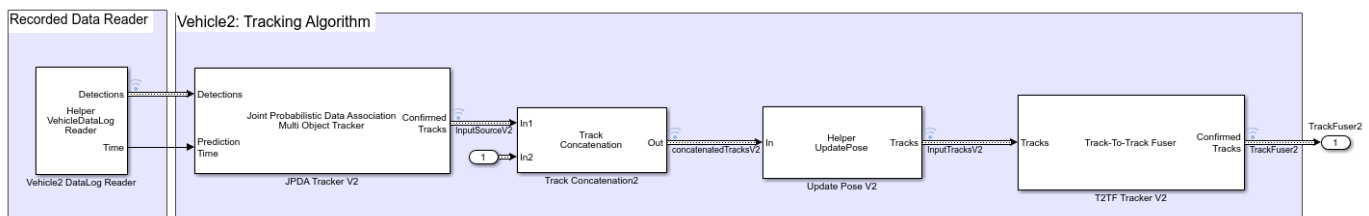
`Vehicle2` subsystem follows similar setup as `Vehicle1` subsystem as described above.

Visualization

The Visualization block is implemented using the MATLAB System block and is defined using `HelperTrackDisplay` block. The block uses `RunTimeObject` parameter of the blocks to display their outputs. See "Access Block Data During Simulation" (Simulink) for further information on how to access block outputs during simulation.

Results

After running the model, you visualize the results as on the figure. The animation below shows the results for this simulation.

The visualization includes two panels. The left panel shows the detections, local tracks, and fused tracks that `Vehicle1` generated during the simulation and represents the situational awareness of the `Vehicle1`. The right panel shows the situational awareness of `Vehicle2`.

The recorded detections are represented by black circles. The local and fused tracks from `Vehicle1` and `Vehicle2` are represented by square and diamond respectively. Notice that during the start of simulation, `Vehicle1` detects vehicles parked on the right side of the street, and tracks associated with the parked vehicles are confirmed. Currently `Vehicle2` only detects `Vehicle1` which is immediately in front of it. As the simulation continues the confirmed tracks from `Vehicle1` are broadcasts to the fuser on `Vehicle2`. After fusing the tracks, `vehicle2` becomes aware of the objects prior to detecting these objects on its own. Similarly, `Vehicle2` tracks are broadcasts to `Vehicle1`. `Vehicle1` fuses these tracks and becomes aware of the objects prior to detecting them on its own.

In particular, you observe that the pedestrian standing between the blue and purple car on the right side of the street is detected and tracked by `Vehicle1`. `Vehicle2` first becomes aware of the pedestrian by fusing the track from `Vehicle1` at around 2.5 seconds. It takes `Vehicle2` roughly 2 seconds before it starts detecting the pedestrian using its own sensor. The ability to track a pedestrian based on inputs from `Vehicle1` allows `Vehicle2` to extend its situational awareness and to mitigate the risk of accident.
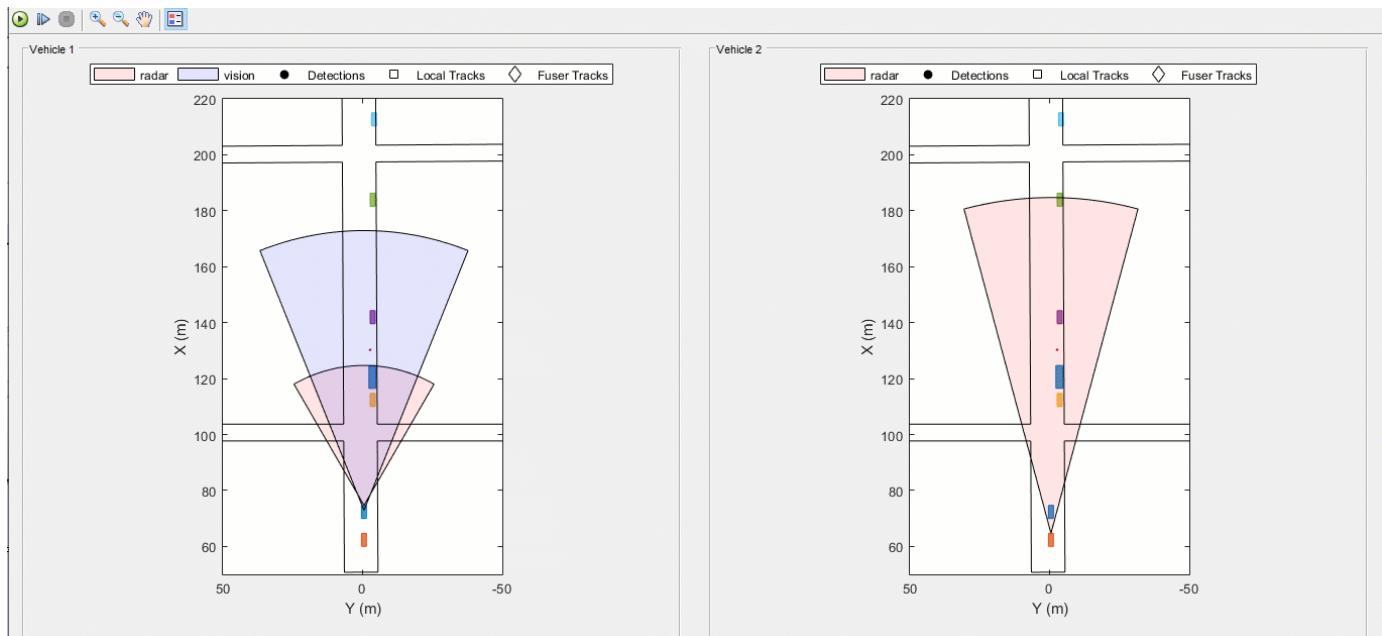
Summary

This example showed how to perform track-to-track fusion in Simulink. You learned how to perform tracking using a decentralized tracking architecture, where each vehicle is responsible for maintaining its own local tracks, fuse tracks from other vehicles, and communicate the tracks to the other vehicle. You also used a JPDA tracker block to generate the local tracks.

Track Point Targets in Dense Clutter Using GM-PHD Tracker

This example shows you how to track points targets in dense clutter using a Gaussian mixture probability hypothesis density (GM-PHD) tracker using a constant velocity model.

Setup Scenario

The scenario used in this example is created using `trackingScenario`. The scenario consists of five point targets moving at a constant velocity. The targets move within the field of view of a static 2-D radar sensor. You use the `monostaticRadarSensor` to simulate the 2-D sensor and mount it on a static platform. You use the `FalseAlarmRate` property of the sensor to control the density of the clutter. The value of the `FalseAlarmRate` property represents the probability of generating a false alarm in one resolution cell of the sensor. Based on a false alarm rate of 10^{-3} and the resolution of the sensor defined in this example, there are approximately 48 false alarms generated per step.

```matlab
% Reproducible target locations
rng(2022);

% Create a trackingScenario object
scene = trackingScenario('UpdateRate',10,'StopTime',10);
numTgts = 5;

% Initialize position and velocity of each target
x = 100*(2*rand(numTgts,1) - 1);
y = 100*(2*rand(numTgts,1) - 1);
z = zeros(numTgts,1);
vx = 5*randn(numTgts,1);
vy = 5*randn(numTgts,1);
vz = zeros(numTgts,1);

% Add platforms to scenario with given positions and velocities.
for i = 1:numTgts
    thisTgt = platform(scene);
    thisTgt.Trajectory.Position = [x(i) y(i) z(i)];
    thisTgt.Trajectory.Velocity = [vx(i) vy(i) vz(i)];
end

% Add a detecting platform to the scene.
detectingPlatform = platform(scene);
detectingPlatform.Trajectory.Position = [-200 0 0];

% Simulate 2-D radar using a monostaticRadarSensor.
radar = monostaticRadarSensor(1,...
    'UpdateRate',scene.UpdateRate,...
    'DetectionProbability',0.9,...% Pd
    'FalseAlarmRate',1e-3,...% Pfa
    'FieldOfView',[120 1],...
    'ScanMode','No Scanning',...
    'DetectionCoordinates','scenario',...% Report in scenario frame
    'HasINS',true,...                % Enable INS to report in scenario frame
    'ReferenceRange',300,...         % Short-range
    'ReferenceRCS',10,...            % Default RCS of platforms
    'MaxUnambiguousRange',400,...    % Short-range
    'RangeResolution',1,...
    'AzimuthResolution',1,...
    'HasFalseAlarms',true);          % Reports false alarms
```

```
% Mount the radar on the detecting platform.
detectingPlatform.Sensors = radar;
```
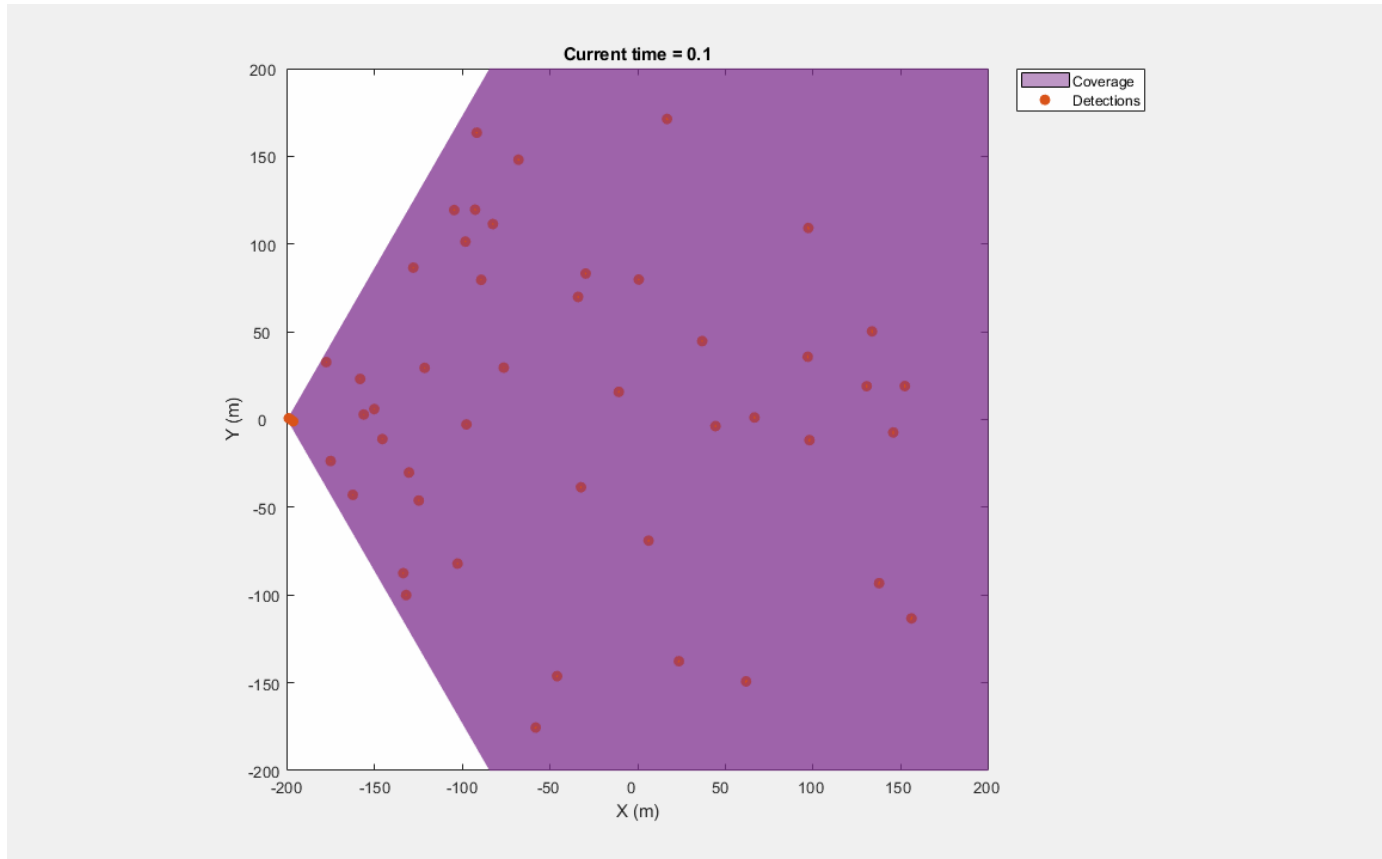

Set Up Tracker and Metrics

Tracker

You use a GM-PHD point object tracker to track targets. The first step towards configuring a PHD tracker is to set up the configuration of the sensor. You define the configuration using a `trackingSensorConfiguration` object.

The `SensorIndex` of the configuration is set to 1 to match that of the simulated sensor. As the sensor is a point object sensor that outputs at most one detection per object per scan, you set the `MaxNumDetsPerObject` property of the configuration to 1.

The `SensorTransformFcn`, `SensorTransformParameters,` and `SensorLimits` together allow you to define the region in which the tracks can be detected by the sensor. The `SensorTransformFcn` defines the transformation of a track state (x_{track}) into an intermediate space used by the sensor (x_{sensor}) to define track detectability. The overall calculation to calculate detection probability is shown below:

$$x_{\text{sensor}} = \text{SensorTransformFcn}(x_{\text{track}}, \text{SensorTransformParameters})$$

$$P_d =$$

$$\begin{cases} \text{Configuration.DetectionProbability} & \text{SensorLimits}(:,1) \le x_{\text{sensor}} \le \text{SensorLimits}(:,2) \\ \text{Configuration.MinDetectionProbability} & \text{otherwise} \end{cases}$$

To compute detection probability of an uncertain state with a given state covariance, the tracker generates samples of the state using sigma-point calculations similar to an unscented Kalman filter.

Notice that the signature of the `SensorTransformFcn` is similar to a typical measurement model. Therefore, you can use functions like `cvmeas`, `cameas` as `SensorTransformFcn`. In this example, you assume that all tracks are detectable. Therefore, the `SensorTransformFcn` is defined as `@(x,params)x` and SensorLimits are defined as `[-inf inf]` for all states.

```
% Transform function and limits
sensorTransformFcn = @(x,params)x;
sensorTransformParameters = struct;
sensorLimits = [-inf inf].*ones(6,1);

% Detection probability for sensor configuration
Pd = radar.DetectionProbability;

config = trackingSensorConfiguration('SensorIndex',1,...
    'IsValidTime',true,...% Update every step
    'MaxNumDetsPerObject',1,...
    'SensorTransformFcn',sensorTransformFcn,...
    'SensorTransformParameters',sensorTransformParameters,...
    'SensorLimits',sensorLimits,...
    'DetectionProbability',Pd);
```

The `ClutterDensity` property of the configuration refers to false alarm rate per-unit volume of the measurement-space. In this example, the measurement-space is defined as the Cartesian coordinates as the detections are reported in the scenario frame. As the volume of the sensor's resolution in Cartesian coordinates changes with azimuth and range of the resolution, an approximate value can be computed at the mean-range of the sensor.

```
% Sensor Parameters to calculate Volume of the resolution bin
Rm = radar.MaxUnambiguousRange/2; % mean -range
dTheta = radar.ElevationResolution;
% Bias fractions reduce the "effective" resolution size
dPhi = radar.AzimuthBiasFraction*radar.AzimuthResolution;
dR = radar.RangeBiasFraction*radar.RangeResolution;

% Cell volume
VCell = 2*((Rm+dR)^3 - Rm^3)/3*(sind(dTheta/2))*deg2rad(dPhi);

% False alarm rate
Pfa = radar.FalseAlarmRate;

% Define clutter density
config.ClutterDensity = Pfa/VCell;
```

You also define a `FilterInitializationFcn` to specify the type of filter and the distribution of components in the filter, initialized by this sensor. In this example, you set the `FilterInitializationFcn` to `initcvgmphd`, which creates a constant-velocity GM-PHD filter and adds 1 component per low-likelihood detection from the tracker. The `initcvgmphd` does not add any component when called without a detection. This means that under this configuration, the birth components are only added to the filter when detections fall outside of the `AssignmentThreshold` of multi-target filer. See `AssignmentThreshold` property of `trackerPHD` for more details.

```
config.FilterInitializationFcn = @initcvgmphd;
```

Next you create the tracker using this configuration using the `trackerPHD` System object™. While configuring a tracker, you specify the `BirthRate` property to define the number of targets appearing in the field of view per unit time. The `FilterInitializationFcn` used with the configuration adds one component per unassigned detection. At each time-step, you can expect the number of components to be approximately equal to the number of false alarms and new targets. The tracker distributes the `BirthRate` to all these components uniformly.

```
% Use number of radar cells to compute number of false alarms per unit time.
NCells = radar.MaxUnambiguousRange/radar.RangeResolution*radar.FieldOfView(1)/radar.AzimuthResolu
numFalse = Pfa*NCells;
```

```
% Choose an initial weight of 0.05 for each new component. As number of new
```
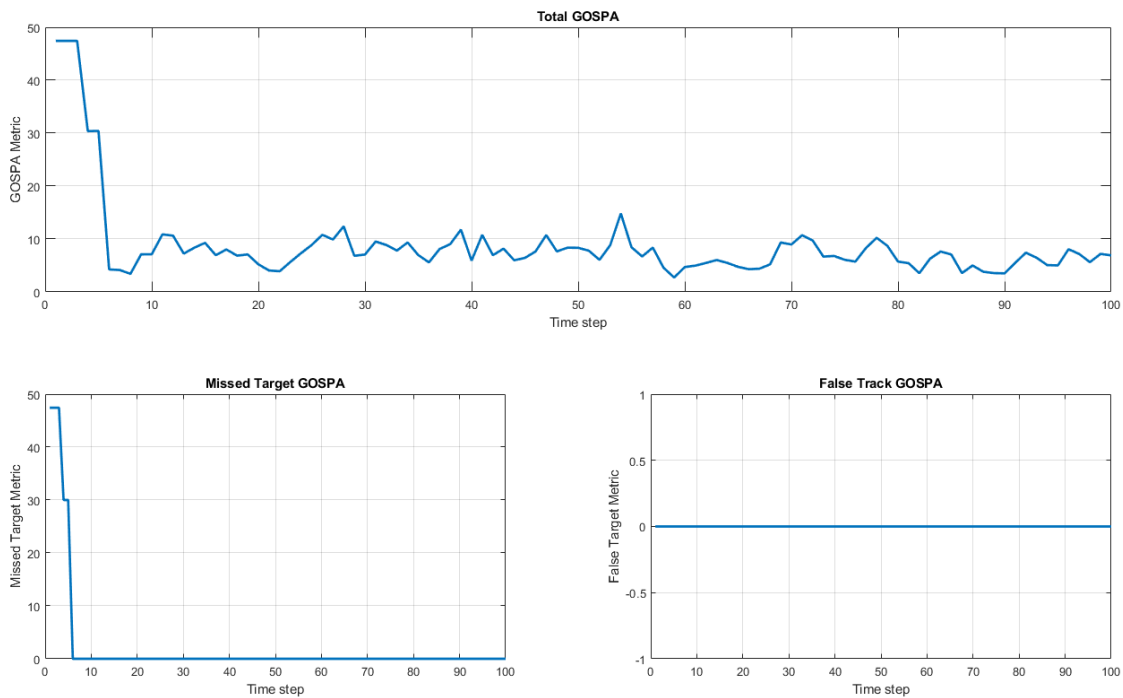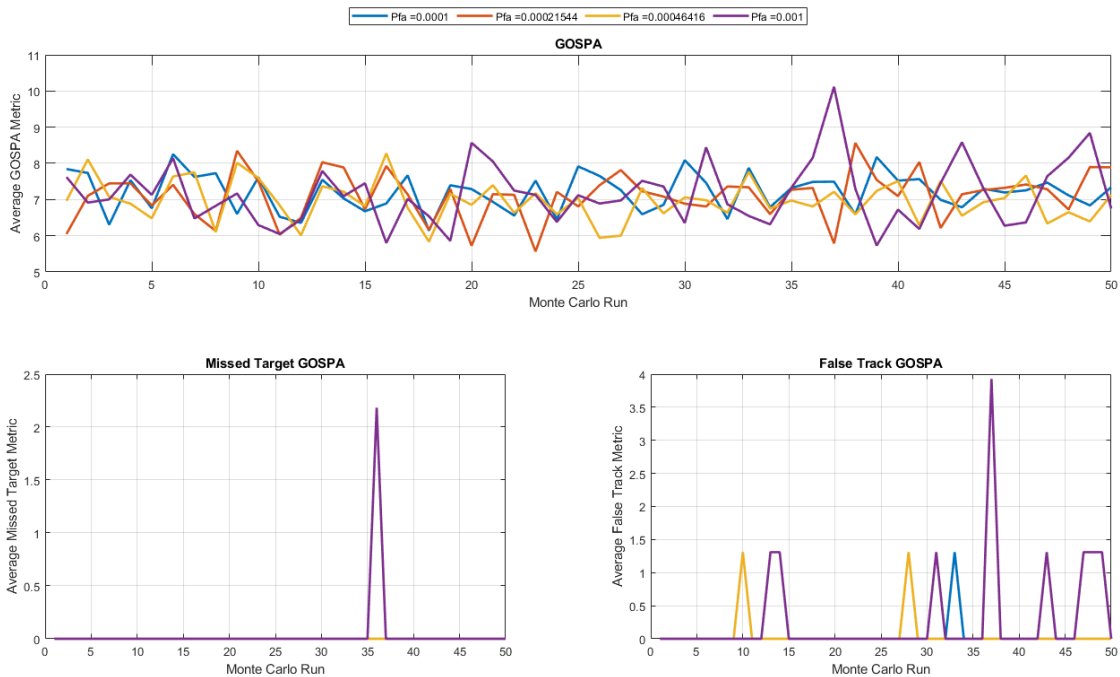
```
% targets is not known, new number of components are simply used as number
% of false alarms. 0.1 is the update rate.
birthRate = 0.05*(numFalse)/0.1;

% Create a tracker with the defined sensor configuration.
tracker = trackerPHD('BirthRate',birthRate,...
    'SensorConfigurations', config);
```

Metrics

To evaluate the performance of the tracker, you also set up a metric for performance evaluation. In this example, you use the Generalized Optimal Subpattern Assignment (GOSPA) metric. The GOSPA metric aims to evaluate the performance of a tracker by assigning it a single cost value. The better the tracking performance, the lower the GOSPA cost. A value of zero represents perfect tracking.

```
% Create gospa metric object.
gospa = trackGOSPAMetric;

% Initialize variable for storing metric at each step.
gospaMetric = zeros(0,1);
loc = zeros(0,1);
mt = zeros(0,1);
ft = zeros(0,1);
```

Run Simulation

Next, you advance the scenario, collect detections from the scenario, and run the PHD tracker on the simulated detections.

```
% Create a display
display = helperClutterTrackingDisplay(scene);

count = 1; % Counter for storing metric data
rng(2018); % Reproducible run

while advance(scene)
    % Current time
    time = scene.SimulationTime;

    % Current detections
    detections = detect(scene);

    % Tracks
    tracks = tracker(detections, time);

    % Update display
    display(scene, detections, tracks);

    % Compute GOSPA. getTruth function is defined below.
    truths = getTruth(scene);
    [~,gospaMetric(count),~,loc(count),mt(count),ft(count)] = gospa(tracks, truths);
    count = count + 1;
end
```
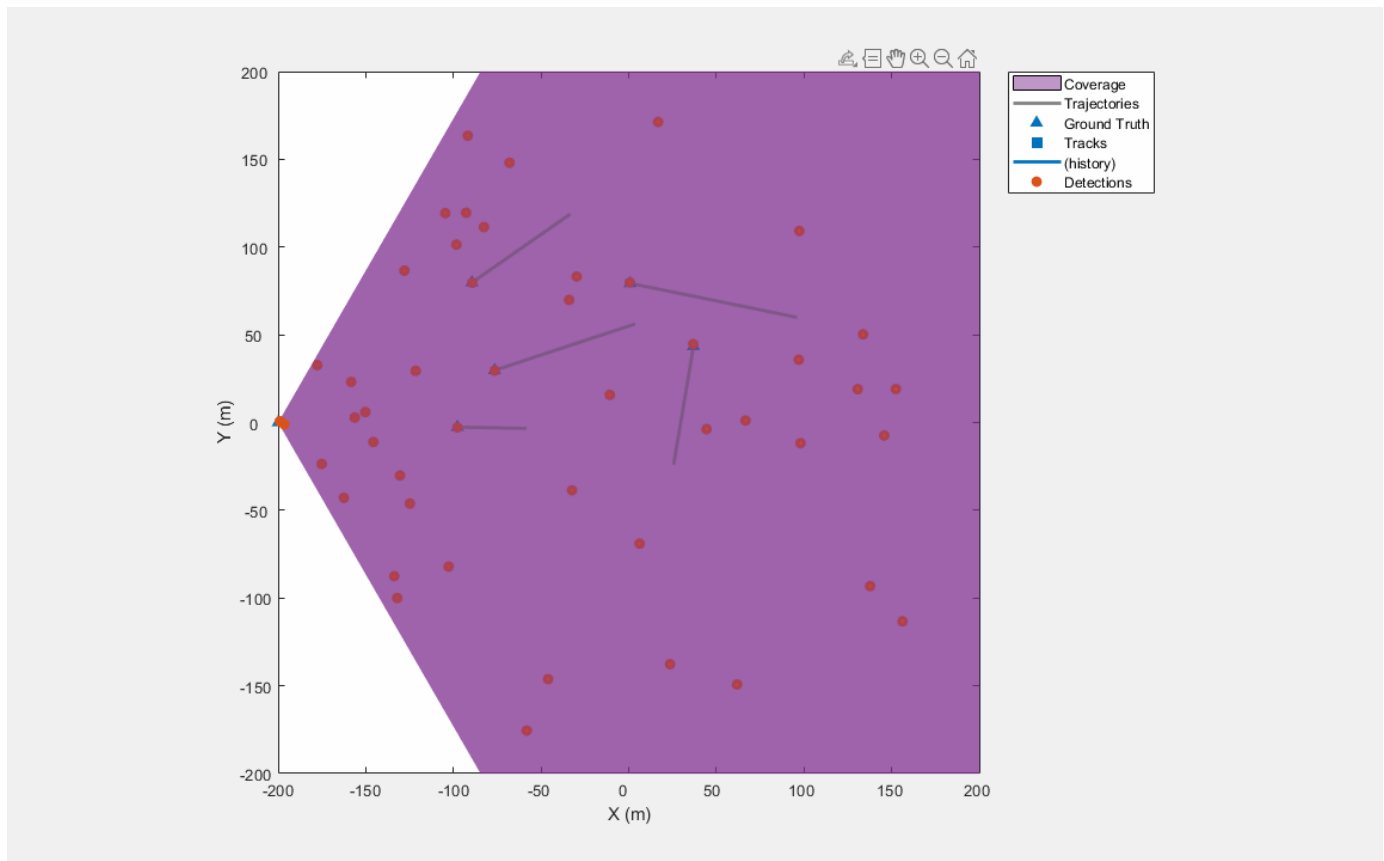

You can visualize that all targets were tracked by the PHD tracker in this scenario. This can also be quantitatively evaluated using the GOSPA metric and associated components. In the figure below,

notice that GOSPA metric goes down after a few steps. The initial value of GOSPA metric is higher because of establishment delay for each track.

```
figure('Units','normalized','Position',[0.1 0.1 0.8 0.8])
subplot(2,2,[1 2]); plot(gospaMetric,'LineWidth',2); title('Total GOSPA');
ylabel('GOSPA Metric'); xlabel('Time step'); grid on;
subplot(2,2,3); plot(mt,'LineWidth',2); title('Missed Target GOSPA');
ylabel('Missed Target Metric'); xlabel('Time step'); grid on;
subplot(2,2,4); plot(ft,'LineWidth',2); title('False Track GOSPA');
ylabel('False Target Metric'); xlabel('Time step'); grid on;
```

Analyze Performance

A typical method to qualify the performance of a tracker is by running several simulations on different realizations of the scenario. Monte Carlo simulations help to nullify the effect of random events such as locations of false alarms, events of target misses, and noise in measurements.

In this section, you run different realizations of the scenario and the tracker with different false alarm rates and compute the average GOSPA of the system for each realization. The process of running a scenario and computing the average GOSPA for the system is wrapped inside the helper function helperRunMonteCarloAnalysis on page 5-0 . To accelerate the Monte Carlo simulations, you can generate code for the `monostaticRadarSensor` model as well as the `trackerPHD` using MATLAB® Coder™ Toolbox. The process for generating code is wrapped inside the helper function helperGenerateCode on page 5-0 . To generate code for an algorithm, you assemble the code into a standalone function. This function is named `clutterSimTracker_kernel` in this example. The `clutterSimTracker_kernel` function is written to support four false alarm rates. To regenerate code with different false alarm rates, you can use the following command.

```
Pfas = 10.^(-4,-3,4); % Choose your false alarm settings
helperGenerateCode(scene, Pfas); % Generate code
```

To regenerate code for more false alarm settings, you can modify the functions to include more persistent variables. For more details on how to generate code for System objects™, refer to the "How to Generate C Code for a Tracker" on page 5-313 example.

The `helperRunMonteCarloAnalysis` function uses a `parfor` to execute each Monte Carlo run. If you have Parallel Computing Toolbox™ license, the Monte Carlo runs can be distributed across several workers to further accelerate the simulations.

```
% Turn on flag to run Monte Carlo Simulations
Pfas = 10.^linspace(-4,-3,4);
runMonteCarlo = false;
if runMonteCarlo
    numRunsPerPfa = 50; %#ok<UNRCH>
    gospaMCs = zeros(4,numRunsPerPfa,4);
    for i = 1:4
        gospaMCs(i,:,:) = helperRunMonteCarloAnalysis(scene, Pfas, Pfas(i), numRunsPerPfa);
    end
    save clutterTrackingMCRuns.mat gospaMCs;
else
    % Reload results from last run
    load('clutterTrackingMCRuns.mat','gospaMCs');
end

% Plot results
figure('Units','normalized','Position',[0.1 0.1 0.8 0.8])
subplot(2,2,[1 2]); plot(gospaMCs(:,:,1)','LineWidth',2); title('GOSPA');
```

```
legend(strcat('Pfa = ',string(Pfas)),'Orientation','horizontal','Location','NorthOutside');
ylabel('Average GOSPA Metric'); xlabel('Monte Carlo Run'); grid on;
subplot(2,2,3); plot(gospaMCs(:,:,3)','LineWidth',2); title('Missed Target GOSPA');
ylabel('Average Missed Target Metric'); xlabel('Monte Carlo Run'); grid on;
subplot(2,2,4); plot(gospaMCs(:,:,4)','LineWidth',2); title('False Track GOSPA');
ylabel('Average False Track Metric'); xlabel('Monte Carlo Run'); grid on;
```

The plots above show performance of the tracker in this scenario by running 50 Monte Carlo realizations for each false alarm rate. As the false alarm rate increases, the probability of generating a false track increases. This probability is even higher in the vicinity of the sensor, where density of resolution cells is much higher. As false alarms are closely spaced and appear frequently in this region, they can be misclassified as a low-velocity false-track. This behavior of the tracker can be observed in the averaged "False Track Component" of the GOSPA metric per scenario run. Notice that as the false alarm rate increases, the number of peaks in the plot also increase. This also causes an increase in the total GOSPA metric. The "Missed Target Component" is zero for all except one run. This type of event is caused by multiple misses of the target by the sensor.

Summary

In this example, you learned how to configure and initialize a GM-PHD tracker to track point targets for a given false alarm rate. You also learned how to evaluate the performance of the tracker using the GOSPA metric and its associated components. In addition, you learned how to run several realizations of the scenario under different false alarm settings to qualify the performance characteristics of the tracker.

Utility Functions

helperRunMonteCarloAnalysis

```
function gospaMC = helperRunMonteCarloAnalysis(scene, Pfas, Pfai, numRuns)

clear clutterSimTracker_kernel;

% Compute which tracker to run based on provided Pfa
settingToRun = find(Pfas == Pfai,1,'first');

% Initialize ospa for each Monte Carlo run
gospaMC = zeros(numRuns,4);

% Use parfor for parallel simulations
parfor i = 1:numRuns
    rng(i, 'twister');

    % Restart scenario before each run
    restart(scene);

    % metrics vs time in this scenario
    gospaMetric = zeros(0,4);

    % Counter
    count = 0;

    % Advance scene and run
    while advance(scene)
        count = count + 1;

        % Use radar sensor using targetPoses function
```

```matlab
            own = scene.Platforms{end};
            tgtPoses = targetPoses(own,'rotmat');
            insPose = pose(own);
            poses = platformPoses(scene,'rotmat');
            truths = poses(1:end-1);
            time = scene.SimulationTime;

            % Reset tracker at the end of step so at the next call tracker is
            % reset to time 0.
            systemToReset = settingToRun*(abs(time - scene.StopTime) < 1e-3);

            % Store ospa metric. Tracks and detections can be outputted to run
            % scenario in code generation without needing Monte Carlo runs.
            [~,~,gospaMetric(count,:)] = clutterSimTracker_kernel(Pfas, tgtPoses, insPose, truths, ti
    end

    % Compute average of GOSPA in this run
    % Start from time step 20 to allow track establishment.
    gospaMC(i,:) = mean(gospaMetric(20:end,:));
end

end
```

helperGenerateCode

```matlab
function helperGenerateCode(scene,Pfas) %#ok<DEFNU>

% Generate sample pose using platformPoses
poses = platformPoses(scene,'rotmat');

% Generate sample inputs for the kernel function
%
% Pfas must be compile-time constant and they cannot change with time
Pfas = coder.Constant(Pfas);

% Target poses as a variable size arrray of max 20 elements
tgtPoses = coder.typeof(poses(1),[20 1],[1 0]);

% Scalar ins pose
insPose = pose(scene.Platforms{1},'true');

% Truth information with maximum 20 targets
truths = coder.typeof(poses(1),[20 1],[1 0]);

% Time
time = 0;

% Which false-alarm setting to run and which to reset. Reset is necessary
% after each run to reinitialize the tracker
systemToRun = 1;
systemToReset = 1;

inputs = {Pfas, tgtPoses, insPose, truths, time, systemToRun, systemToReset}; %#ok<NASGU>

% Same name as MATLAB file allows to shadow the MATLAB function when
% MEX file is available and execute code in MEX automatically.
codegen clutterSimTracker_kernel -args inputs -o clutterSimTracker_kernel;
```

getTruth

```
function truths = getTruth(scenario)
platPoses = platformPoses(scenario); % True Information
truths = platPoses(1:end-1); % Last object is ownship
end
```

clutterSimTracker_kernel

This function is defined in an external file named clutterSimTracker_kernel, available in the same working folder as this script.

```
function [detections, tracks, ospaMetric] = clutterSimTracker_kernel(Pfas, tgtPoses, insPose, tru

assert(numel(Pfas) == 4,'Only 4 false alarm settings supported. Rewrite more persistent variables

persistent tracker1 tracker2 tracker3 tracker4 ...
    radar1 radar2 radar3 radar4 ....
    reset1 reset2 reset3 reset4 ....
    gospa

if isempty(gospa) || isempty(reset1) || isempty(reset2) || isempty(reset3) || isempty(reset4)
    gospa = trackGOSPAMetric('CutoffDistance',50);
end

if isempty(tracker1) || isempty(radar1) || isempty(reset1)
    tracker1 = setupTracker(Pfas(1));
    radar1 = setupRadar(Pfas(1));
    reset1 = zeros(1,1);
end

if isempty(tracker2) || isempty(radar2) || isempty(reset2)
    tracker2 = setupTracker(Pfas(2));
    radar2 = setupRadar(Pfas(2));
    reset2 = zeros(1,1);
end

if isempty(tracker3) || isempty(radar3) || isempty(reset3)
    tracker3 = setupTracker(Pfas(3));
    radar3 = setupRadar(Pfas(3));
    reset3 = zeros(1,1);
end

if isempty(tracker4) || isempty(radar4) || isempty(reset4)
    tracker4 = setupTracker(Pfas(4));
    radar4 = setupRadar(Pfas(4));
    reset4 = zeros(1,1);
end

switch systemToRun
    case 1
        detections = radar1(tgtPoses, insPose, time);
        tracks = tracker1(detections, time);
    case 2
        detections = radar2(tgtPoses, insPose, time);
```

```
            tracks = tracker2(detections, time);
        case 3
            detections = radar3(tgtPoses, insPose, time);
            tracks = tracker3(detections, time);
        case 4
            detections = radar4(tgtPoses, insPose, time);
            tracks = tracker4(detections, time);
        otherwise
            error('Idx out of bounds');
end

[~,gp,~,loc,mT,fT] = gospa(tracks, truths);

ospaMetric = [gp loc mT fT];

switch systemToReset
    case 1
        reset1 = zeros(0,1);
    case 2
        reset2 = zeros(0,1);
    case 3
        reset3 = zeros(0,1);
    case 4
        reset4 = zeros(0,1);
end

end

function tracker = setupTracker(Pfa)
Pd = 0.9;
VCell = 0.0609; % Inlined value of cell volume;
Kc = Pfa/VCell; % Clutter density

% Birth rate
birthRate = 0.05*(5 + Pfa*48000)/0.1;

% Transform function and limits
sensorTransformFcn = @(x,params)x;
sensorTransformParameters = struct;
sensorLimits = bsxfun(@times,[-inf inf],ones(6,1));

% Assemble information into a trackingSensorConfiguration
config = trackingSensorConfiguration('SensorIndex',1,...
    'IsValidTime',true,...% Update every step
    'DetectionProbability',Pd,...% Detection Probability of detectable states
    'ClutterDensity',Kc,...% Clutter Density
    'SensorTransformFcn',sensorTransformFcn,...
    'SensorTransformParameters',sensorTransformParameters,...
    'SensorLimits',sensorLimits,...
    'MaxNumDetsPerObject',1,...
    'FilterInitializationFcn',@initcvgmphd);

tracker = trackerPHD('SensorConfigurations', config,...
    'BirthRate',birthRate);

end

function radar = setupRadar(Pfa)
```

```
Pd = 0.9;
radar = monostaticRadarSensor(1,...
    'UpdateRate',10,...
    'DetectionProbability',Pd,...
    'FalseAlarmRate',Pfa,...
    'FieldOfView',[120 1],...
    'ScanMode','No Scanning',...
    'DetectionCoordinates','Scenario',...% Report in scenario frame
    'HasINS',true,...% Enable INS to report in scenario frame
    'ReferenceRange',300,...% Short-range
    'ReferenceRCS',10,...
    'MaxUnambiguousRange',400,...% Short-range
    'RangeResolution',1,...
    'AzimuthResolution',1,...
    'HasFalseAlarms',true);
end
```

Introduction to Tracking Metrics

While designing a multi-object tracking system, it is essential to devise a method to evaluate its performance against the available ground truth. This ground truth is typically available from a simulation environment or by using techniques like ground-truth extraction using manual or automated labeling on recorded data. Though it is possible to qualitatively evaluate a tracking algorithm using visualization tools, the approach is usually not scalable. This example introduces different quantitative analysis tools in Sensor Fusion and Tracking Toolbox™ for assessing a tracker's performance. You will also use some common events like false tracks, track swaps etc. encountered while tracking multiple objects to understand the strengths and limitations of these tools.

Assignment and Error Metrics

The general workflow for evaluating a multi-object tracker in the presence of ground truth can be divided into 2 main steps. First, tracks are assigned to truths using an assignment algorithm. Second, using the computed assignment as an input, state-estimation accuracy for each track is computed. In each step, some key metrics for tracker's performance can be obtained. In Sensor Fusion and Tracking Toolbox™, the metrics corresponding to these two steps are termed as *assignment metrics* and *error metrics* respectively.
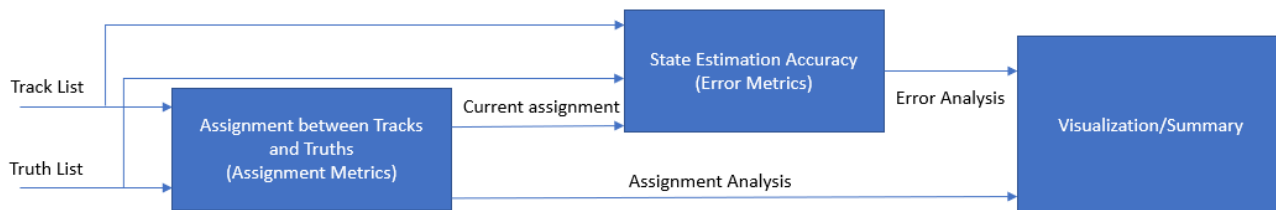
The assignment metrics assesses the characteristics of a tracker resulting due to incorrect assignment of sensor data to tracks due to ambiguity, missed targets or presence of false alarms. Incorrect assignment of sensor data can result in events like swapping of tracks between targets, creation of redundant tracks for a target, dropping of a true target track and creation of a false track. On the other hand, error metrics provide an assessment on state-estimation or object-level accuracy of the tracker. Even when a tracker does a good job at assignment, inaccuracies at track level can happen due to noise in the measurements as well as inaccuracies in modeling the target's motion model and sensor measurement model.

Tracks and Truths Definition

The assignment metrics use a gated greedy nearest-neighbor approach which allows many-to-one track assignments, which means that multiple tracks can be assigned to the same truth. At each step, the metrics aims to assign tracks to truths while accounting for their assignment in the previous time step. This helps the assignment metrics classify tracks into certain categories based on the current as well as previous assignments. The categories used by the assignment metrics for tracks are described in the images below.

Associated Track
T1 is initialized inside d_1 and assigned to truth P1

False Track
T2 is initialized outside d_1 and declared a false track

Associated Track
T1 remains assigned to P1

False Track
T2 remains false

Redundant Track
T3 is initialized and assigned to P1 but declared redundant because P1 is already associated to T1

Associated Track
T1 drifts outside d_1 but remains assigned to P1 because it is inside d_2

False Track
T2 remains false

Redundant Track
T3 remains redundant

Divergent Track
T1 drifts outside d_2 and is declared divergent. T1 becomes eligible for reassignment *

T2 does not survive in this step

Redundant Track
T3 remains redundant

Associated Tracks
T1 falls inside assignment threshold of truth P2 and becomes assigned to P2

T3 becomes assigned to P1

Assignment Threshold (d_1)

Divergence Threshold (d_2)

k_1 $k_1 + 1$ $k_1 + 2$ $k_1 + 3$ $k_1 + 4$ Time Steps

Multiple tracks can be assigned to the same truth (i.e. target) but only one track can be "associated" to that truth at any given time.

*A divergent track is still associated to the truth from it diverged till it gets reassigned (e.g. it can be stolen by another target).

The assignments metrics also represents truths as different categories of "targets" from the tracker's perspective. This helps the assignment metrics to record events such as track establishment and target break. The establishment and break events are shown in the image below.

Missed Target
P5 is a missed target because there are no tracks inside d_1 *and* no tracks were previously associated with P5.

Tracked Target
P5 becomes a tracked target because T7 is initialized inside d_1 (and associated to P5).

Tracked Target
P5 remains a tracked target because T7, although divergent, is still associated P5.

Tracked Target
P9 becomes a tracked target because T7 becomes associated to P9.

Dropped Target
P5 becomes a dropped target because it is no longer associated to any track.

Assignment Threshold (d_1)

Divergence Threshold (d_2)

k_2 $k_2 + 1$ $k_2 + 2$ $k_2 + 3$ Time Steps

Track Establishment   *Track Divergence*   *Target Break*

* A target break happens when a tracked target no longer has any associated track

By classifying tracks and truths into multiple categories, the assignment metrics output a cumulative analysis of the tracker's performance. The metrics also provide detailed information about assignment information for each truth and track. Furthermore, the assignment results at each step from the assignment metrics can also be used to evaluate the error metrics. The error metrics output a cumulative analysis of a tracker's accuracy in state-estimation. Similar to assignment metrics, error metrics also provide detailed information for each track and truth.

Next, you will learn how to use the assignment and error metrics in MATLAB.

Compute and Analyze Metrics

To use the assignment metrics, you create the `trackAssignmentMetric` System object™. You also specify properties of the object using name-value pairs to configure the metric for different applications. In this example, you specify the assignment and divergence distance as absolute error in position between track and truth. You also specify the distance thresholds for assignment and divergence of a track.

```
assignmentMetrics = trackAssignmentMetrics(...
    'AssignmentDistance','posabserr',...
    'DivergenceDistance','posabserr',...
    'DivergenceThreshold',1,...
    'AssignmentThreshold',0.85)

assignmentMetrics =
  trackAssignmentMetrics with properties:

    DistanceFunctionFormat: 'built-in'
                MotionModel: 'constvel'
        AssignmentThreshold: 0.8500
```

```
        DivergenceThreshold: 1
         AssignmentDistance: 'posabserr'
         DivergenceDistance: 'posabserr'
```

To use the error metrics, you create the `trackErrorMetrics` System object. Since no name-value pairs are provided, the default values are used. In this example, the targets are assumed to be tracked using a constant-velocity model.

```
errorMetrics = trackErrorMetrics

errorMetrics =
  trackErrorMetrics with properties:

    ErrorFunctionFormat: 'built-in'
            MotionModel: 'constvel'
```

The assignment and error metrics are driven by simulated tracks and truths by using a helper class, `helperMetricEvaluationScenarios`. An object of this class outputs a list of tracks and truths at each time step.

```
trackTruthSimulator = helperMetricEvaluationScenarios;
```

You also visualize the results of assignment and error metrics using a helper class named `helperMetricEvaluationDisplay`. These helper classes are included with the example in the same working folder as this script.

```
display = helperMetricEvaluationDisplay(...
    'PlotTrackClassification',true,...
    'PlotTruthClassification',true,...
    'PlotTrackErrors',false,...
    'PlotTruthErrors',false,...
    'PlotAssignments',true,...
    'RecordGIF',true,...
    'ErrorToPlot','posRMS');
```

You generate the list of tracks and truths at each time and run the assignment and error metrics using the following workflow.

```
% Time stamps at which the metrics are updated
timeStamps = 0:0.1:10;
n = numel(timeStamps);

% Initialization of recorded variables
posRMSE = zeros(n,1);
velRMSE = zeros(n,1);
posANEES = zeros(n,1);
velANEES = zeros(n,1);
truthError = cell(n,1);
trackError = cell(n,1);

% Loop over time stamps
for i = 1 : n
    % Current time
    time = timeStamps(i);

    % Generate tracks and truths using simulation
```

```
    [tracks, truths] = trackTruthSimulator(time);

    % You provide the tracks and truths as input to the assignment metrics.
    % This outputs a cumulative summary from all the tracks and the truths.
    [trackAssignmentSummary(i), truthAssignmentSummary(i)] = assignmentMetrics(tracks, truths); %

    % For detailed assignment information about each track, you use the
    % trackMetricsTable method.
    trackTable = trackMetricsTable(assignmentMetrics);

    % Similarly, for detailed assignment information about each truth, you
    % use the truthMetricsTable method.
    truthTable = truthMetricsTable(assignmentMetrics);

    % For running error metrics, you obtain the current assignment of
    % tracks to truths using the currentAssignment method.
    [assignedTrackIDs, assignedTruthIDs] = currentAssignment(assignmentMetrics);

    % The current assignment information as well as the list of tracks and
    % truths at the current time can then be passed as an input to the
    % error metrics.
    [posRMSE(i),velRMSE(i),posANEES(i),velANEES(i)] = errorMetrics(tracks, assignedTrackIDs, tru

    % For detailed error information about each record truth and track, you
    % use the cumulative metric methods on the error metrics.
    trackError{i} = cumulativeTrackMetrics(errorMetrics);
    truthError{i} = cumulativeTruthMetrics(errorMetrics);

    display(tracks, truths, assignedTrackIDs, assignedTruthIDs, trackTable, truthTable);
end
```

Analyze Assignment Metrics

The animation below shows a visual representation of the assignment metrics. In the beginning, both P1 and P2 did not have any associated tracks due to establishment delay. Therefore, they were both categorized as missed targets. Around X = 5 meters, P1 was not tracked for a few time steps. Notice that the assignment metrics outputs P2 as a dropped target during this time. When the tracks swapped between P1 and P2, they were initially declared divergent as they moved out of divergence thresholds. After a few time steps, the tracks reached assignment gate of new truths and became associated to the new truth.
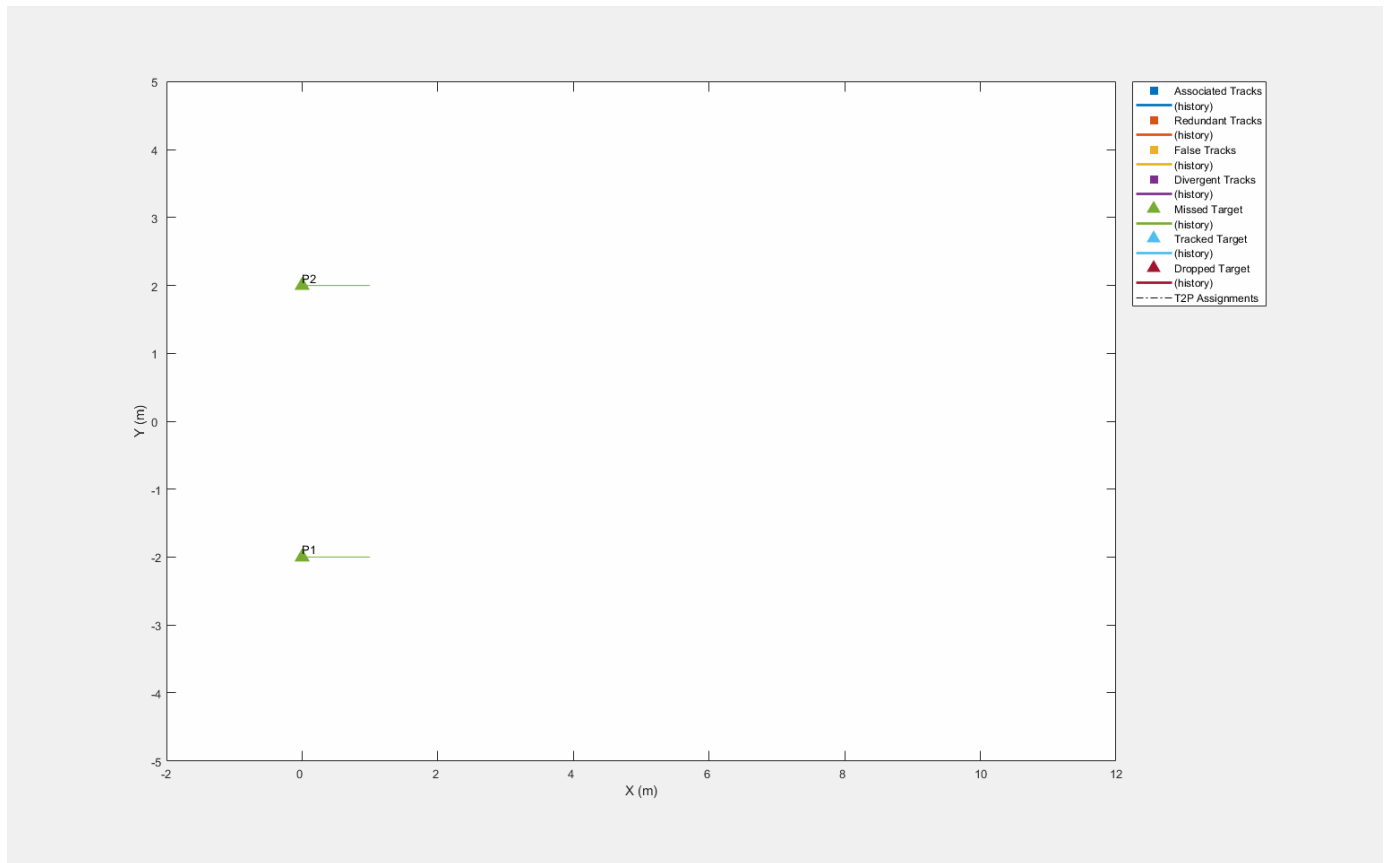
The visualization of the metric for each track and truth can help in easier understanding of different events. However, it can become overwhelming for large number of tracks and truths. An alternate approach is to use the summarized output of the assignment metrics. These outputs provide an overall summary of the assignment metrics up to the current time step. The fields and values of the assignment metrics summary at end of the simulation is shown below.

disp(trackAssignmentSummary(end));

```
        TotalNumTracks: 5
        NumFalseTracks: 1
          MaxSwapCount: 1
        TotalSwapCount: 2
    MaxDivergenceCount: 1
  TotalDivergenceCount: 3
   MaxDivergenceLength: 7
 TotalDivergenceLength: 17
    MaxRedundancyCount: 1
  TotalRedundancyCount: 1
   MaxRedundancyLength: 34
 TotalRedundancyLength: 34
```

disp(truthAssignmentSummary(end));

```
        TotalNumTruths: 2
      NumMissingTruths: 0
  MaxEstablishmentLength: 5
TotalEstablishmentLength: 10
```

```
    MaxBreakCount: 2
  TotalBreakCount: 4
   MaxBreakLength: 9
 TotalBreakLength: 9
```

By capturing the assignment summary at each step, you can also plot the cumulative analysis. The following plots show a subset of the fields for both track and truth assignment summary.

```
helperPlotStructArray(display,trackAssignmentSummary);
```
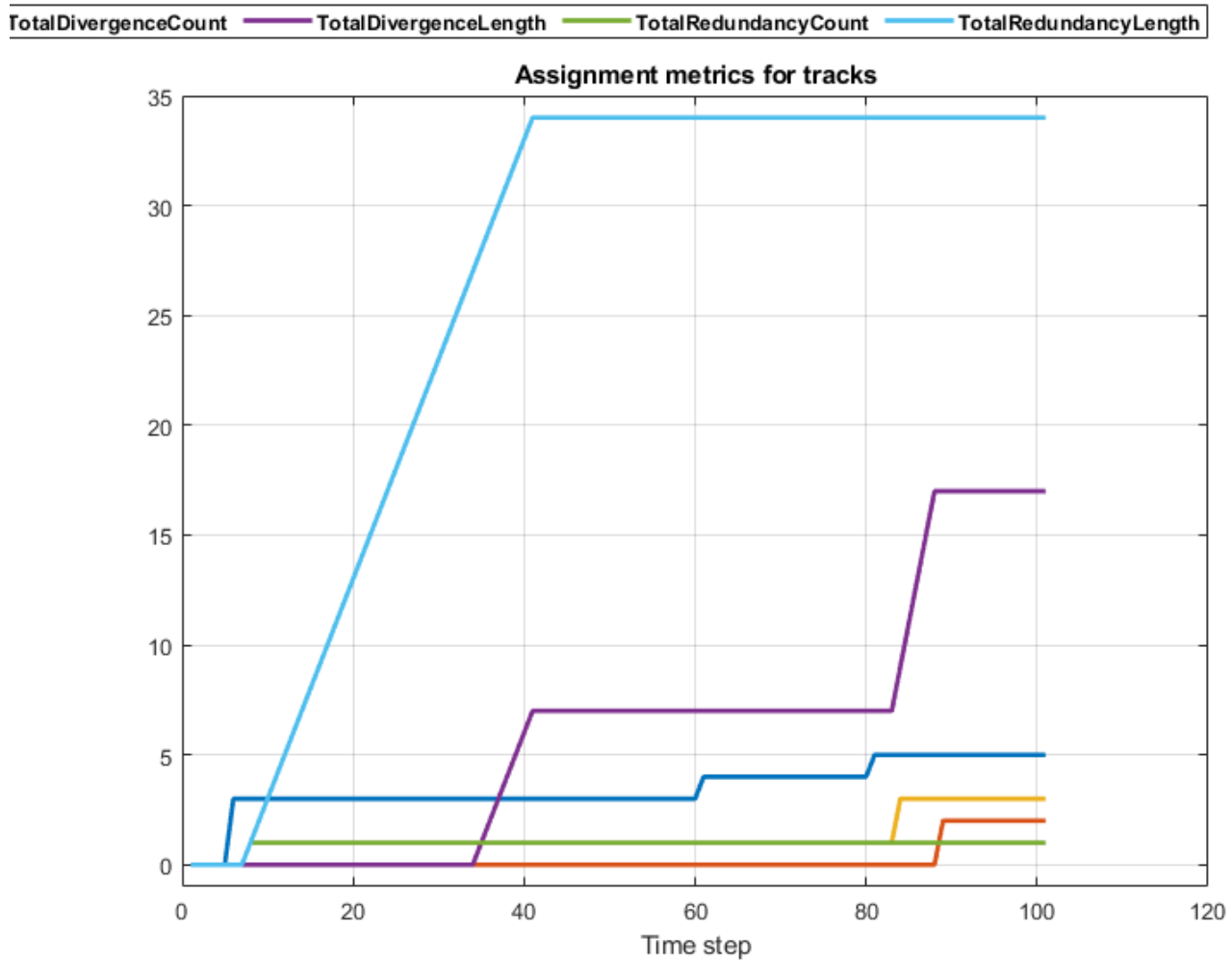


```
helperPlotStructArray(display,truthAssignmentSummary);
```
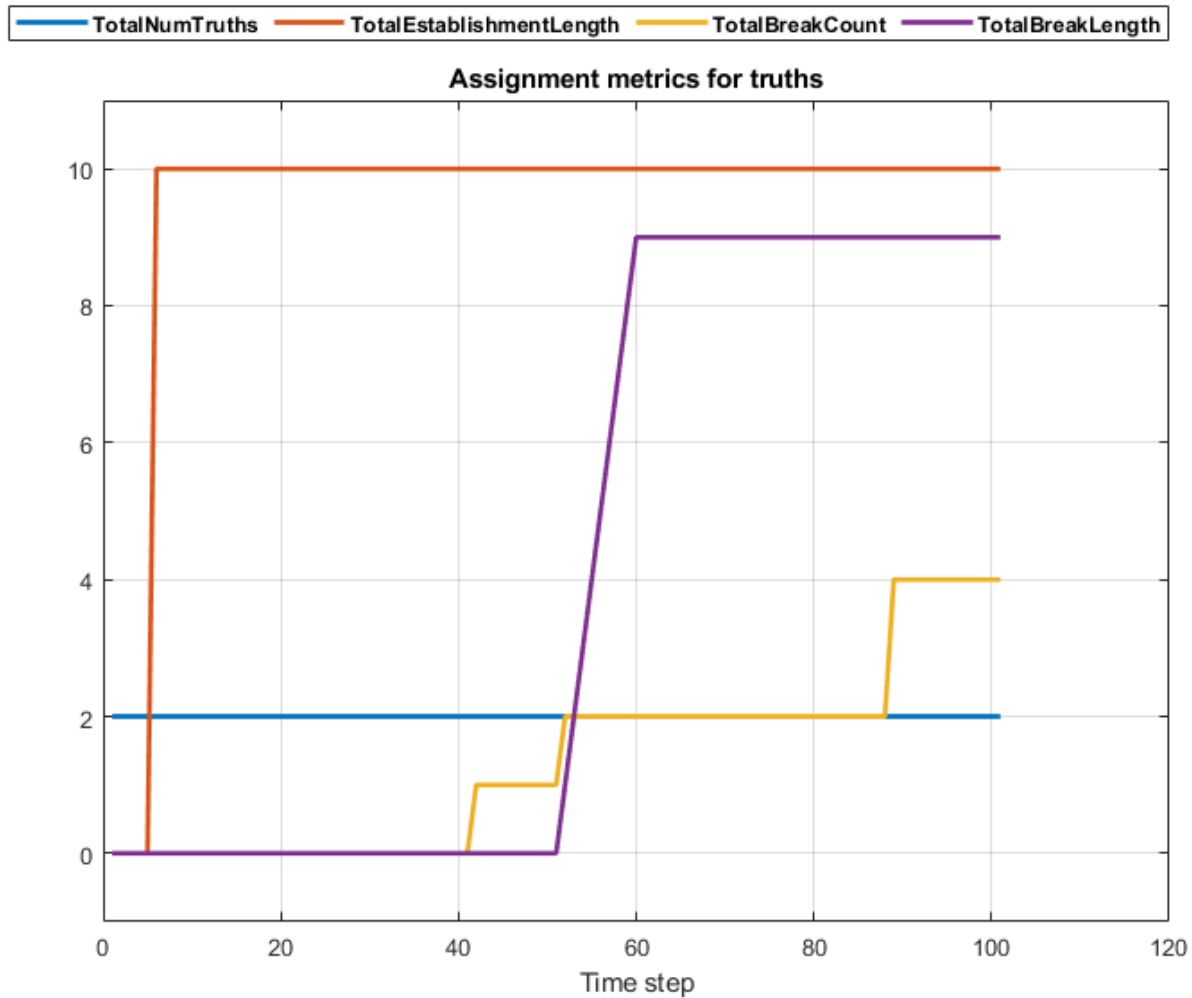
Analyze Error Metrics

Similar to assignment metrics, the error metrics can also be computed and visualized per track and truth. The fields of the estimation error for each track and truth at the end of simulation are shown below. Each of these fields represent the cumulative error over each track and truth.

Estimation error for each track

```
disp(trackError{end});
```

TrackID	posRMS	velRMS	posANEES	velANEES
1	0.72278	0.87884	0.52241	0.77236
2	0.16032	0.083823	0.025702	0.0070263
3	0.52898	1.3192	0.27982	1.7403
4	0.74411	2.112	0.55369	4.4604

Estimation error for each truth

```
disp(truthError{end});
```

TruthID	posRMS	velRMS	posANEES	velANEES
1	0.52259	1.3918	0.2731	1.9372
2	0.58988	1.259	0.34795	1.5851

By recording the errors at each time-step, you can also plot error vs time for each track and truth. The following plots show the RMS error in position for each track and truth. After a track is deleted, the error in its estimate remains constant. Notice that in the first-half of the simulation, the error in estimate of T3 is higher than that of T4 because the trajectory of T3 did not match the trajectory of P2 perfectly. Also notice the drop in error for estimate of T3 in the first-half as it slowly converged to the true position. During the track-swap event. the error in estimate of both T3 and T4 increased till their assignments switched.

```
% A utility to plot the table array. 'posRMS' is the field to plot
helperPlotTableArray(display,trackError,'posRMS');
```
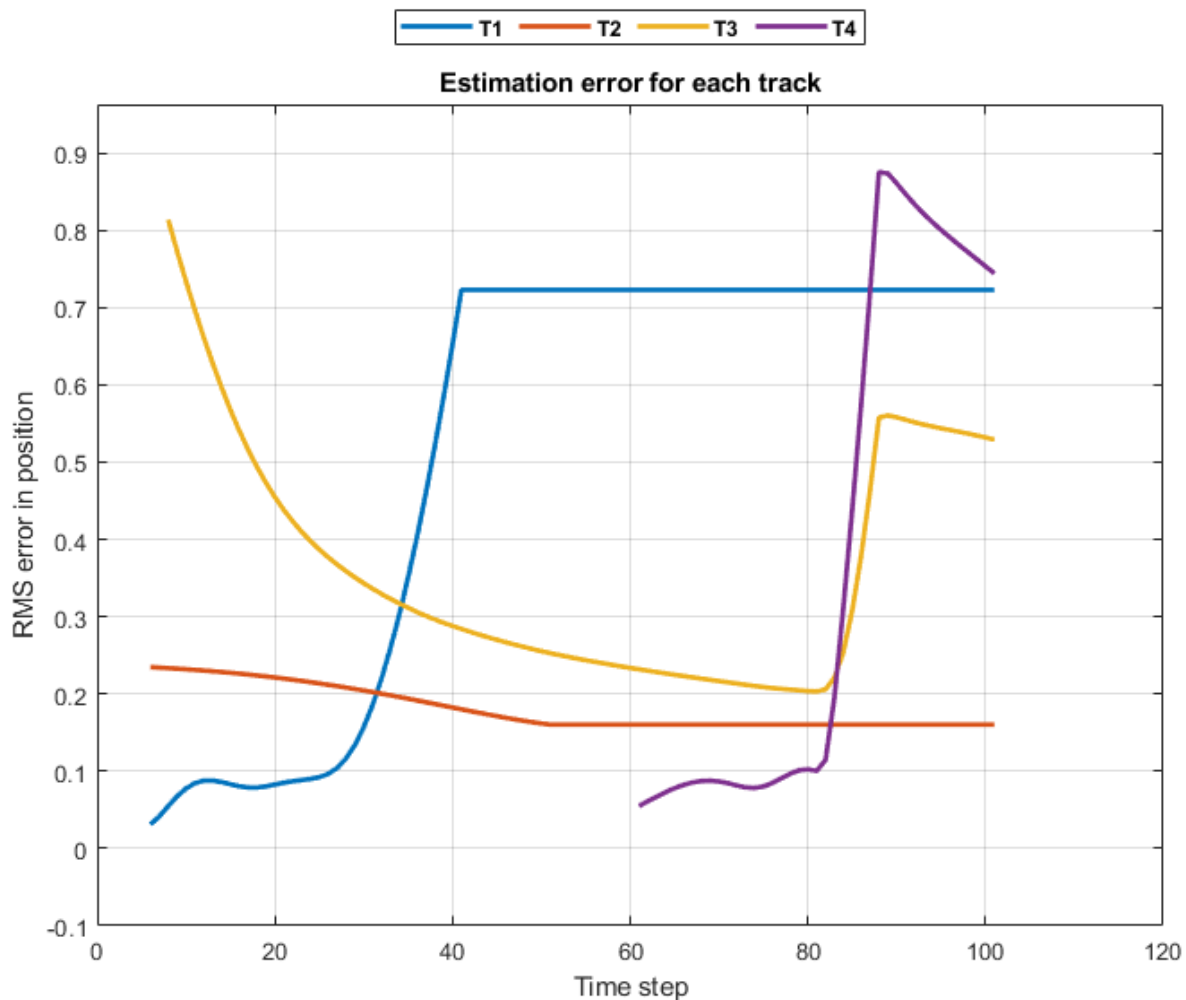
The error in estimate of each truth is simply the cumulation of estimation error of all tracks *assigned* to it. When P2 was assigned a redundant track around 8th time-step of the simulation, its estimation error jumped. As the redundant track approached P2, the error decreases, but again increases when its associated track diverged.

```
helperPlotTableArray(display,truthError,'posRMS');
```
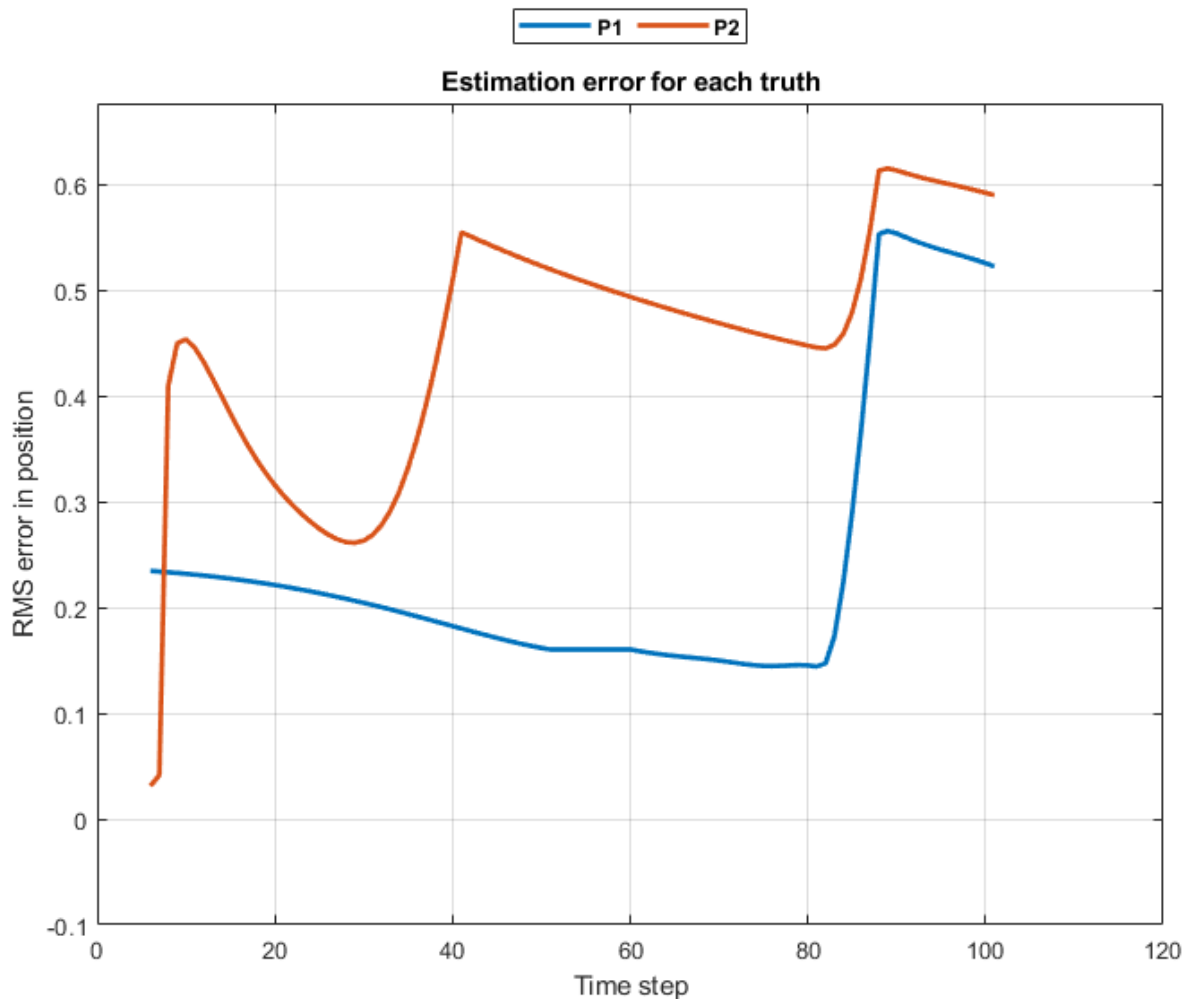

For a large number of tracks and truths, you can also use cumulative error metrics for *all* assigned tracks during their life-cycle. Notice the increase in error when the tracks swapped (near the 80th step). Also notice the smaller peak around the 25th time step when T1 diverted from its assigned truth P2 and was overtaken by the redundant track T3.

```
f = figure('Units','normalized','Position',[0.1 0.1 0.6 0.6]);
ax = axes(f);

subplot(2,2,1)
plot(posRMSE,'LineWidth',2);
xlabel('Time step');
```

```
ylabel('RMS Error in Position (m)');
grid('on');

subplot(2,2,2)
plot(velRMSE,'LineWidth',2);
xlabel('Time step');
ylabel('RMS Error in Velocity (m/s)');
grid('on');

subplot(2,2,3)
plot(posANEES,'LineWidth',2);
xlabel('Time step');
ylabel('Average Normalized Error in Position');
grid('on');

subplot(2,2,4)
plot(velANEES,'LineWidth',2);
xlabel('Time step');
ylabel('Average Normalized Error in Velocity');
grid('on');

sgtitle('Cumulative errors for all tracks','FontWeight','bold');
```

Cumulative errors for all tracks
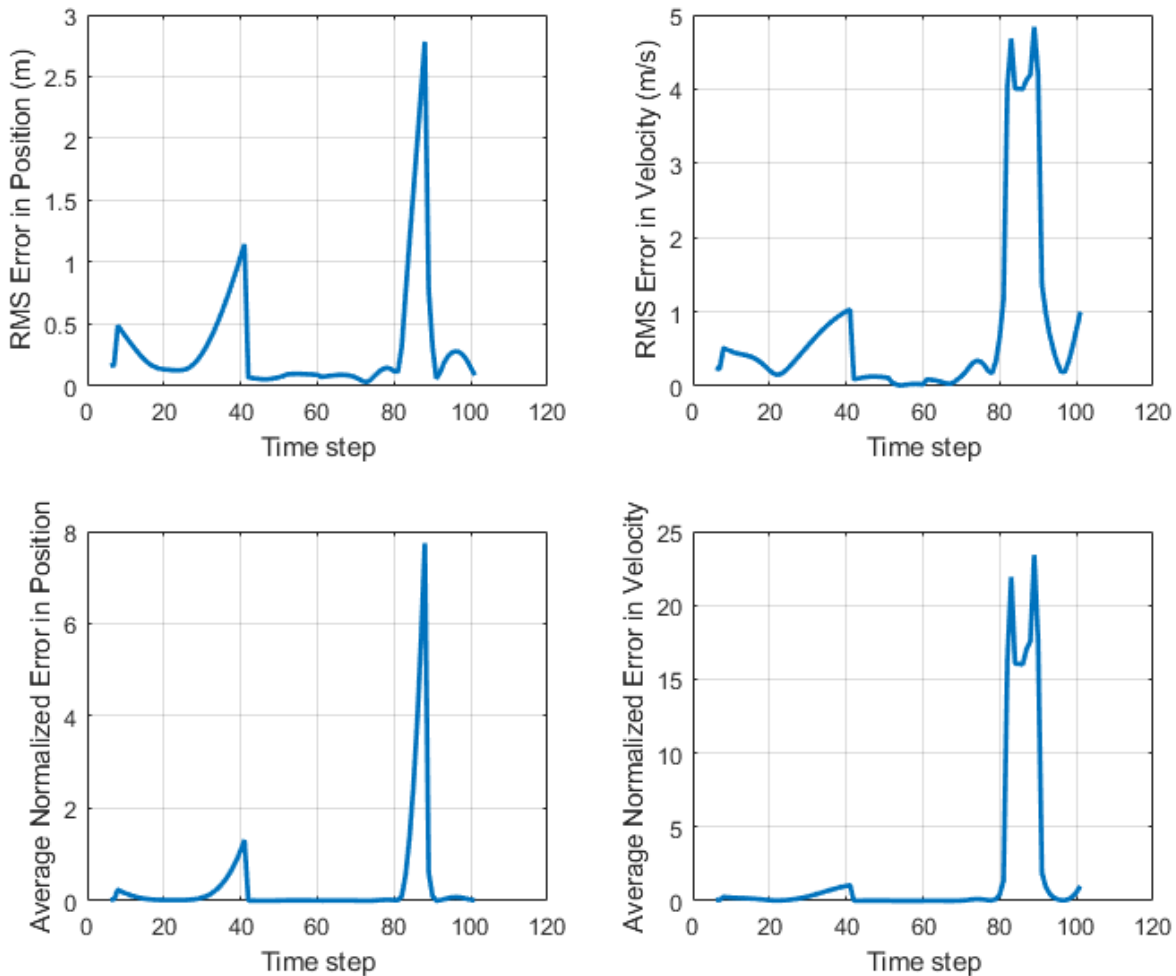
Summarizing Metrics as Scores

A common use-case for evaluating the performance of a multi-object tracker is to tune its parameters. This is typically done by combining the metrics into a single cost value that can serve as a function for an optimization algorithm. The assignment and error metrics provide several different measures of effectiveness (MoEs) and can be combined by first choosing the most relevant metrics and then performing a weighted combination depending on the application. The selection of the correct MoEs as well as their combination into a single score can be challenging. As an alternative to this approach, you can use the Optimal SubPattern Assignment (OSPA) [1] metric and Generalized Optimal SubPattern Assignment (GOSPA) [2] metric. Both OSPA and GOSPA metric assess the performance of a multi-object tracker by combining both assignment as well as state-estimation accuracy into a single cost value. Next, you will learn about OSPA and GOSPA metric and the workflow to compute these metrics in MATLAB.

OSPA Metric

The OSPA metric can be considered as a statistical distance between multiple tracks and truths. To compute the OSPA metric, the algorithm first assigns existing tracks and truths to each other using a Global Nearest Neighbor (GNN) algorithm. Once the assignment is computed, the metric divides the overall distance into two sub-components - localization and cardinality-mismatch. The localization component captures the errors resulting from state-estimation accuracy, while the cardinality-mismatch component captures the effect of redundant tracks, false tracks and missed truths. The traditional OSPA metric does not take into account the temporal history of tracks i.e. the assignments from previous step do not affect the metric at the current step. Therefore, effects like track-switches are not captured in the traditional OSPA metric. To circumvent this, a new sub-component was introduced for OSPA called the "labeling" component [3]. Combination of traditional OSPA with "labeling" component is sometimes referred to as "OSPA for Tracks" (OSPA-T) or Labelled-OSPA (LOSPA) [4].

To use the OSPA metric in MATLAB, you use the `trackOSPAMetric` System object. You can switch from OSPA to Labelled-OSPA by providing a non-zero valued `LabelingError` property. To understand how each sub-component is calculated, refer to the "Algorithms" section of `trackOSPAMetric`.

```
ospaMetric = trackOSPAMetric('Distance','posabserr',...
    'CutoffDistance',1,...
    'LabelingError',0.25);
```

Next you run the same scenario and compute the OSPA metric at each time step.

```
timeStamps = 0:0.1:10;
n = numel(timeStamps);

% Scene simulation
trackTruthSimulator = helperMetricEvaluationScenarios;

% Initialize variables
ospa = zeros(n,1);
locComponent = zeros(n,1);
cardComponent = zeros(n,1);
labelingComponent = zeros(n,1);

% Loop over time stamps
for i = 1:numel(timeStamps)
    time = timeStamps(i);

    % Track and truth
    [tracks, truths] = trackTruthSimulator(time);

    % Call the System object as a function and record OSPA and all 3 components
    [ospa(i),locComponent(i),cardComponent(i),labelingComponent(i)] = ospaMetric(tracks, truths);
end
```

Analyze OSPA Metric

```
figure('Units','normalized','Position',[0.1 0.1 0.6 0.6]);

subplot(2,1,1);
plot(ospa,'LineWidth',2);
xlabel('Time step');
ylabel('Cost');
```

```
title('OSPA');
grid('on');

subplot(2,1,2);
plot([locComponent cardComponent labelingComponent],'LineWidth',2);
xlabel('Time step');
ylabel('Component Cost');
title('OSPA Components');
legend('Localization','Cardinality-mismatch','Labeling');
grid('on');
```
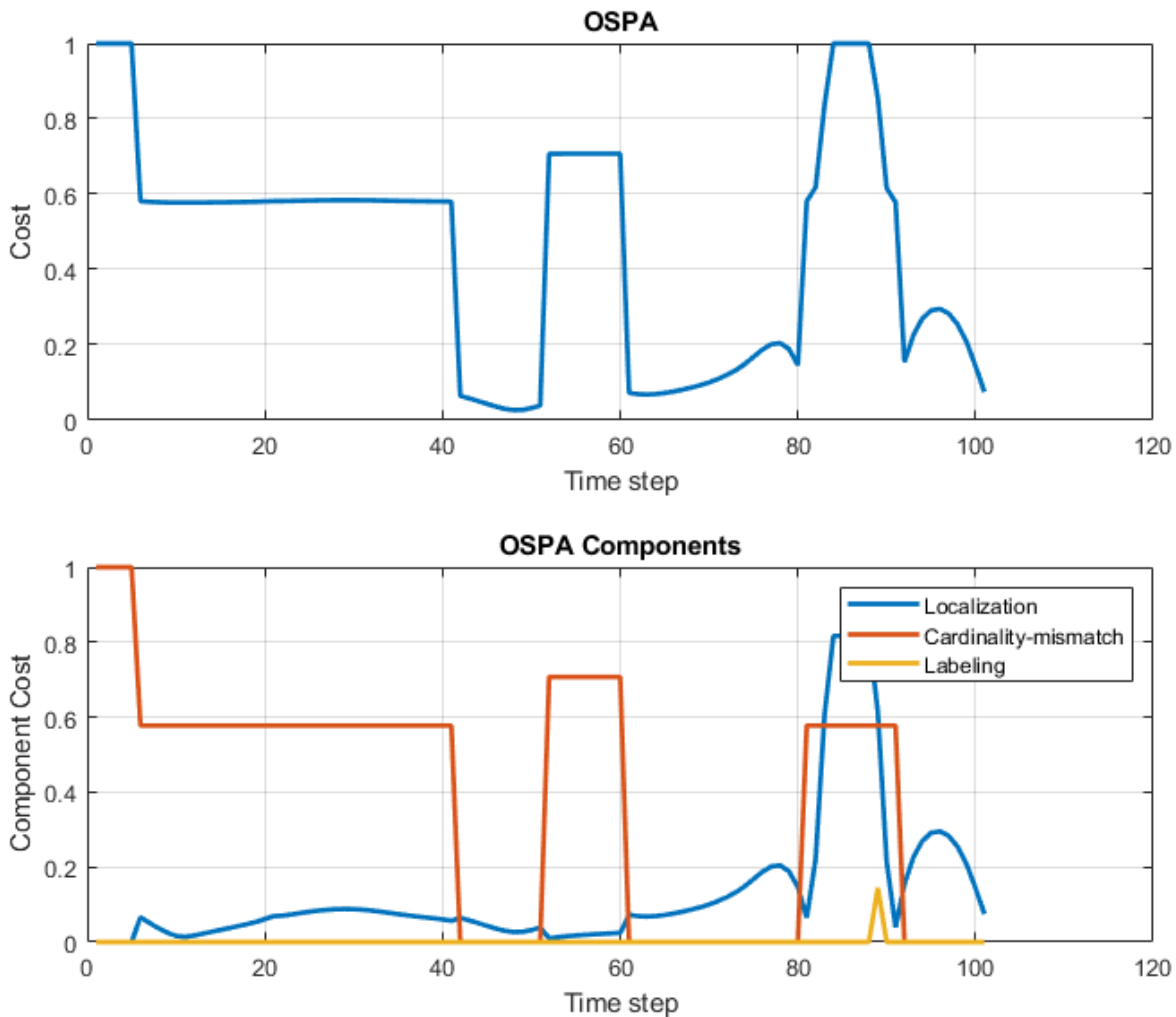

Notice the correlation between OSPA metric and different events in the scenario. The OSPA is high initially because of establishment delay. After establishment, the OSPA stays still relatively high due to the presence of a redundant track. After the redundant track was deleted, the OSPA dropped to a lower value. Around 50th time step the OSPA gained value as truth P1 was dropped. Notice that while the OSPA metric captures all these events correctly by providing a higher value, it does not provide finer details about each truth and track and the accuracy in their estimate. The only information available from OSPA is via its components. A higher localization component indicates that the

assigned tracks do not estimate the state of the truths correctly. This localization component is computed using the same type of distance as the assignment. A higher cardinality component indicates the presence of missed targets and false or redundant tracks. A higher labeling error indicates that the tracks are not labelled correctly, which indicates that the tracks are associated to their closest available truths.

GOSPA Metric

The approach used to compute GOSPA is similar to OSPA metric. Using a slightly different mathematical formulation, the GOSPA metric additionally computes subcomponents such as "missed targets component" and "false tracks component". Similar to traditional OSPA, the GOSPA also does not take into account temporal history of tracks. However, a metric similar to Labelled-OSPA can be achieved by adding a switching component [5]. The switching component captures the effect of switching assignments between truths. Each assignment change for the truth is penalized after being categorized as half-switch or full-switch. A half-switch refers to the event when a truth switches assignment from a track to being unassigned or vice-versa. A full-switch refers to the event when a truth switches assignment from one track to another.

To use the GOSPA metric in MATLAB, you create the `trackGOSPAMetric` System object. To account for track-switching, you provide a positive value for `SwitchingPenalty`.

```
gospaMetric = trackGOSPAMetric('Distance','posabserr',...
    'CutoffDistance',1,...
    'SwitchingPenalty',0.25);
```

Next you run the same scenario and compute the GOSPA metric at each time step.

```
timeStamps = 0:0.1:10;
n = numel(timeStamps);
trackTruthSimulator = helperMetricEvaluationScenarios;

% Initialize variables
labeledGospa = zeros(n,1);
traditionalGospa = zeros(n,1);
locComponent = zeros(n,1);
missedTargetComponent = zeros(n,1);
falseTrackComponent = zeros(n,1);
switchingComponent = zeros(n,1);

% Loop over time stamps
for i = 1:numel(timeStamps)
    time = timeStamps(i);

    % Track and truth
    [tracks, truths] = trackTruthSimulator(time);

    % Call the System object as a function and get all 4 components
    [labeledGospa(i),traditionalGospa(i),switchingComponent(i),locComponent(i),missedTargetCompon
end
```

Analyze Results

```
figure('Units','normalized','Position',[0.1 0.1 0.6 0.6]);

subplot(2,1,1);
plot(labeledGospa,'LineWidth',2);
xlabel('Time step');
```

```
ylabel('Cost');
title('GOSPA');
grid('on');

subplot(2,1,2);
plot([locComponent missedTargetComponent falseTrackComponent switchingComponent],'LineWidth',2);
xlabel('Time step');
ylabel('Component Cost');
title('GOSPA Components');
legend('Localization','Missed Target','False Track','Switching');
grid('on');
```

GOSPA

GOSPA Components
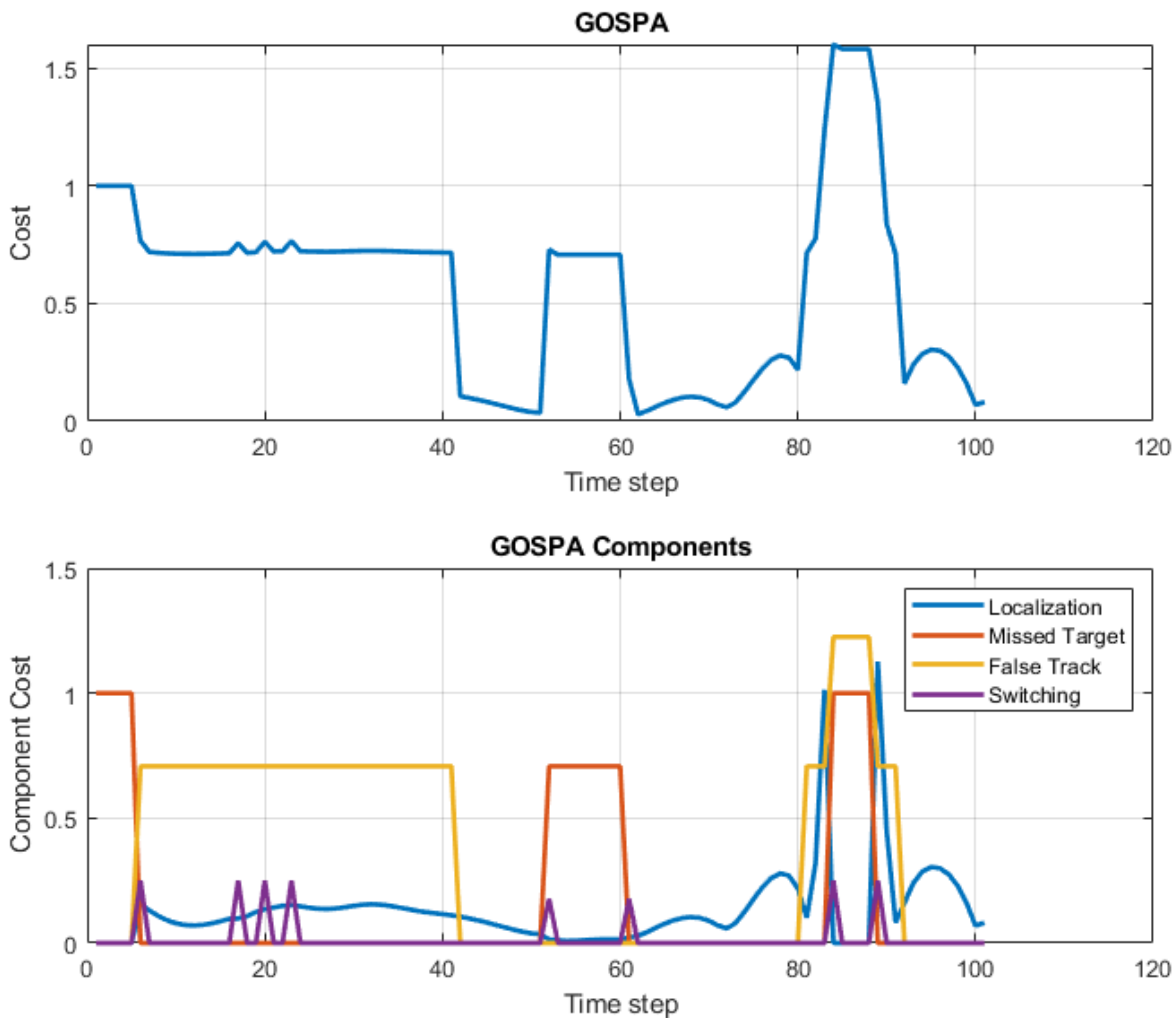
Notice that the GOSPA metric also captures the effect of different events during the scenario similar to the OSPA metric. In contrast to OSPA, it also provides information if the metric is higher due to false tracks or missed targets. Notice the peak is missed target component around 50th time step. This peak denotes the event when P1 was missed for a few steps. The peak around 80th time-step in missed target component denotes the event when the tracks swapped. The delay between divergence and reassignment resulted in missed targets as well as a false track components.

The peaks in the track switching component denote different events. The first peak accounts for truths switching from unassigned to assigned. The second peak accounts for the switching of tracks on P2. The third and fourth peak captures that truth P1 was unassigned and then assigned to another track respectively. The last two peaks account for truth unassignment and then reassignment.

Similar to OSPA, GOSPA also does not provide detailed information about each track and truth. The information from GOSPA is available via its components. As the name states, a higher missed target component denotes that targets are not being tracked and a higher false track component denotes the presence of false tracks. A higher switching penalty denotes events like establishment, track swaps and dropped tracks. The subdivision of components to include missed targets and false tracks assists in modifying the correct parameter of the tracker. For example, if false tracks are being created, a typical solution is to try and increase the threshold for track-confirmation.

Summary

In this example you learned about three different tools to analyze the performance of a multi-object tracking system. You also learned about the workflow in MATLAB to use these metrics. You also leaned about the type of information they provide and how it can be analyzed. The tools described in this example are flexible and can be customized for different applications and use-cases. There are "built-in" options available on all the tools to specify a different types of distances to be used for the metric. For example, absolute error in position or velocity or normalized error in position or velocity. You can also specify three different motion models for the tracks: constant velocity, constant turn-rate and constant acceleration. These "built-in" functionalities support tracks in form of `objectTrack` and truths generated by scenario simulations using `trackingScenario.` If the format of tracks and truths for your application is different, you can pre-process them to convert them. This allows you to use all the "built-in" functionality of the metrics. Alternatively, you can specify a custom distance function between a track and a truth. This allows you to control the distance as well as the format of each track and truth. To switch between custom and built-in functionality for OSPA and GOSPA metric, specify the `Distance` as 'custom'. To switch between custom and built-in functionality for assignment and error metrics, change the `DistanceFunctionFormat` and `ErrorFunctionFormat` to 'custom' for assignment and error metrics respectively.

References

[1] Schuhmacher, Dominic, Ba-Tuong Vo, and Ba-Ngu Vo. "A consistent metric for performance evaluation of multi-object filters." *IEEE transactions on signal processing* 56.8 (2008): 3447-3457.

[2] Rahmathullah, Abu Sajana, Ángel F. García-Fernández, and Lennart Svensson. "Generalized optimal sub-pattern assignment metric." *2017 20th International Conference on Information Fusion (Fusion)*. IEEE, 2017.

[3] Ristic, Branko, et al. "A metric for performance evaluation of multi-target tracking algorithms." *IEEE Transactions on Signal Processing* 59.7 (2011): 3452-3457.

[4] Mahler, Ronald PS. *Advances in statistical multisource-multitarget information fusion*. Artech House, 2014.

[5] Rahmathullah, Abu Sajana, Ángel F. García-Fernández, and Lennart Svensson. "A metric on the space of finite sets of trajectories for evaluation of multi-target tracking algorithms." *arXiv preprint arXiv:1605.01177* (2016).

Track-Level Fusion of Radar and Lidar Data

This example shows you how to generate an object-level track list from measurements of a radar and a lidar sensor and further fuse them using a track-level fusion scheme. You process the radar measurements using an extended object tracker and the lidar measurements using a joint probabilistic data association (JPDA) tracker. You further fuse these tracks using a track-level fusion scheme. The schematics of the workflow is shown below.
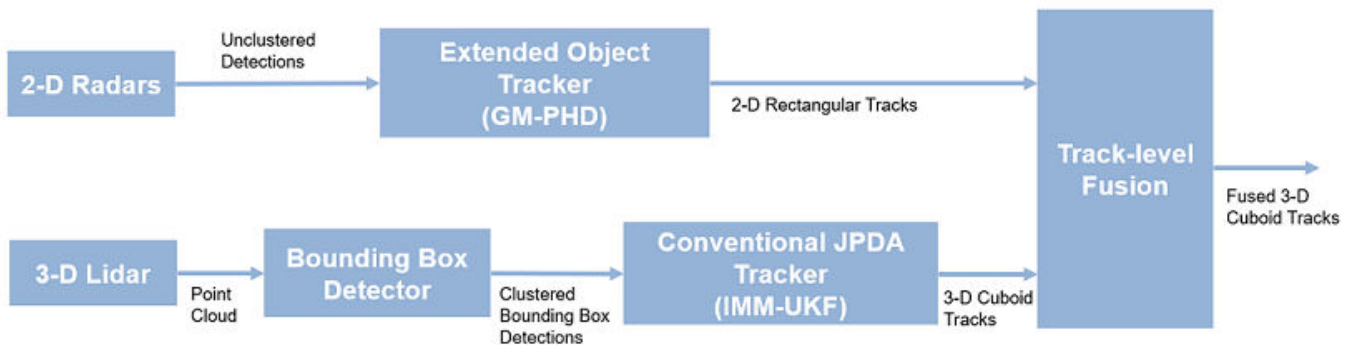
Setup Scenario for Synthetic Data Generation

The scenario used in this example is created using `drivingScenario` (Automated Driving Toolbox). The data from radar and lidar sensors is simulated using `radarDetectionGenerator` (Automated Driving Toolbox) and `lidarPointCloudGenerator` (Automated Driving Toolbox), respectively. The creation of the scenario and the sensor models is wrapped in the helper function `helperCreateRadarLidarScenario`. For more information on scenario and synthetic data generation, refer to "Create Driving Scenario Programmatically" (Automated Driving Toolbox).

```
% For reproducible results
rng(2019);
```

```
% Create scenario, ego vehicle and get radars and lidar sensor
[scenario, egoVehicle, radars, lidar] = helperCreateRadarLidarScenario;
```

The Ego vehicle is mounted with four 2-D radar sensors. The front and rear radar sensors have a field of view of 45 degrees. The left and right radar sensors have a field of view of 150 degrees. Each radar has a resolution of 6 degrees in azimuth and 2.5 meters in range. The Ego is also mounted with one 3-D lidar sensor with a field of view of 360 degrees in azimuth and 40 degrees in elevation. The lidar has a resolution of 0.2 degrees in azimuth and 1.25 degrees in elevation (32 elevation channels). Visualize the configuration of the sensors and the simulated sensor data in the animation below. Notice that the radars have higher resolution than objects and therefore return multiple measurements per object. Also notice that the lidar interacts with the low-poly mesh of the actor as well as the road surface to return multiple points from these objects.
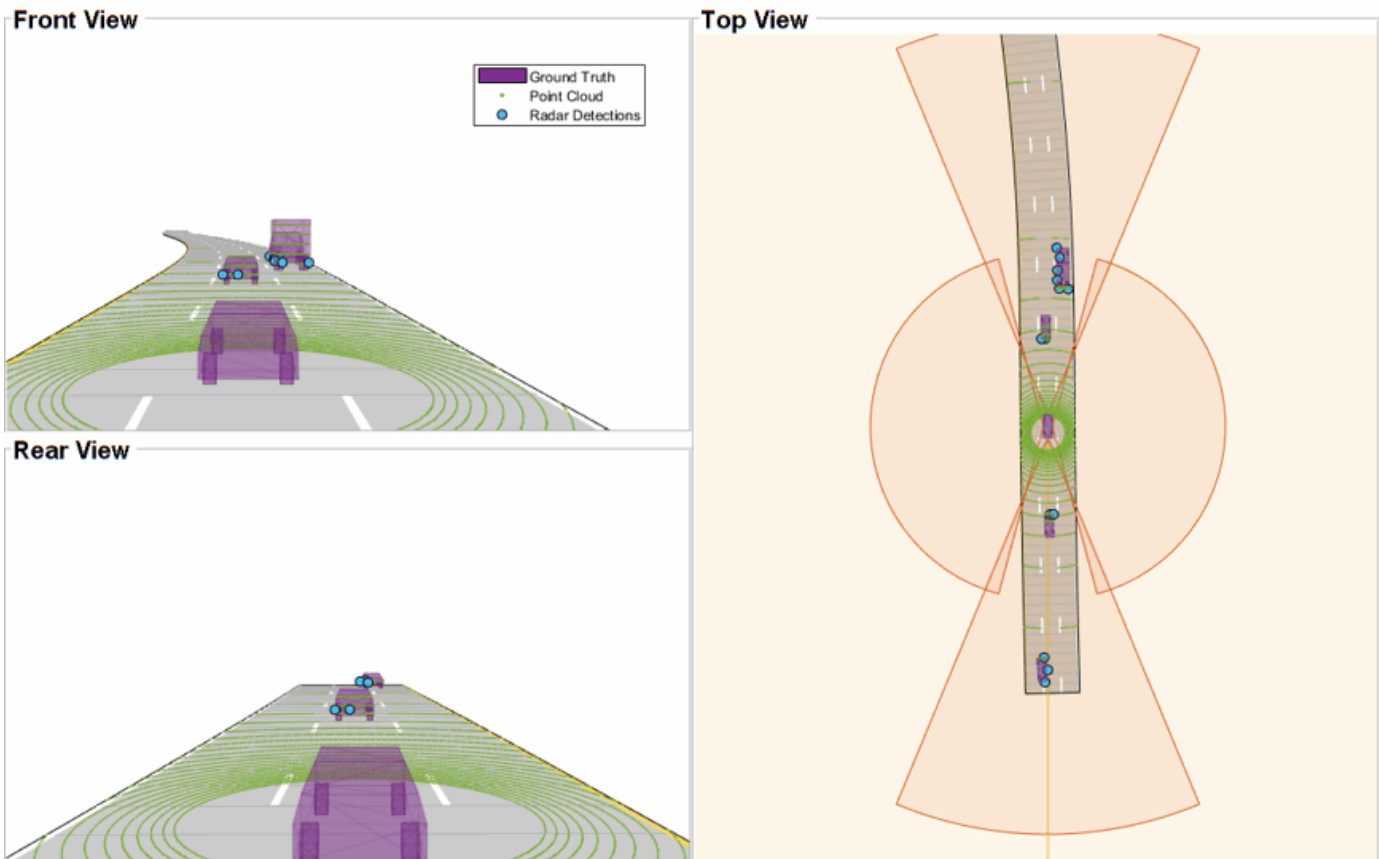
Radar Tracking Algorithm

As mentioned, the radars have higher resolution than the objects and return multiple detections per object. Conventional trackers such as Global Nearest Neighbor (GNN) and Joint Probabilistic Data Association (JPDA) assume that the sensors return at most one detection per object per scan. Therefore, the detections from high-resolution sensors must be either clustered before processing it with conventional trackers or must be processed using extended object trackers. Extended object trackers do not require pre-clustering of detections and usually estimate both kinematic states (for example, position and velocity) and the extent of the objects. For a more detailed comparison between conventional trackers and extended object trackers, refer to the "Extended Object Tracking of Highway Vehicles with Radar and Camera" on page 5-147 example.

In general, extended object trackers offer better estimation of objects as they handle clustering and data association simultaneously using temporal history of tracks. In this example, the radar detections are processed using a Gaussian mixture probability hypothesis density (GM-PHD) tracker (`trackerPHD` and `gmphd`) with a rectangular target model. For more details on configuring the tracker, refer to the "GM-PHD Rectangular Object Tracker" section of the "Extended Object Tracking of Highway Vehicles with Radar and Camera" on page 5-147 example.

The algorithm for tracking objects using radar measurements is wrapped inside the helper class, `helperRadarTrackingAlgorithm`, implemented as a System object™. This class outputs an array of `objectTrack` objects and define their state according to the following convention:

$$[x \; y \; s \; \theta \; \omega \; L \; W]$$
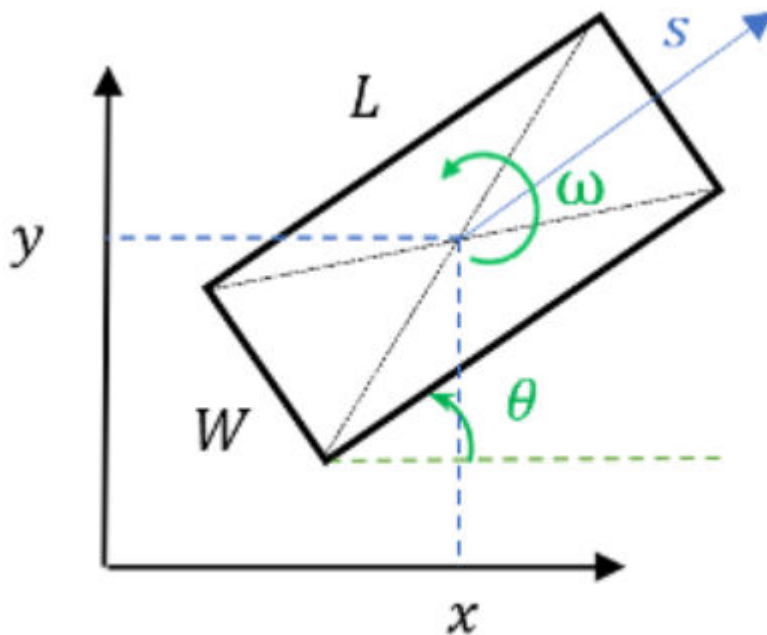
```
radarTrackingAlgorithm = helperRadarTrackingAlgorithm(radars);
```

Lidar Tracking Algorithm

Similar to radars, the lidar sensor also return multiple measurements per object. Further, the sensor returns a large number of points from the road, which must be removed to before used as inputs for an object-tracking algorithm. While lidar data from obstacles can be directly processed via extended object tracking algorithm, conventional tracking algorithms are still more prevalent for tracking using lidar data. The first reason for this trend is mainly observed due to higher computational complexity of extended object trackers for large data sets. The second reason is the investments into advanced Deep learning-based detectors such as PointPillars [1], VoxelNet [2] and PIXOR [3], which can segment a point cloud and return bounding box detections for the vehicles. These detectors can help in overcoming the performance degradation of conventional trackers due to improper clustering.

In this example, the lidar data is processed using a conventional joint probabilistic data association (JPDA) tracker, configured with an interacting multiple model (IMM) filter. The pre-processing of lidar data to remove point cloud is performed by using a RANSAC-based plane-fitting algorithm and bounding boxes are formed by performing a Euclidian-based distance clustering algorithm. For more information about the algorithm, refer to the "Track Vehicles Using Lidar: From Point Cloud to Track List" on page 5-369 example. Compared the linked example, the tracking is performed in the scenario frame and the tracker is tuned differently to track objects of different sizes. Further the states of the variables are defined differently to constraint the motion of the tracks in the direction of its estimated heading angle.

The algorithm for tracking objects using lidar data is wrapped inside the helper class, `helperLidarTrackingAlgorithm` implemented as System object. This class outputs an array of `objectTrack` objects and defines their state according to the following convention:

$$[x \; y \; s \; \theta \; \omega \; z \; \dot{z} \; L \; W \; H]$$

The states common to the radar algorithm are defined similarly. Also, as a 3-D sensor, the lidar tracker outputs three additional states, z, $\dot{z}$ and H, which refer to z-coordinate (m), z-velocity (m/s), and height (m) of the tracked object respectively.

```
lidarTrackingAlgorithm = helperLidarTrackingAlgorithm(lidar);
```

Set Up Fuser, Metrics, and Visualization

Fuser

Next, you will set up a fusion algorithm for fusing the list of tracks from radar and lidar trackers. Similar to other tracking algorithms, the first step towards setting up a track-level fusion algorithm is defining the choice of state vector (or state-space) for the fused or central tracks. In this case, the state-space for fused tracks is chosen to be same as the lidar. After choosing a central track state-space, you define the transformation of the central track state to the local track state. In this case, the local track state-space refers to states of radar and lidar tracks. To do this, you use a `fuserSourceConfiguration` object.

Define the configuration of the radar source. The `helperRadarTrackingAlgorithm` outputs tracks with `SourceIndex` set to 1. The `SourceIndex` is provided as a property on each tracker to uniquely identify it and allows a fusion algorithm to distinguish tracks from different sources. Therefore, you set the `SourceIndex` property of the radar configuration as same as those of the radar tracks. You set `IsInitializingCentralTracks` to `true` to let that unassigned radar tracks initiate new central tracks. Next, you define the transformation of a track in central state-space to the radar state-space and vice-versa. The helper functions `central2radar` and `radar2central` perform the two transformations and are included at the end of this example.

```
radarConfig = fuserSourceConfiguration('SourceIndex',1,...
    'IsInitializingCentralTracks',true,...
    'CentralToLocalTransformFcn',@central2radar,...
    'LocalToCentralTransformFcn',@radar2central);
```

Define the configuration of the lidar source. Since the state-space of a lidar track is same as central track, you do not define any transformations.

```
lidarConfig = fuserSourceConfiguration('SourceIndex',2,...
    'IsInitializingCentralTracks',true);
```

The next step is to define the state-fusion algorithm. The state-fusion algorithm takes multiple states and state covariances in the central state-space as input and return a fused estimate of the state and the covariances. In this example, you use a covariance intersection algorithm provided by the helper function, `helperRadarLidarFusionFcn`. A generic covariance intersection algorithm for two Gaussian estimates with mean x_i and covariance P_i can be defined according to the following equations:

$$P_F^{-1} = w_1 P_1^{-1} + w_2 P_2^{-1}$$

$$x_F = P_F(w_1 P_1^{-1} x_1 + w_2 P_2^{-1} x_2)$$

where x_F and P_F are the fused state and covariance and w_1 and w_2 are mixing coefficients from each estimate. Typically, these mixing coefficients are estimated by minimizing the determinant or the trace of the fused covariance. In this example, the mixing weights are estimated by minimizing the determinant of positional covariance of each estimate. Furthermore, as the radar does not estimate 3-D states, 3-D states are only fused with lidars. For more details, refer to the `helperRadarLidarFusionFcn` function shown at the end of this script.

Next, you assemble all the information using a `trackFuser` object.

```
% The state-space of central tracks is same as the tracks from the lidar,
% therefore you use the same state transition function. The function is
% defined inside the helperLidarTrackingAlgorithm class.
f = lidarTrackingAlgorithm.StateTransitionFcn;

% Create a trackFuser object
fuser = trackFuser('SourceConfigurations',{radarConfig;lidarConfig},...
    'StateTransitionFcn',f,...
    'ProcessNoise',diag([1 3 1]),...
    'HasAdditiveProcessNoise',false,...
    'AssignmentThreshold',[250 inf],...
    'ConfirmationThreshold',[3 5],...
    'DeletionThreshold',[5 5],...
    'StateFusion','Custom',...
    'CustomStateFusionFcn',@helperRadarLidarFusionFcn);
```

Metrics

In this example, you assess the performance of each algorithm using the Generalized Optimal SubPattern Assignment Metric (GOSPA) metric. You setup three separate metrics using `trackGOSPAMetric` for each of the trackers. GOSPA metric aims to evaluate the performance of a tracking system by providing a scalar cost. A lower value of the metric indicates better performance of the tracking algorithm.

To use the GOSPA metric with custom motion models like the one used in this example, you set the `Distance` property to 'custom' and define a distance function between a track and its associated ground truth. These distance functions, shown at the end of this example are `helperRadarDistance`, and `helperLidarDistance`.

```
% Radar GOSPA
gospaRadar = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperRadarDistance,...
    'CutoffDistance',25);

% Lidar GOSPA
gospaLidar = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperLidarDistance,...
    'CutoffDistance',25);

% Central/Fused GOSPA
gospaCentral = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperLidarDistance,...% State-space is same as lidar
    'CutoffDistance',25);
```

Visualization

The visualization for this example is implemented using a helper class `helperLidarRadarTrackFusionDisplay`. The display is divided into 4 panels. The display plots the measurements and tracks from each sensor as well as the fused track estimates. The legend for the display is shown below. Furthermore, the tracks are annotated by their unique identity (`TrackID`) as well as a prefix. The prefixes "R", "L" and "F" stand for radar, lidar, and fused estimate, respectively.

```
% Create a display.
% FollowActorID controls the actor shown in the close-up
```

```
% display
display = helperLidarRadarTrackFusionDisplay('FollowActorID',3);

% Show persistent legend
showLegend(display,scenario);
```

Ground Truth

Radar Tracks

Lidar Tracks

Fused Tracks

· Point Cloud

Radar Detections

Lidar Bounding Box Detections
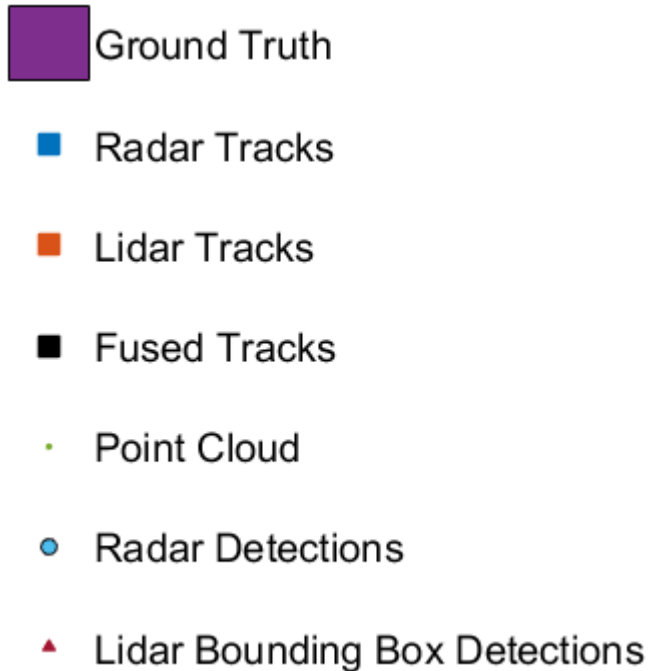
Run Scenario and Trackers

Next, you advance the scenario, generate synthetic data from all sensors and process it to generate tracks from each of the system. You also compute the metric for each tracker using the ground truth available from the scenario.

```
% Initialzie GOSPA metric and its components for all tracking algorithms.
gospa = zeros(3,0);
missTarget = zeros(3,0);
falseTracks = zeros(3,0);

% Initialize fusedTracks
fusedTracks = objectTrack.empty(0,1);

% A counter for time steps elapsed for storing gospa metrics.
idx = 1;

% Ground truth for metrics. This variable updates every time-step
% automatically being a handle to the actors.
groundTruth = scenario.Actors(2:end);

while advance(scenario)
    % Current time
    time = scenario.SimulationTime;

    % Collect radar and lidar measurements and ego pose to track in
```

```
    % scenario frame. See helperCollectSensorData below.
    [radarDetections, ptCloud, egoPose] = helperCollectSensorData(egoVehicle, radars, lidar, time

    % Generate radar tracks
    radarTracks = radarTrackingAlgorithm(egoPose, radarDetections, time);

    % Generate lidar tracks and analysis information like bounding box
    % detections and point cloud segmentation information
    [lidarTracks, lidarDetections, segmentationInfo] = ...
        lidarTrackingAlgorithm(egoPose, ptCloud, time);

    % Concatenate radar and lidar tracks
    localTracks = [radarTracks;lidarTracks];

    % Update the fuser. First call must contain one local track
    if ~(isempty(localTracks) && ~isLocked(fuser))
        fusedTracks = fuser(localTracks,time);
    end

    % Capture GOSPA and its components for all trackers
    [gospa(1,idx),~,~,~,missTarget(1,idx),falseTracks(1,idx)] = gospaRadar(radarTracks, groundTru
    [gospa(2,idx),~,~,~,missTarget(2,idx),falseTracks(2,idx)] = gospaLidar(lidarTracks, groundTru
    [gospa(3,idx),~,~,~,missTarget(3,idx),falseTracks(3,idx)] = gospaCentral(fusedTracks, groundT

    % Update the display
    display(scenario, radars, radarDetections, radarTracks, ...
        lidar, ptCloud, lidarDetections, segmentationInfo, lidarTracks,...
        fusedTracks);

    % Update the index for storing GOSPA metrics
    idx = idx + 1;
end

% Update example animations
updateExampleAnimations(display);
```

Evaluate Performance

Evaluate the performance of each tracker using visualization as well as quantitative metrics. Analyze different events in the scenario and understand how the track-level fusion scheme helps achieve a better estimation of the vehicle state.

Track Maintenance

The animation below shows the entire run every three time-steps. Note that each of the three tracking systems - radar, lidar, and the track-level fusion - were able to track all four vehicles in the scenario and no false tracks were confirmed.
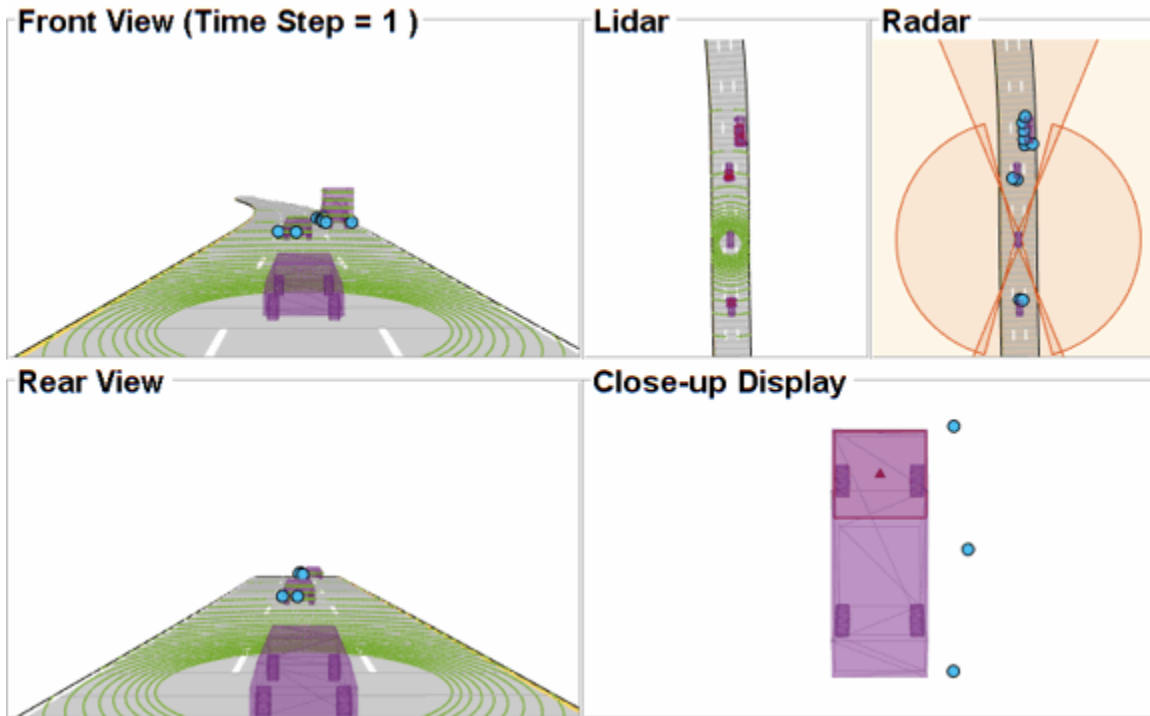
You can also quantitatively measure this aspect of the performance using "missed target" and "false track" components of the GOSPA metric. Notice in the figures below that missed target component starts from a higher value due to establishment delay and goes down to zero in about 5-10 steps for each tracking system. Also, notice that the false track component is zero for all systems, which indicates that no false tracks were confirmed.

```
% Plot missed target component
figure; plot(missTarget','LineWidth',2); legend('Radar','Lidar','Fused');
title("Missed Target Metric"); xlabel('Time step'); ylabel('Metric'); grid on;

% Plot false track component
figure; plot(falseTracks','LineWidth',2); legend('Radar','Lidar','Fused');
title("False Track Metric"); xlabel('Time step'); ylabel('Metric'); grid on;
```
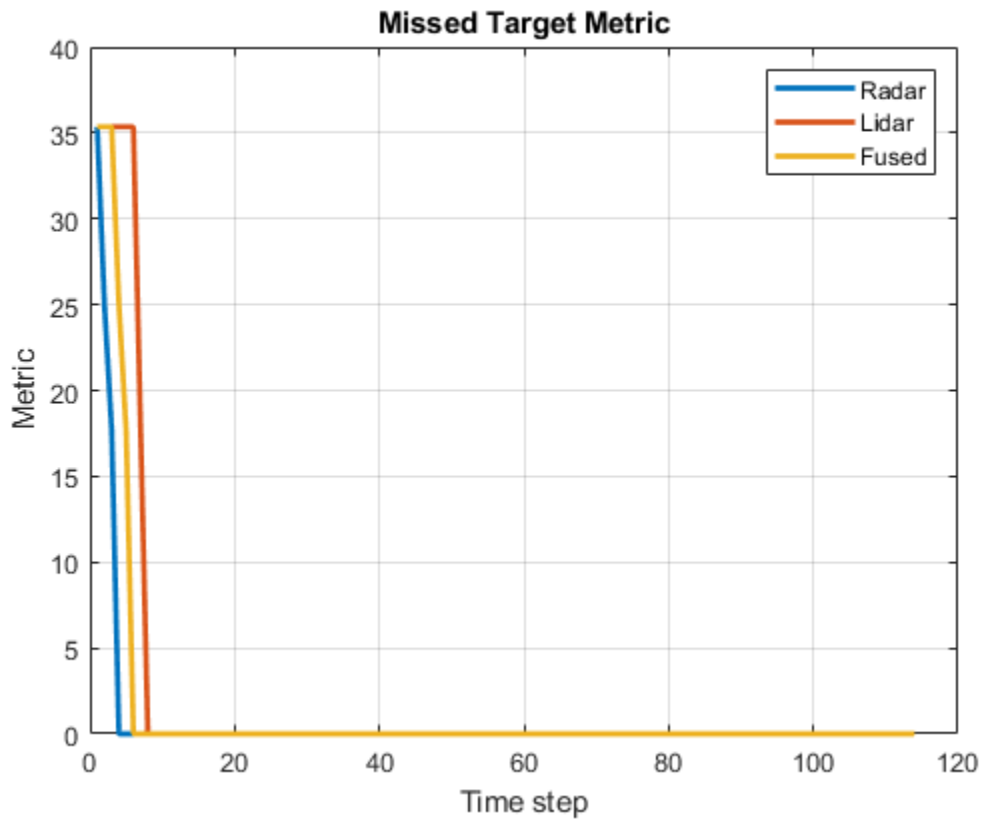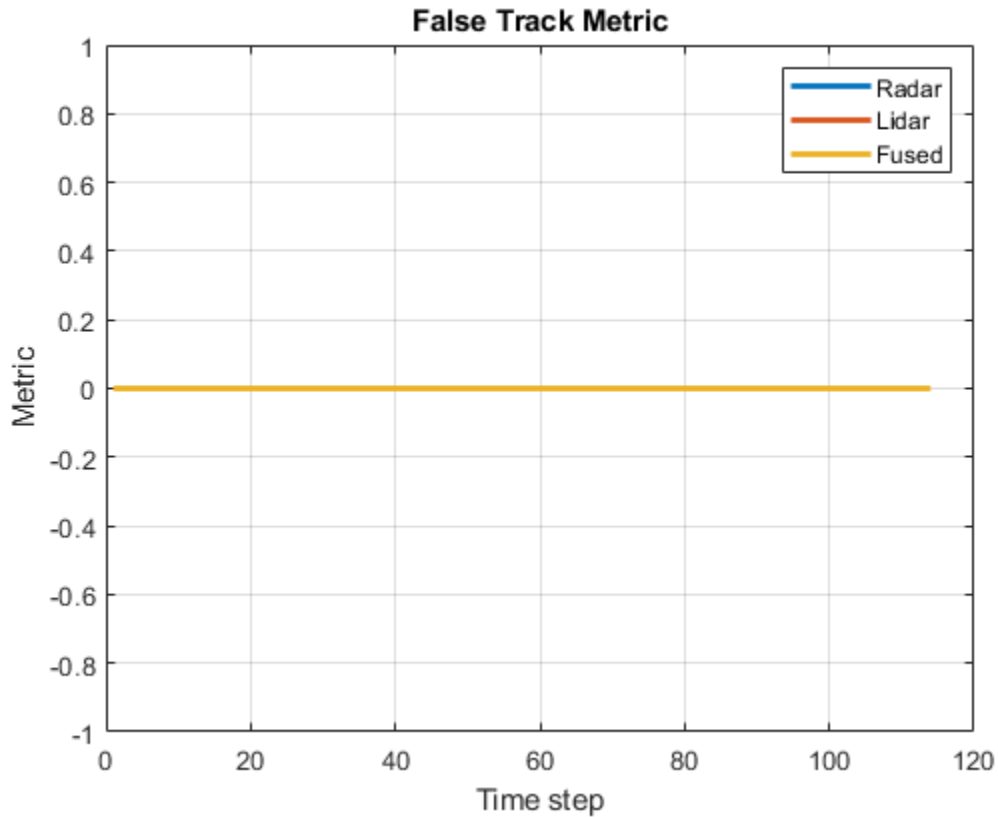
Track-level Accuracy

The track-level or localization accuracy of each tracker can also be quantitatively assessed by the GOSPA metric at each time step. A lower value indicates better tracking accuracy. As there were no missed targets or false tracks, the metric captures the localization errors resulting from state estimation of each vehicle.

Note that the GOSPA metric for fused estimates is lower than the metric for individual sensor, which indicates that track accuracy increased after fusion of track estimates from each sensor.

```
% Plot GOSPA
figure; plot(gospa','LineWidth',2); legend('Radar','Lidar','Fused');
title("GOSPA Metric"); xlabel('Time step'); ylabel('Metric'); grid on;
```
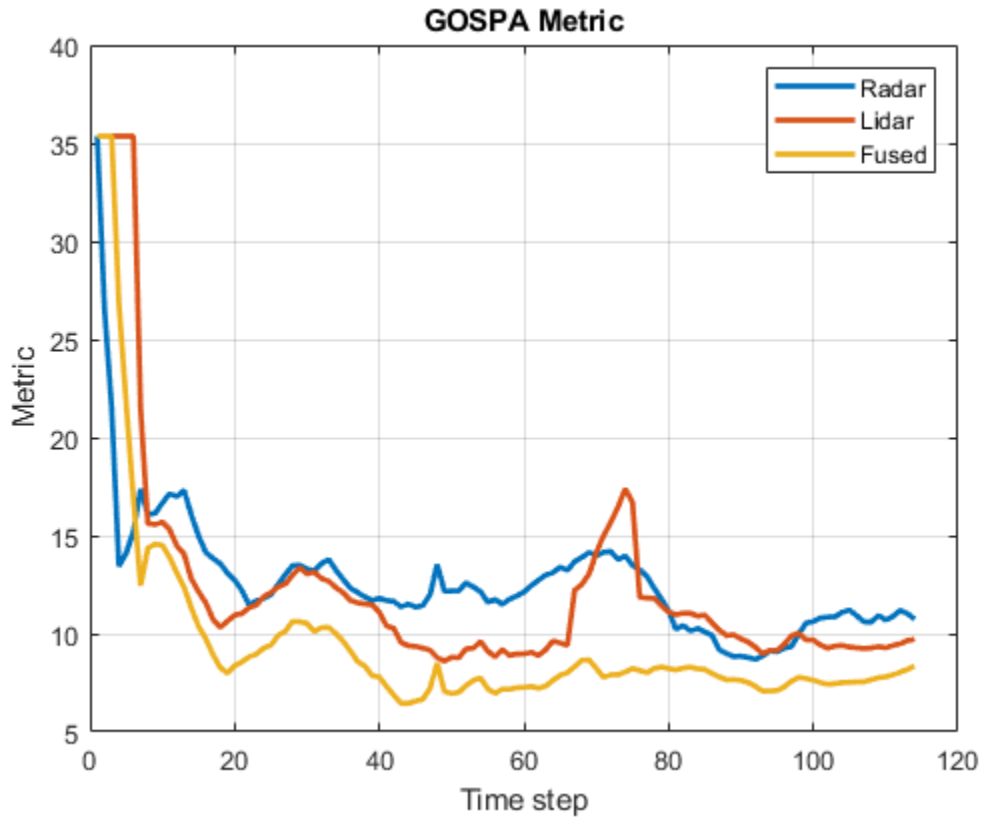
Closely-spaced targets

As mentioned earlier, this example uses a Euclidian-distance based clustering and bounding box fitting to feed the lidar data to a conventional tracking algorithm. Clustering algorithms typically suffer when objects are closely-spaced. With the detector configuration used in this example, when the passing vehicle approaches the vehicle in front of the ego vehicle, the detector clusters the point cloud from each vehicle into a bigger bounding box. You can notice in the animation below that the track drifted away from the vehicle center. Because the track was reported with higher certainty in its estimate for a few steps, the fused estimated was also affected initially. However, as the uncertainty increases, its association with the fused estimate becomes weaker. This is because the covariance intersection algorithm chooses a mixing weight for each assigned track based on the certainty of each estimate.
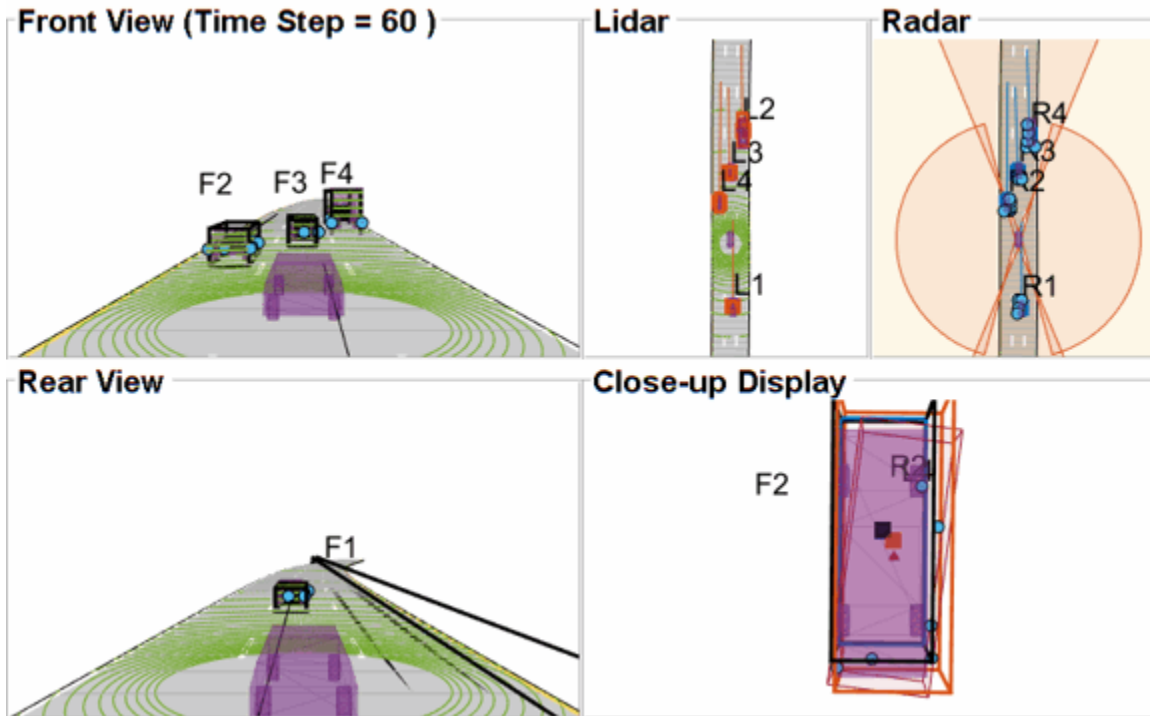
This effect is also captured in the GOSPA metric. You can notice in the GOSPA metric plot above that the lidar metric shows a peak around the 65th time step.

The radar tracks are not affected during this event because of two main reasons. Firstly, the radar sensor outputs range-rate information in each detection, which is different beyond noise-levels for the passing car as compared to the slower moving car. This results in an increased statistical distance between detections from individual cars. Secondly, extended object trackers evaluate multiple possible clustering hypothesis against predicted tracks, which results in rejection of improper clusters and acceptance of proper clusters. Note that for extended object trackers to properly choose the best clusters, the filter for the track must be robust to a degree that can capture the difference between two clusters. For example, a track with high process noise and highly uncertain dimensions may not be able to properly claim a cluster because of its premature age and higher flexibility to account for uncertain events.

Targets at long range

As targets recede away from the radar sensors, the accuracy of the measurements degrade because of reduced signal-to-noise ratio at the detector and the limited resolution of the sensor. This results in high uncertainty in the measurements, which in turn reduces the track accuracy. Notice in the close-up display below that the track estimate from the radar is further away from the ground truth for the radar sensor and is reported with a higher uncertainty. However, the lidar sensor reports enough measurements in the point cloud to generate a "shrunk" bounding box. The shrinkage effect modeled in the measurement model for lidar tracking algorithm allows the tracker to maintain a track with correct dimensions. In such situations, the lidar mixing weight is higher than the radar and allows the fused estimate to be more accurate than the radar estimate.
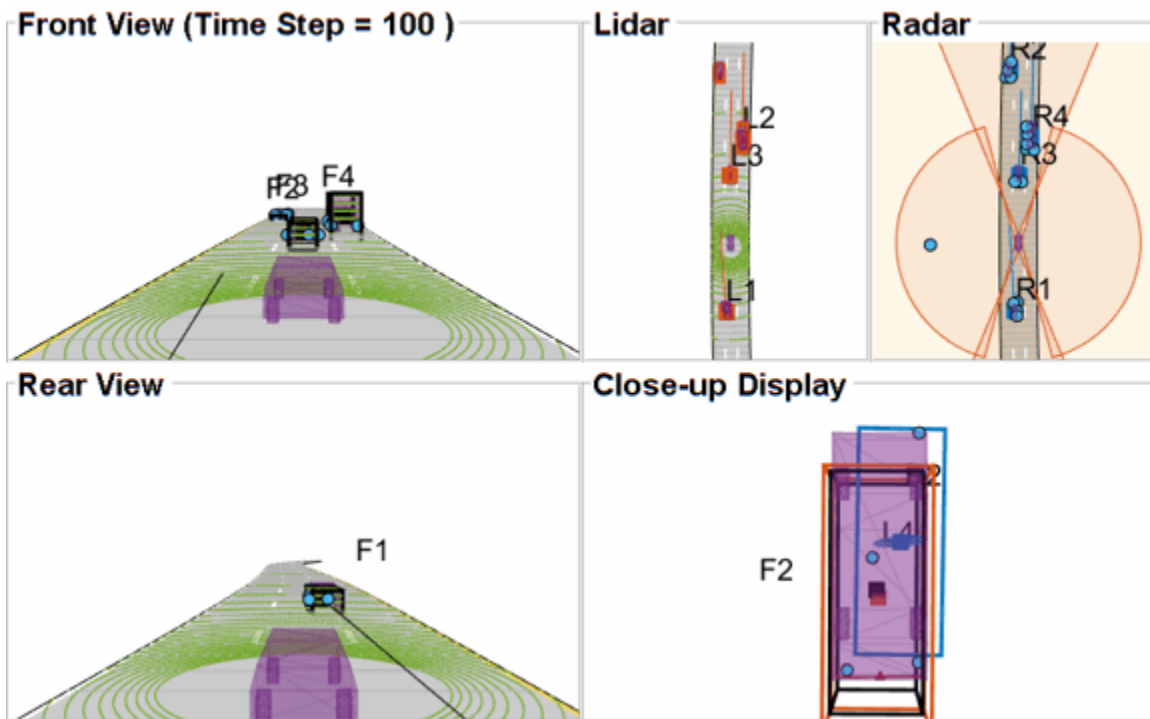
Summary

In this example, you learned how to setup a track-level fusion algorithm for fusing tracks from radar and lidar sensors. You also learned how to evaluate a tracking algorithm using the Generalized Optimal Subpattern Metric and its associated components.

Utility Functions

collectSensorData

A function to generate radar and lidar measurements at the current time-step.

```
function [radarDetections, ptCloud, egoPose] = helperCollectSensorData(egoVehicle, radars, lidar

% Current poses of targets with respect to ego vehicle
tgtPoses = targetPoses(egoVehicle);

radarDetections = cell(0,1);
for i = 1:numel(radars)
    thisRadarDetections = step(radars{i},tgtPoses,time);
    radarDetections = [radarDetections;thisRadarDetections]; %#ok<AGROW>
end

% Generate point cloud from lidar
rdMesh = roadMesh(egoVehicle);
ptCloud = step(lidar, tgtPoses, rdMesh, time);

% Compute pose of ego vehicle to track in scenario frame. Typically
% obtained using an INS system. If unavailable, this can be set to
% "origin" to track in Ego vehicle's frame.
egoPose = pose(egoVehicle);
```

```
end
```

radar2cental

A function to transform a track in the radar state-space to a track in the central state-space.

```
function centralTrack = radar2central(radarTrack)

% Initialize a track of the correct state size
centralTrack = objectTrack('State',zeros(10,1),...
    'StateCovariance',eye(10));

% Sync properties of radarTrack except State and StateCovariance with
% radarTrack See syncTrack defined below.
centralTrack = syncTrack(centralTrack,radarTrack);

xRadar = radarTrack.State;
PRadar = radarTrack.StateCovariance;

H = zeros(10,7); % Radar to central linear transformation matrix
H(1,1) = 1;
H(2,2) = 1;
H(3,3) = 1;
H(4,4) = 1;
H(5,5) = 1;
H(8,6) = 1;
H(9,7) = 1;

xCentral = H*xRadar;  % Linear state transformation
PCentral = H*PRadar*H'; % Linear covariance transformation

PCentral([6 7 10],[6 7 10]) = eye(3); % Unobserved states

% Set state and covariance of central track
centralTrack.State = xCentral;
centralTrack.StateCovariance = PCentral;

end
```

central2radar

A function to transform a track in the central state-space to a track in the radar state-space.

```
function radarTrack = central2radar(centralTrack)

% Initialize a track of the correct state size
radarTrack = objectTrack('State',zeros(7,1),...
    'StateCovariance',eye(7));

% Sync properties of centralTrack except State and StateCovariance with
% radarTrack See syncTrack defined below.
radarTrack = syncTrack(radarTrack,centralTrack);

xCentral = centralTrack.State;
PCentral = centralTrack.StateCovariance;

H = zeros(7,10); % Central to radar linear transformation matrix
```

```
H(1,1) = 1;
H(2,2) = 1;
H(3,3) = 1;
H(4,4) = 1;
H(5,5) = 1;
H(6,8) = 1;
H(7,9) = 1;

xRadar = H*xCentral;  % Linear state transformation
PRadar = H*PCentral*H'; % Linear covariance transformation

% Set state and covariance of radar track
radarTrack.State = xRadar;
radarTrack.StateCovariance = PRadar;
end
```

syncTrack

A function to syncs properties of one track with another except the "State" and "StateCovariance" property

```
function tr1 = syncTrack(tr1,tr2)
props = properties(tr1);
notState = ~strcmpi(props,'State');
notCov = ~strcmpi(props,'StateCovariance');

props = props(notState & notCov);
for i = 1:numel(props)
    tr1.(props{i}) = tr2.(props{i});
end
end
```

pose

A function to return pose of the ego vehicle as a structure.

```
function egoPose = pose(egoVehicle)
egoPose.Position = egoVehicle.Position;
egoPose.Velocity = egoVehicle.Velocity;
egoPose.Yaw = egoVehicle.Yaw;
egoPose.Pitch = egoVehicle.Pitch;
egoPose.Roll = egoVehicle.Roll;
end
```

helperLidarDistance

Function to calculate a normalized distance between the estimate of a track in radar state-space and the assigned ground truth.

```
function dist = helperLidarDistance(track, truth)

% Calculate the actual values of the states estimated by the tracker

% Center is different than origin and the trackers estimate the center
rOriginToCenter = -truth.OriginOffset(:) + [0;0;truth.Height/2];
rot = quaternion([truth.Yaw truth.Pitch truth.Roll],'eulerd','ZYX','frame');
actPos = truth.Position(:) + rotatepoint(rot,rOriginToCenter')';
```

```
% Actual speed and z-rate
actVel = [norm(truth.Velocity(1:2));truth.Velocity(3)];

% Actual yaw
actYaw = truth.Yaw;

% Actual dimensions.
actDim = [truth.Length;truth.Width;truth.Height];

% Actual yaw rate
actYawRate = truth.AngularVelocity(3);

% Calculate error in each estimate weighted by the "requirements" of the
% system. The distance specified using Mahalanobis distance in each aspect
% of the estimate, where covariance is defined by the "requirements". This
% helps to avoid skewed distances when tracks under/over report their
% uncertainty because of inaccuracies in state/measurement models.

% Positional error.
estPos = track.State([1 2 6]);
reqPosCov = 0.1*eye(3);
e = estPos - actPos;
d1 = sqrt(e'/reqPosCov*e);

% Velocity error
estVel = track.State([3 7]);
reqVelCov = 5*eye(2);
e = estVel - actVel;
d2 = sqrt(e'/reqVelCov*e);

% Yaw error
estYaw = track.State(4);
reqYawCov = 5;
e = estYaw - actYaw;
d3 = sqrt(e'/reqYawCov*e);

% Yaw-rate error
estYawRate = track.State(5);
reqYawRateCov = 1;
e = estYawRate - actYawRate;
d4 = sqrt(e'/reqYawRateCov*e);

% Dimension error
estDim = track.State([8 9 10]);
reqDimCov = eye(3);
e = estDim - actDim;
d5 = sqrt(e'/reqDimCov*e);

% Total distance
dist = d1 + d2 + d3 + d4 + d5;

end
```

helperRadarDistance

Function to calculate a normalized distance between the estimate of a track in radar state-space and the assigned ground truth.

```matlab
function dist = helperRadarDistance(track, truth)

% Calculate the actual values of the states estimated by the tracker

% Center is different than origin and the trackers estimate the center
rOriginToCenter = -truth.OriginOffset(:) + [0;0;truth.Height/2];
rot = quaternion([truth.Yaw truth.Pitch truth.Roll],'eulerd','ZYX','frame');
actPos = truth.Position(:) + rotatepoint(rot,rOriginToCenter')';
actPos = actPos(1:2); % Only 2-D

% Actual speed
actVel = norm(truth.Velocity(1:2));

% Actual yaw
actYaw = truth.Yaw;

% Actual dimensions. Only 2-D for radar
actDim = [truth.Length;truth.Width];

% Actual yaw rate
actYawRate = truth.AngularVelocity(3);

% Calculate error in each estimate weighted by the "requirements" of the
% system. The distance specified using Mahalanobis distance in each aspect
% of the estimate, where covariance is defined by the "requirements". This
% helps to avoid skewed distances when tracks under/over report their
% uncertainty because of inaccuracies in state/measurement models.

% Positional error
estPos = track.State([1 2]);
reqPosCov = 0.1*eye(2);
e = estPos - actPos;
d1 = sqrt(e'/reqPosCov*e);

% Speed error
estVel = track.State(3);
reqVelCov = 5;
e = estVel - actVel;
d2 = sqrt(e'/reqVelCov*e);

% Yaw error
estYaw = track.State(4);
reqYawCov = 5;
e = estYaw - actYaw;
d3 = sqrt(e'/reqYawCov*e);

% Yaw-rate error
estYawRate = track.State(5);
reqYawRateCov = 1;
e = estYawRate - actYawRate;
d4 = sqrt(e'/reqYawRateCov*e);

% Dimension error
estDim = track.State([6 7]);
reqDimCov = eye(2);
e = estDim - actDim;
d5 = sqrt(e'/reqDimCov*e);
```

```matlab
% Total distance
dist = d1 + d2 + d3 + d4 + d5;

% A constant penality for not measuring 3-D state
dist = dist + 3;

end
```

helperRadarLidarFusionFcn

Function to fuse states and state covariances in central track state-space

```matlab
function [x,P] = helperRadarLidarFusionFcn(xAll,PAll)
n = size(xAll,2);
dets = zeros(n,1);

% Initialize x and P
x = xAll(:,1);
P = PAll(:,:,1);

onlyLidarStates = false(10,1);
onlyLidarStates([6 7 10]) = true;

% Only fuse this information with lidar
xOnlyLidar = xAll(onlyLidarStates,:);
POnlyLidar = PAll(onlyLidarStates,onlyLidarStates,:);

% States and covariances for intersection with radar and lidar both
xToFuse = xAll(~onlyLidarStates,:);
PToFuse = PAll(~onlyLidarStates,~onlyLidarStates,:);

% Sorted order of determinants. This helps to sequentially build the
% covariance with comparable determinations. For example, two large
% covariances may intersect to a smaller covariance, which is comparable to
% the third smallest covariance.
for i = 1:n
    dets(i) = det(PToFuse(1:2,1:2,i));
end
[~,idx] = sort(dets,'descend');
xToFuse = xToFuse(:,idx);
PToFuse = PToFuse(:,:,idx);

% Initialize fused estimate
thisX = xToFuse(:,1);
thisP = PToFuse(:,:,1);

% Sequential fusion
for i = 2:n
    [thisX,thisP] = fusecovintUsingPos(thisX, thisP, xToFuse(:,i), PToFuse(:,:,i));
end

% Assign fused states from all sources
x(~onlyLidarStates) = thisX;
P(~onlyLidarStates,~onlyLidarStates,:) = thisP;

% Fuse some states only with lidar source
valid = any(abs(xOnlyLidar) > 1e-6,1);
```

```matlab
    xMerge = xOnlyLidar(:,valid);
    PMerge = POnlyLidar(:,:,valid);

    if sum(valid) > 1
        [xL,PL] = fusecovint(xMerge,PMerge);
    elseif sum(valid) == 1
        xL = xMerge;
        PL = PMerge;
    else
        xL = zeros(3,1);
        PL = eye(3);
    end

    x(onlyLidarStates) = xL;
    P(onlyLidarStates,onlyLidarStates) = PL;

end

function [x,P] = fusecovintUsingPos(x1,P1,x2,P2)
% Covariance intersection in general is employed by the following
% equations:
% P^-1 = w1*P1^-1 + w2*P2^-1
% x = P*(w1*P1^-1*x1 + w2*P2^-1*x2);
% where w1 + w2 = 1
% Usually a scalar representative of the covariance matrix like "det" or
% "trace" of P is minimized to compute w. This is offered by the function
% "fusecovint". However. in this case, the w are chosen by minimizing the
% determinants of "positional" covariances only.
n = size(x1,1);
idx = [1 2];
detP1pos = det(P1(idx,idx));
detP2pos = det(P2(idx,idx));
w1 = detP2pos/(detP1pos + detP2pos);
w2 = detP1pos/(detP1pos + detP2pos);
I = eye(n);

P1inv = I/P1;
P2inv = I/P2;

Pinv = w1*P1inv + w2*P2inv;
P = I/Pinv;

x = P*(w1*P1inv*x1 + w2*P2inv*x2);

end
```

References

[1] Lang, Alex H., et al. "PointPillars: Fast encoders for object detection from point clouds."
Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2019.

[2] Zhou, Yin, and Oncel Tuzel. "Voxelnet: End-to-end learning for point cloud based 3d object
detection." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018.

[3] Yang, Bin, Wenjie Luo, and Raquel Urtasun. "Pixor: Real-time 3d object detection from point clouds." Proceedings of the IEEE conference on Computer Vision and Pattern Recognition. 2018.

Tuning a Multi-Object Tracker

This example shows how to tune and run a tracker to track multiple objects in the scene. The example explains and demonstrates the importance of key properties of the trackers in the Sensor Fusion and Tracking Toolbox.

Default Tracker Configuration

In order to test the capability of a tracker to track multiple objects, you set up a basic scenario. In the scenario, you define three objects with each moving along a straight line at a constant velocity. Initially, you set the object velocities to be 48 m/s, 60 m/s, and 72 m/s, respectively.

```
stopTime = 10;
v = 60;
scenario = trackingScenario;
scenario.StopTime = stopTime;
scenario.UpdateRate = 0;
p1 = platform(scenario);
p1.Trajectory = waypointTrajectory([30 20 0; 30 .8*v*stopTime 0], [0 stopTime]);
p2 = platform(scenario);
p2.Trajectory = waypointTrajectory([0 0 0; 0 v*stopTime 0], [0 stopTime]);
p3 = platform(scenario);
p3.Trajectory = waypointTrajectory([-30 -20 0; -30 1.2*v*stopTime 0], [0 stopTime]);
```

In addition, you define a radar that stares at the scene and updates 5 times per second. You mount it on a platform located on the side of the moving objects.

```
pRadar = platform(scenario);
pRadar.Trajectory = kinematicTrajectory('Position', [-v*stopTime 0.5*v*stopTime 0]);
radar = monostaticRadarSensor(1, 'No scanning', 'UpdateRate', 5, ...
    'MountingAngles', [0 0 0], 'AzimuthResolution', 1, ...
    'FieldOfView', [100 1], 'HasINS', true, 'DetectionCoordinates', 'Scenario', ...
    'MaxUnambiguousRange', 1000);
pRadar.Sensors = radar;
```

You create a theater plot to display the scene.

```
fig = figure;
ax = axes(fig);
tp = theaterPlot('Parent', ax, 'XLimits', [-11*v 100], 'YLimits', [-50 15*v], 'ZLimits', [-100 10
rp = platformPlotter(tp, 'DisplayName', 'Radar', 'Marker', 'd');
pp = platformPlotter(tp, 'DisplayName', 'Platforms');
dp = detectionPlotter(tp, 'DisplayName', 'Detections');
trp = trackPlotter(tp, 'DisplayName', 'Tracks', 'ConnectHistory', 'on', 'ColorizeHistory', 'on')
covp = coveragePlotter(tp, 'DisplayName', 'Radar Coverage', 'Alpha', [0.1 0]);
```

Finally, you create a default `trackerGNN` object, run the scenario, and observe the results. You use `trackGOSPAMetric` to evaluate the tracker performance.

```
tracker = trackerGNN;
tgm = trackGOSPAMetric("Distance","posabserr");
gospa = zeros(1,51); % number of timesteps is 51
i = 0;

% Define the random number generator seed for repeatable results
s = rng(2019);
while advance(scenario)
```

```
    % Get detections
    dets = detect(scenario);

    % Update the tracker
    if isLocked(tracker) || ~isempty(dets)
        [tracks, ~, ~, info] = tracker(dets, scenario.SimulationTime);
    end

    % Evaluate GOSPA
    i = i + 1;
    truth = platformPoses(scenario);
    gospa(i) = tgm(tracks, truth);

    % Update the display
    updateDisplay(rp, pp, dp, trp, covp, scenario, dets, tracks);
end
```
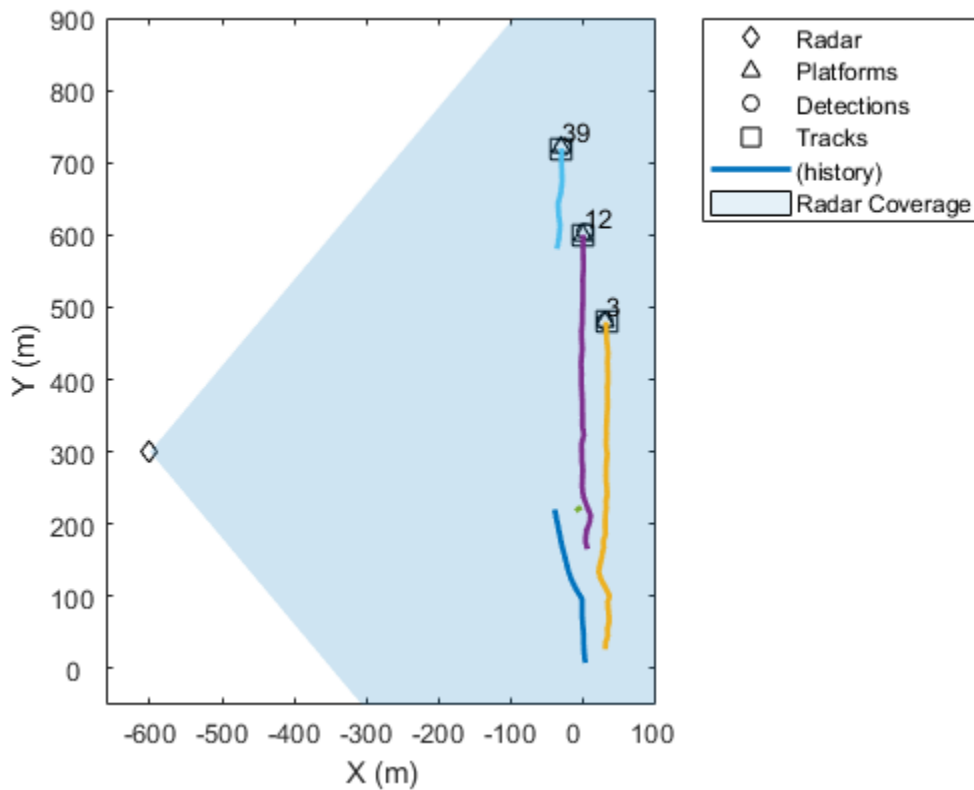


```
rng(s)
figure
plot(gospa)
title('Generalized OSPA vs. Timestep')
```
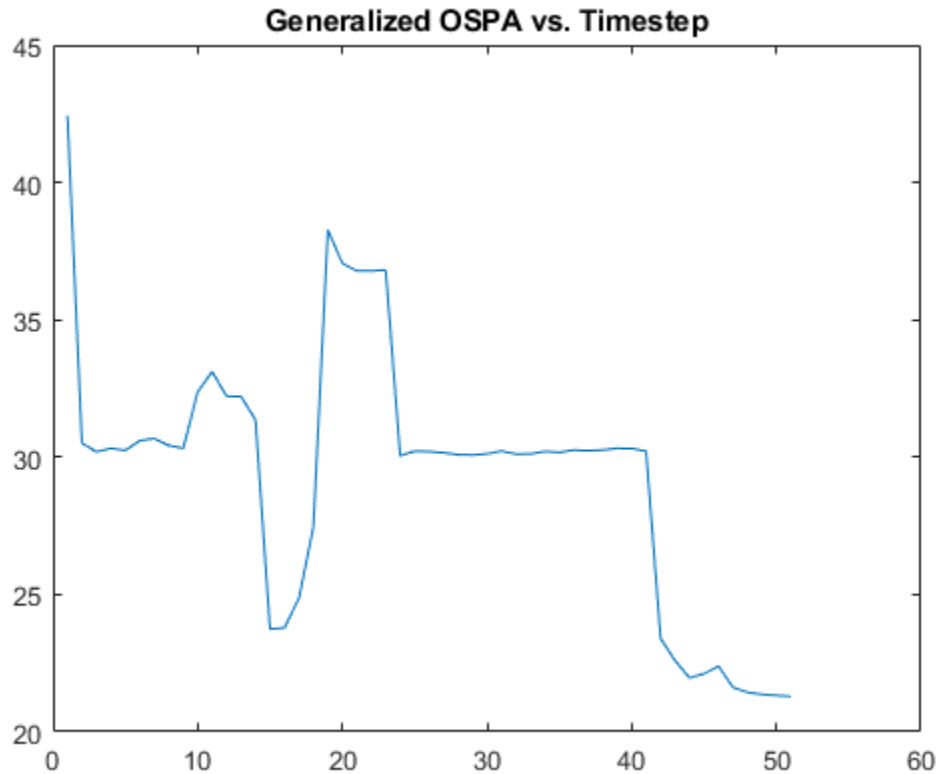
You observe that the tracker was not able to track the three objects. At some point, additional tracks are confirmed and shown in addition to the three expected tracks for the three moving objects. As a result, the value of the GOSPA metric increases. Note that lower values of the GOSPA metric indicate better tracker performance.

Assignment Threshold

You look at the `info` struct that the tracker outputs and observe that the `CostMatrix` and the `Assignments` do not show an assignment of pairs of tracks and objects you expect to happen. This means that the `AssignmentThreshold` is too small and should be increased. Increase the assignment threshold to 50.

```
release(tracker);
tracker.AssignmentThreshold = 50;
rerunScenario(scenario, tracker, tgm, tp);
```
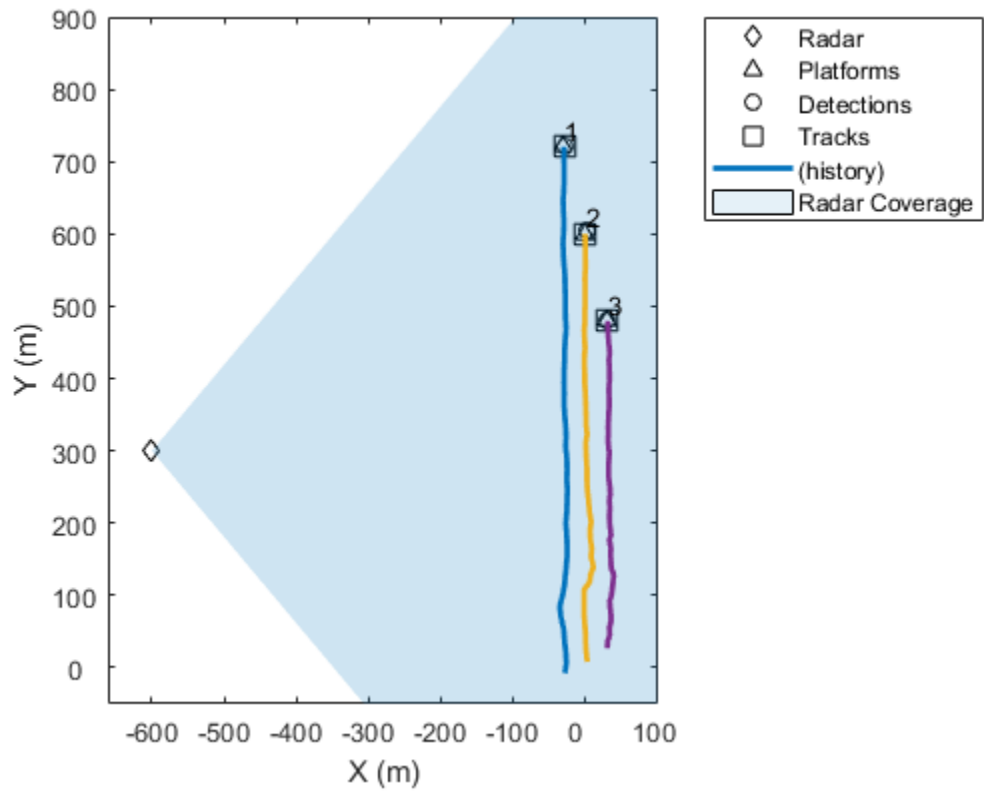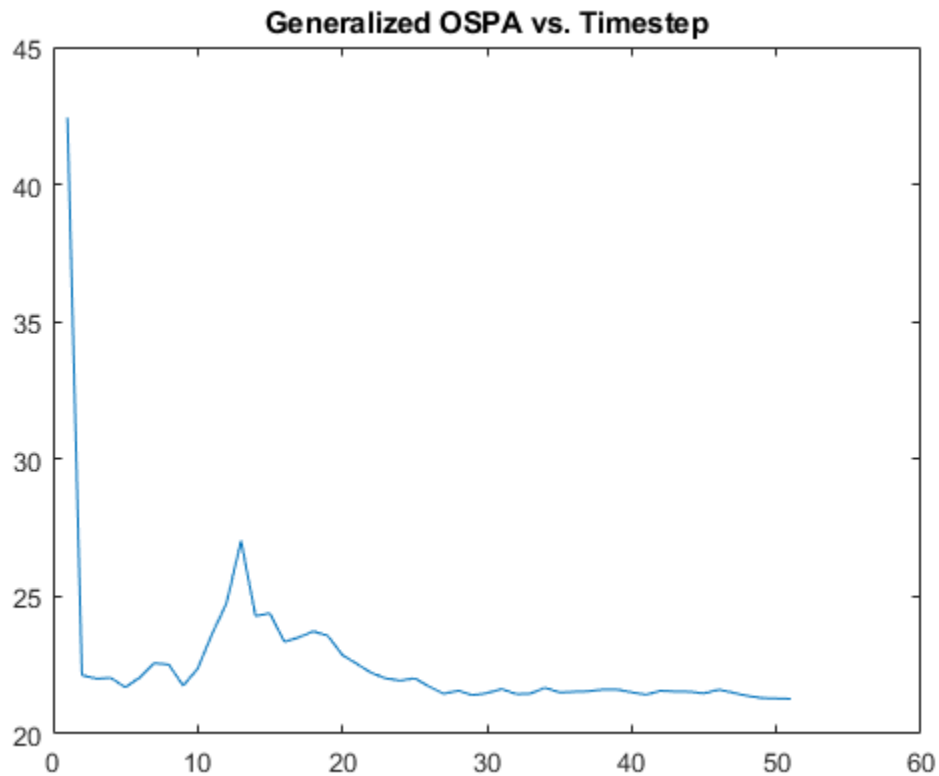
The assignment threshold property also has a maximum distance from a detection that you can set. This is the last element in the `AssignmentThreshold` value. You can use this value to speed the tracker up when handling large number of detections and tracks. Refer to the "How to Efficiently Track Large Numbers of Objects" on page 5-320 example for more details. Here, you set the value to `2000` instead of `inf`, which will reduce the number of combinations of tracks and detections that are used to calculate the assignment cost.

```
release(tracker);
tracker.AssignmentThreshold = [50 2000];
rerunScenario(scenario, tracker, tgm, tp);
```
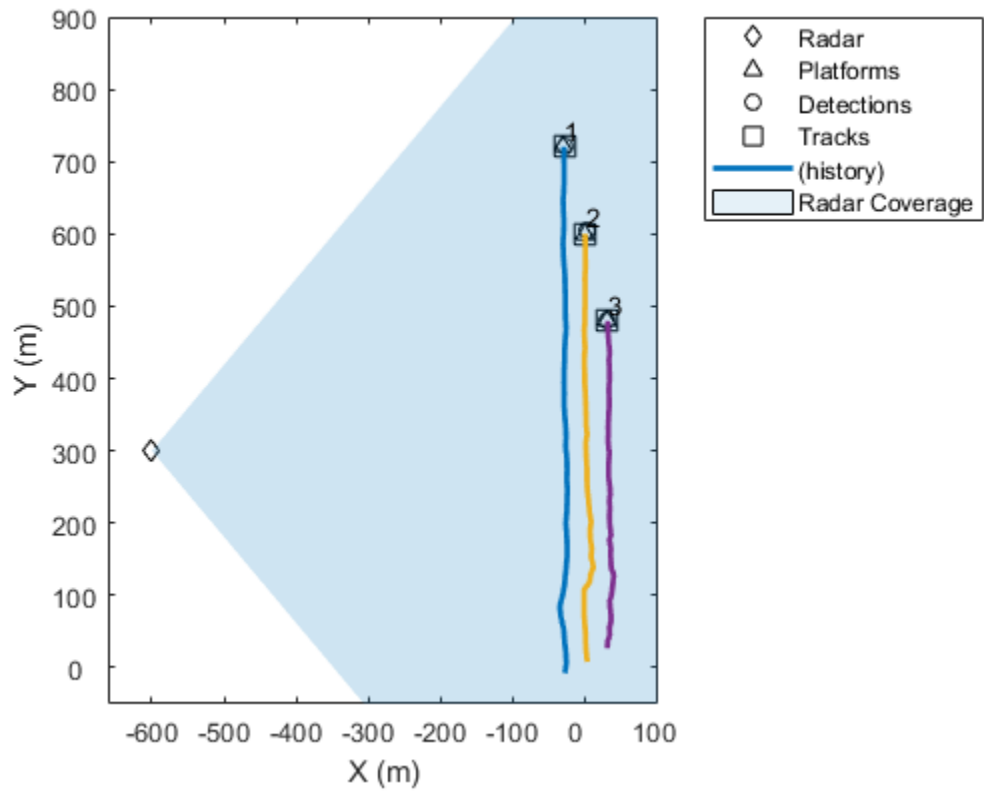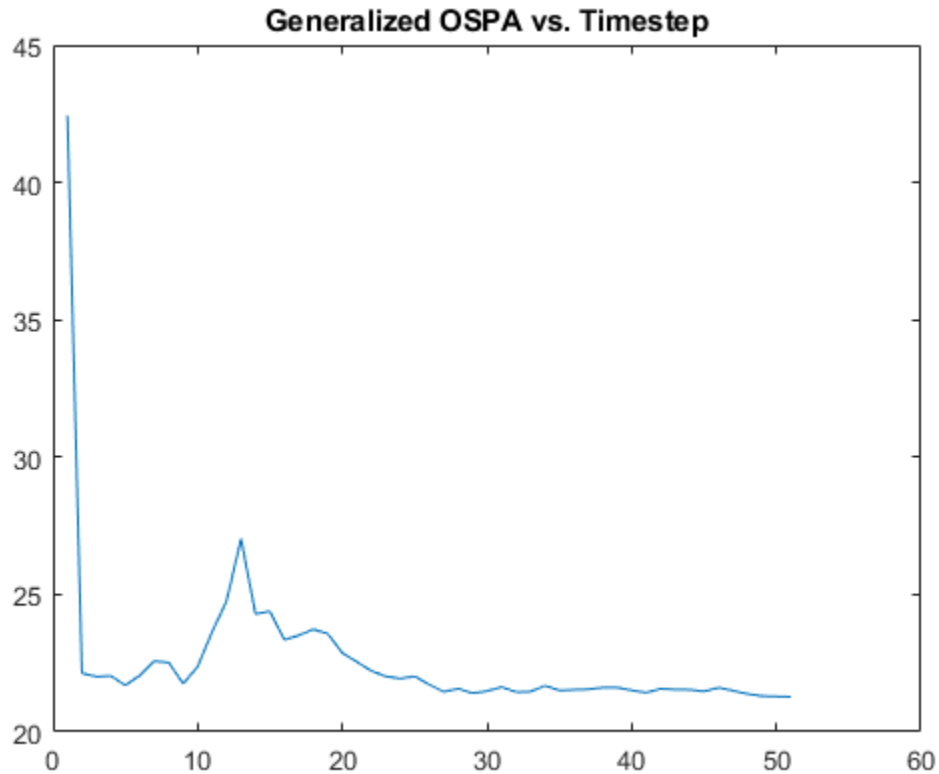
Filter Initialization Function

The results above show that the tracker is capable of maintaining three tracks for the three moving objects without creating false tracks. However, would the tracker's performance hold if the objects move at a faster speed? To check that, you modify the scenario and increase the object speeds to be 160, 200, and 240 m/s, respectively.

```
v = 200;
p1.Trajectory = waypointTrajectory([30 0 0; 30 0.8*v*stopTime 0], [0 stopTime]);
p2.Trajectory = waypointTrajectory([0 0 0; 0 v*stopTime 0], [0 stopTime]);
p3.Trajectory = waypointTrajectory([-30 0 0; -30 1.2*v*stopTime 0], [0 stopTime]);
pRadar.Trajectory = kinematicTrajectory('Position', [-v*stopTime 0.5*v*stopTime 0]);
tp.XLimits = [-100-v*stopTime 300];
tp.YLimits = [-100 100+v*1.2*stopTime];
release(radar);
radar.MaxUnambiguousRange = 3000;
rerunScenario(scenario, tracker, tgm, tp);
```

Generalized OSPA vs. Timestep
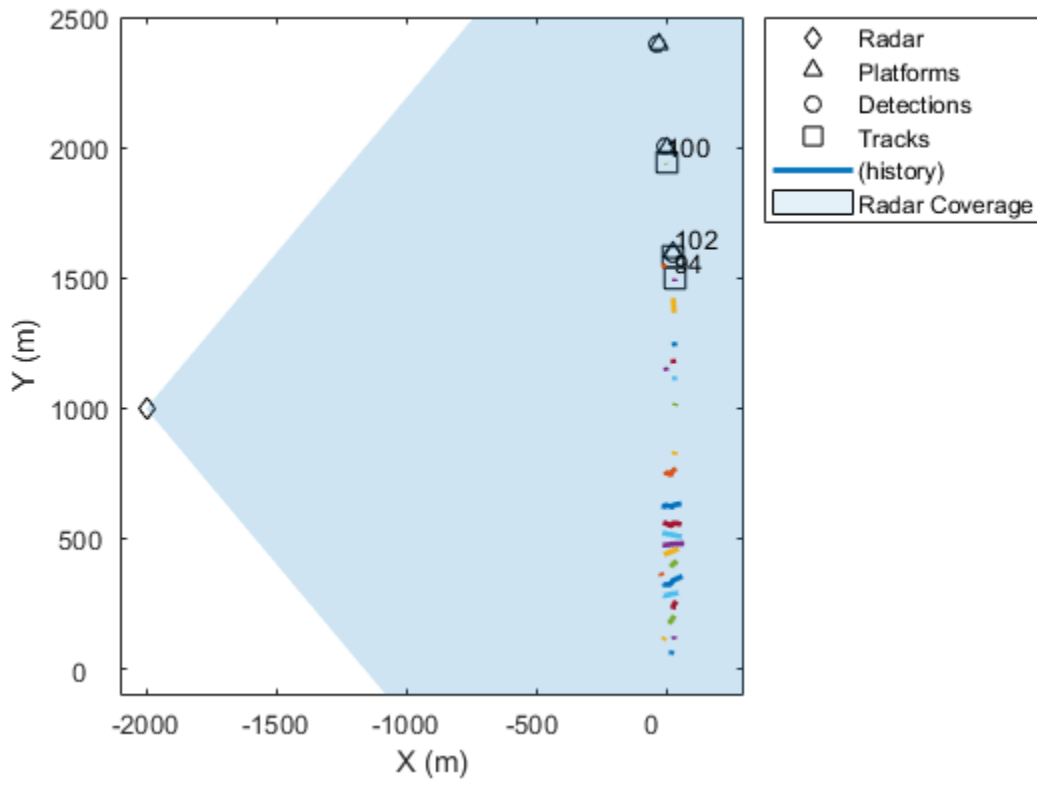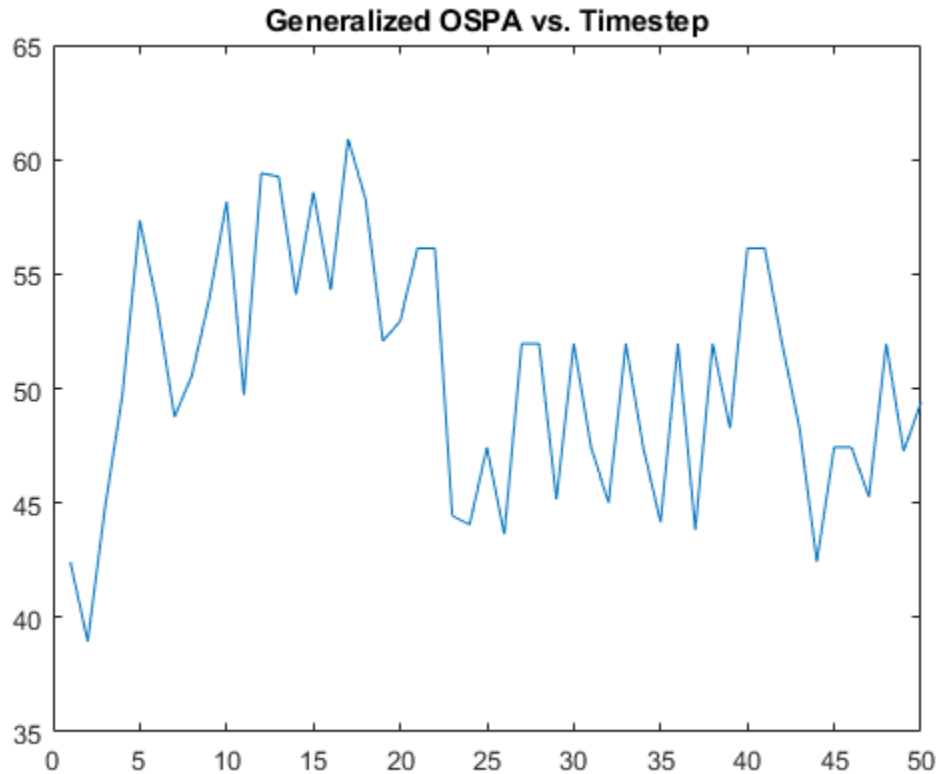
As you probably expected, the tracker could not establish stable tracks of the three objects. A possible solution is to increase the `AssignmentThreshold` even further to allow tracks to be updated. However, this option may increase the chance of random false tracks being assigned to multiple detections and getting confirmed. So, you choose to modify how a filter is being initialized.

You set the `FilterInitializationFcn` property to the function `initFastCVEKF`. The function is the same as the default `initcvekf` except it increases the uncertainty in the velocity components of the state. It allows the initial state to account for larger unknown velocity values, but once the track establishes, the uncertainty decreases again.

```
release(tracker)
tracker.FilterInitializationFcn = @initFastCVEKF;
rerunScenario(scenario, tracker, tgm, tp);
```
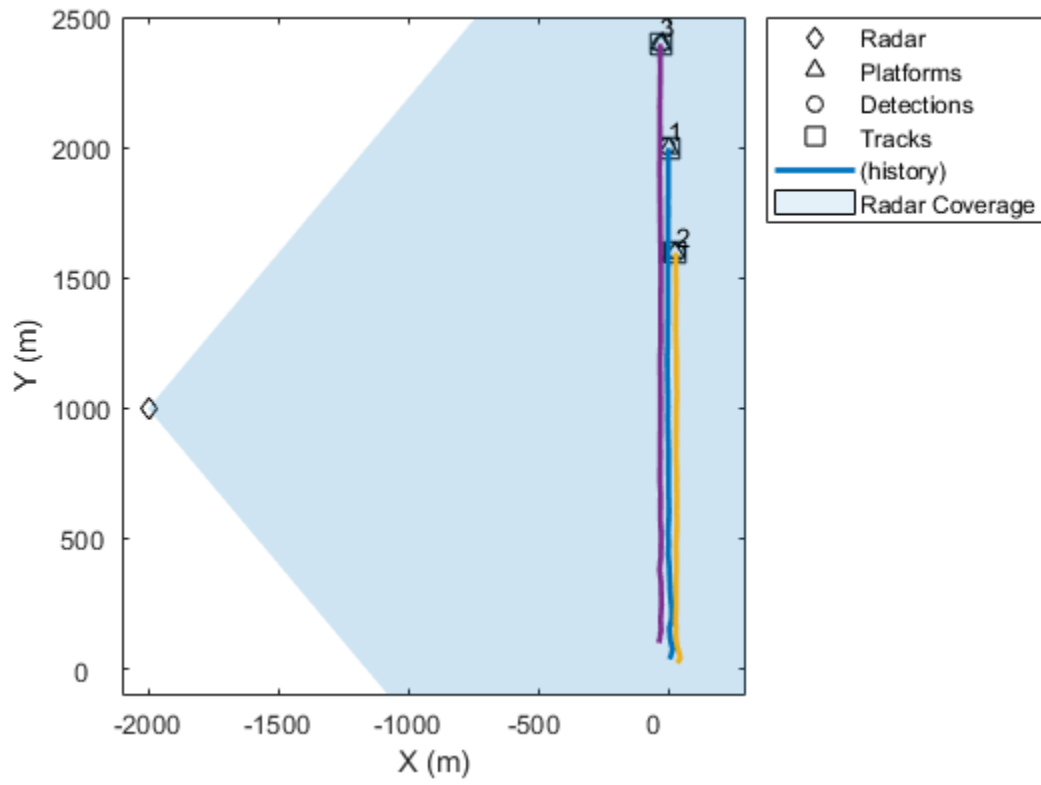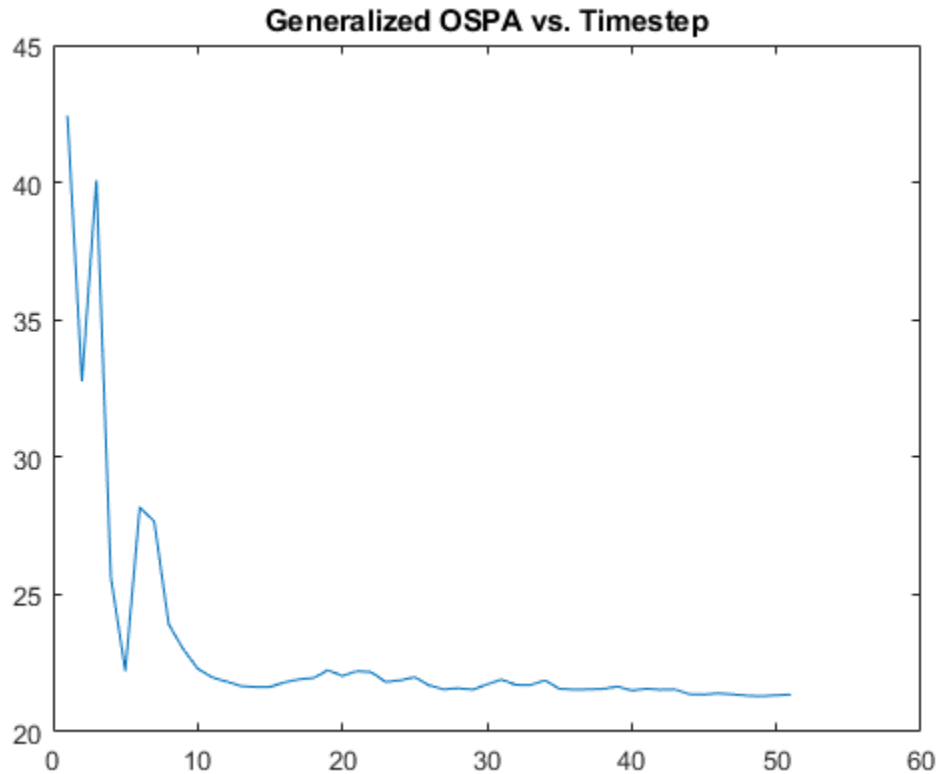
You observe that the tracker is once again able to maintain the tracks, and there are 3 tracks for the three moving objects, even though they are moving faster now. The GOSPA value is also reduced after a few steps.

Confirmation and Deletion Thresholds

You want to make sure that your tracker is robust to a higher false alarm rate. To do that, you configure the radar to have a false alarm rate that is 250 times higher than previously.

You zoom out to view a larger portion of the scene and to see if false tracks are created.

```
release(radar);
radar.FalseAlarmRate = 2.5e-4;
tp.XLimits = [-2100 300];
tp.YLimits = [-100  3100];
tp.ZLimits = [-1000 1000];
rerunScenario(scenario, tracker, tgm, tp);
```

Generalized OSPA vs. Timestep
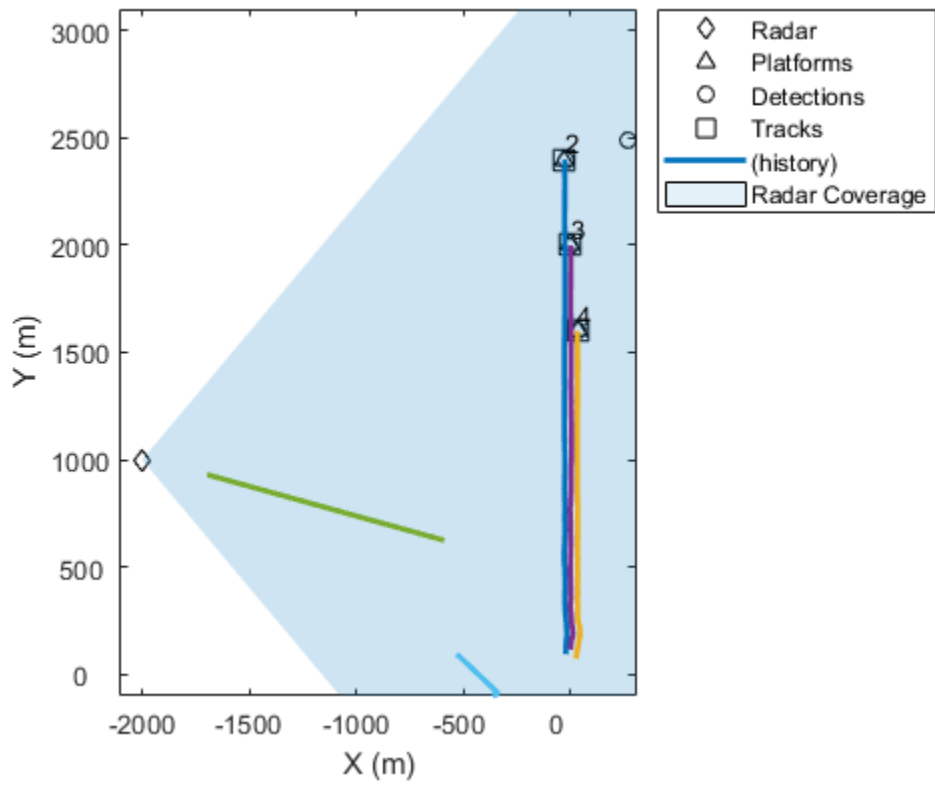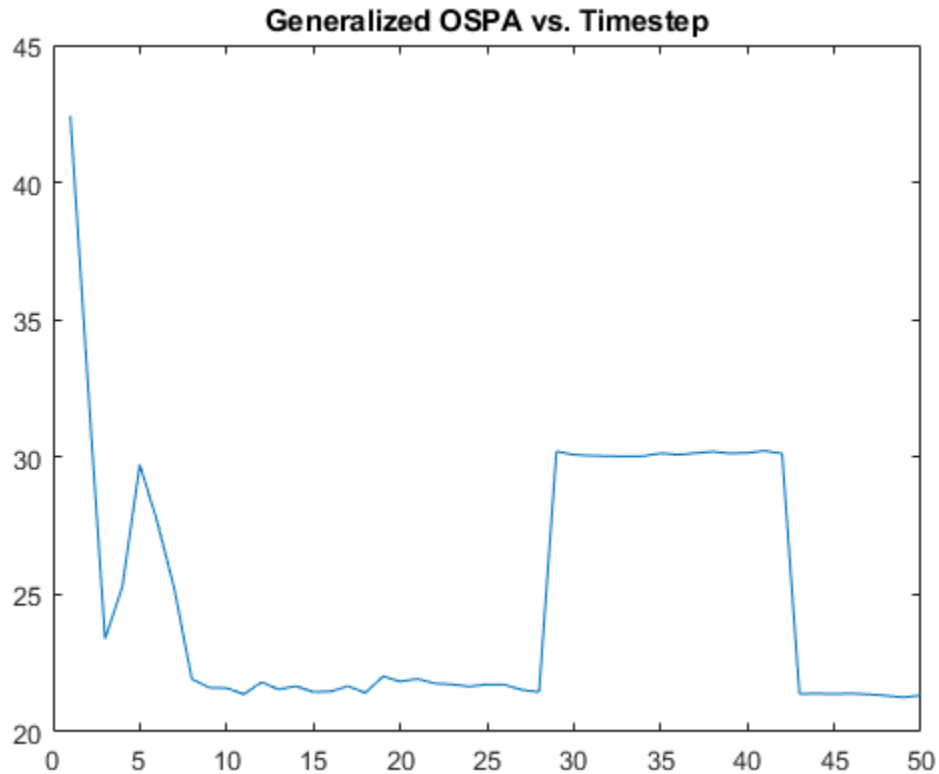
There are a few false tracks being created (not all of them shown) and they increase the GOSPA value.

You want to delete false tracks more quickly, for which you use the `DeletionThreshold` property. The default value for deletion threshold is [5 5], which requires 5 consecutive misses before a confirmed track is deleted. You can make the deletion process quicker by reducing this value to [3 3], or 3-out-of-3 misses. Alternatively, since the radar `DetectionProbability` is high, you can even delete a track after 2-out-of-3 misses, by setting:

```
release(tracker)
tracker.DeletionThreshold = [2 3];
rerunScenario(scenario, tracker, tgm, tp);
```

Generalized OSPA vs. Timestep
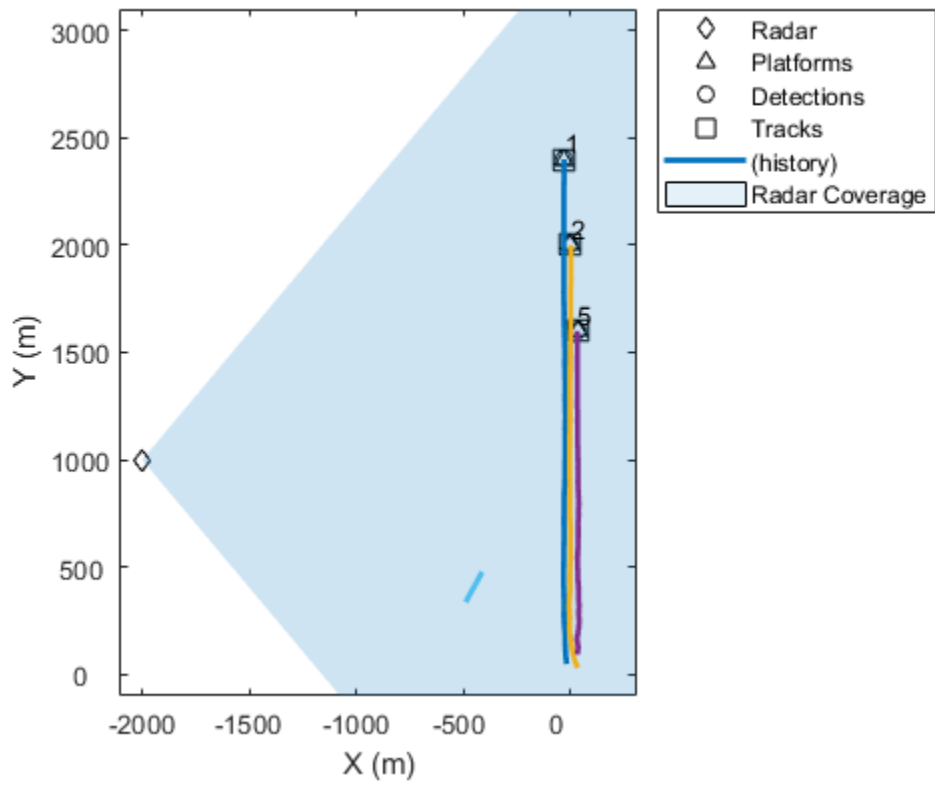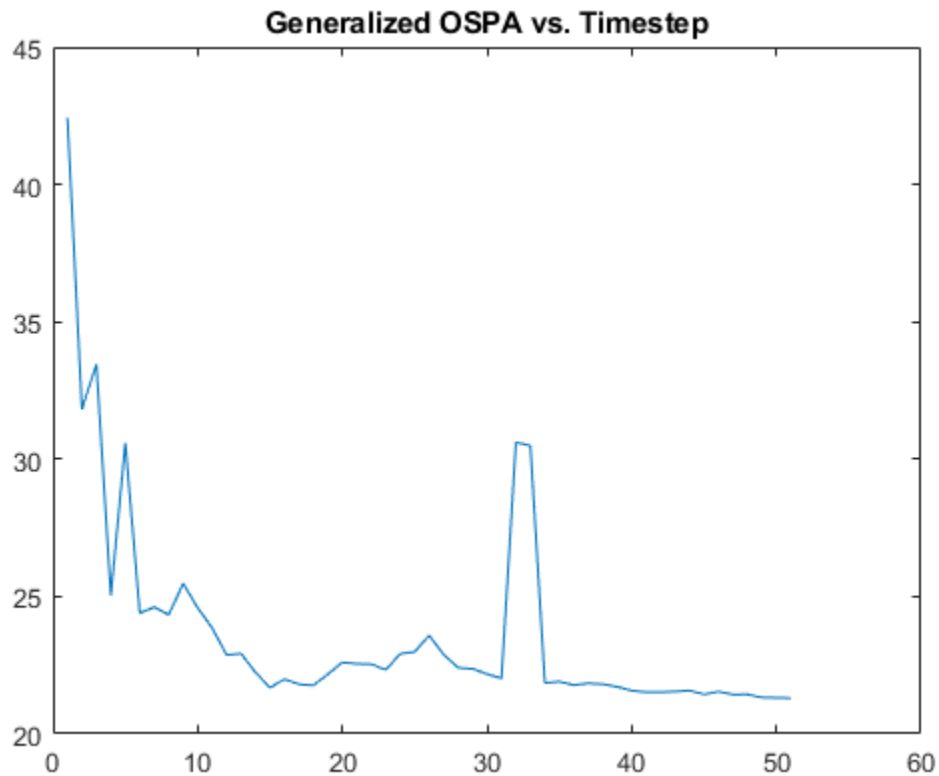
As expected, reducing the number of steps it takes to delete tracks decreases the GOSPA value because these false tracks are short-lived. However, these false tracks are still confirmed and deteriorate the overall tracking quality.

So, you want to make the confirmation of new tracks more stringent in attempt to reduce the number of false tracks. Consider the tracker `ConfirmationThreshold` property.

```
disp(tracker.ConfirmationThreshold);
```

```
     2     3
```

The value shows that the tracker confirms every track if it is assigned two detections in the first three updates. You decide to make it harder to confirm a track and reset the value to be 3-out-of-4 assignments.

```
release(tracker)
tracker.ConfirmationThreshold = [3 4];
rerunScenario(scenario, tracker, tgm, tp);
```
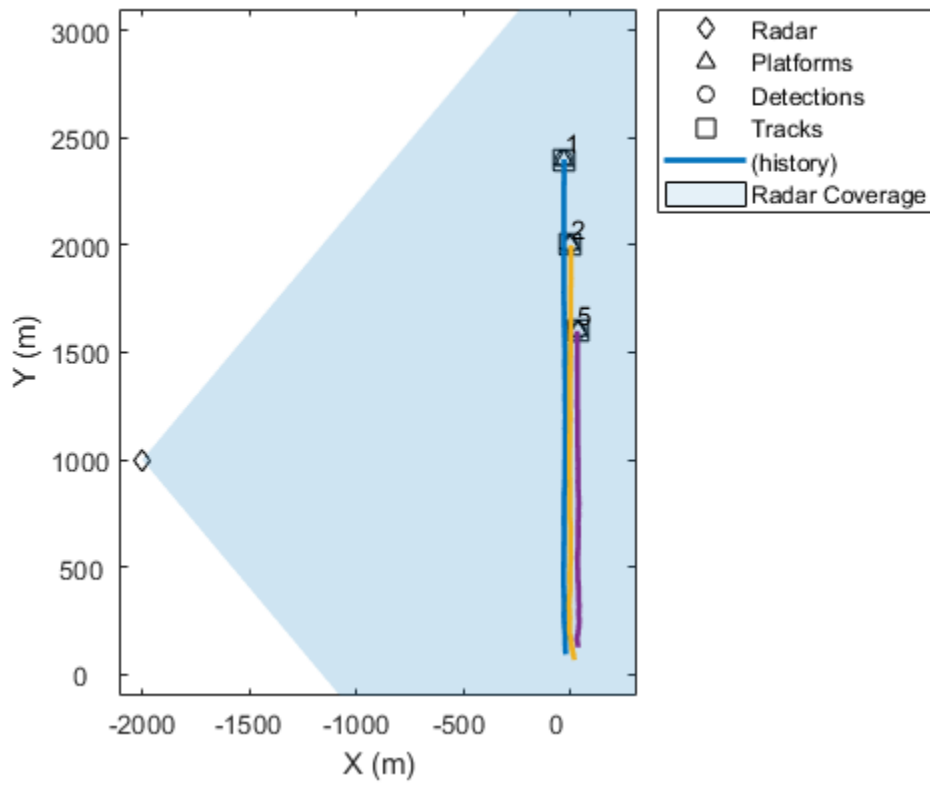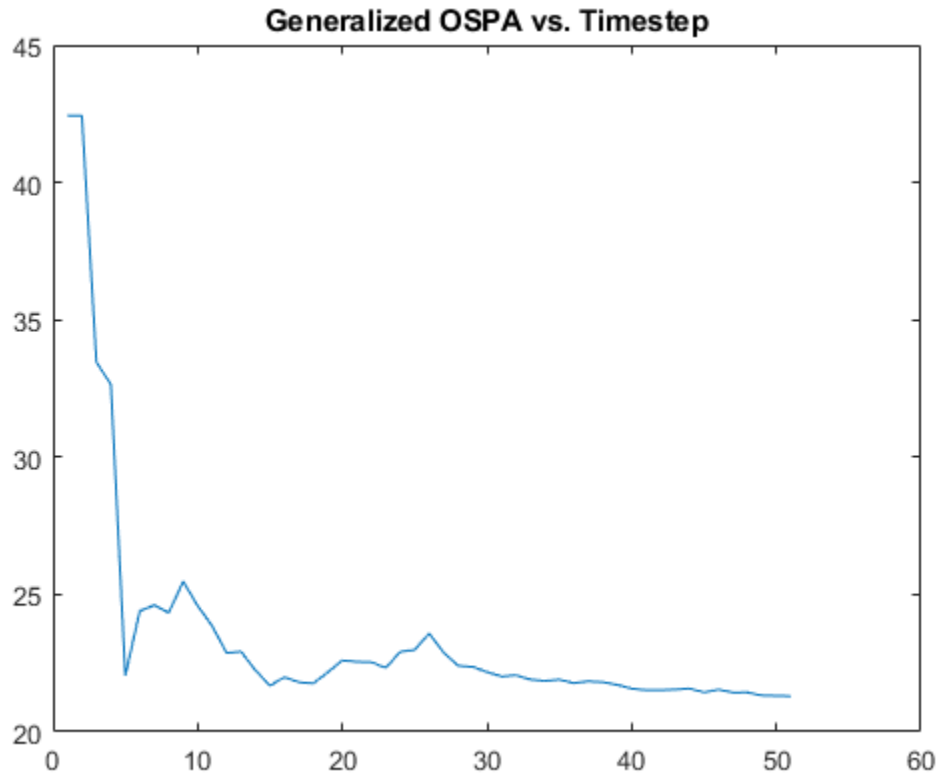
By making track confirmation more stringent you were able to eliminate the false tracks. As a result, the GOSPA values are down to around 20 again, except for the first few steps.

Maximum Number of Tracks

Making the confirmation more stringent allowed you to eliminate false tracks. However, the tracker still initializes a track for each false detection, which you can observe by looking at the number of tracks the tracker currently maintains.

The number of tracks fluctuates through the lifetime of the tracker. However, you don't want it to exceed the maximum number of tracks that the tracker can maintain, defined by `MaxNumTracks`, which is by default 100. If the tracker exceeds this number, it issues a warning that a new track could not be added, but the execution can continue.

Increase the maximum number of tracks to allow the tracker to track even with a higher false alarm rate without issuing the warning. You also want to make track confirmation even more stringent to reduce the number of false tracks and reduce the GOSPA metric.

```
release(tracker)
tracker.MaxNumTracks = 200;
tracker.ConfirmationThreshold = [5 6];

release(radar)
radar.FalseAlarmRate = 1e-3;
rerunScenario(scenario, tracker, tgm, tp);
```

Generalized OSPA vs. Timestep
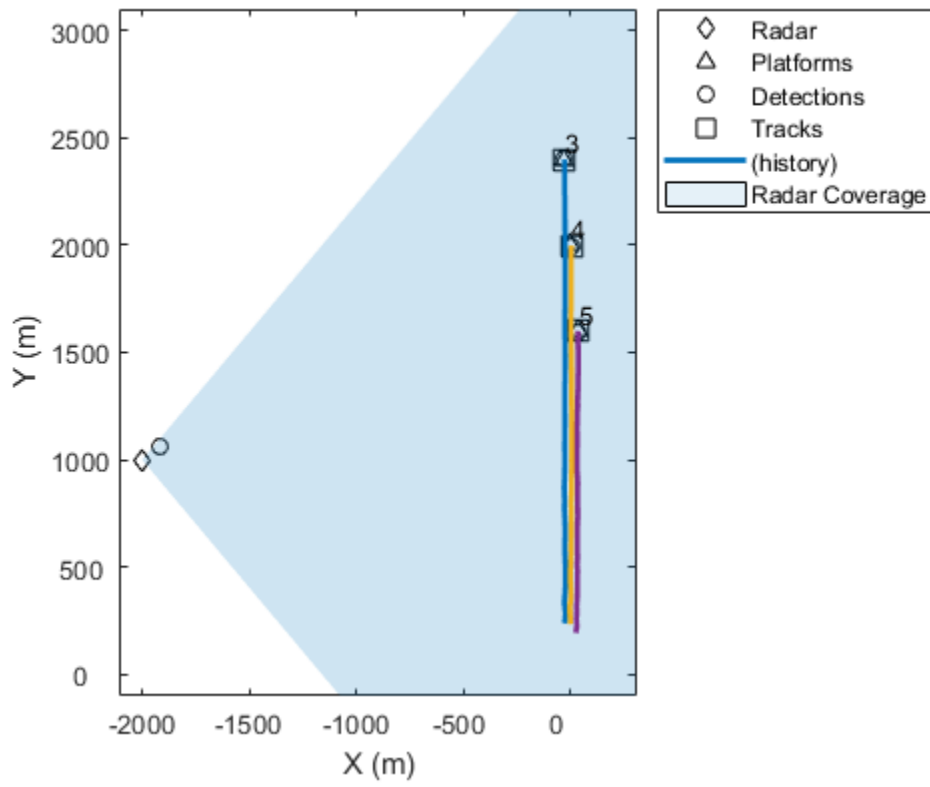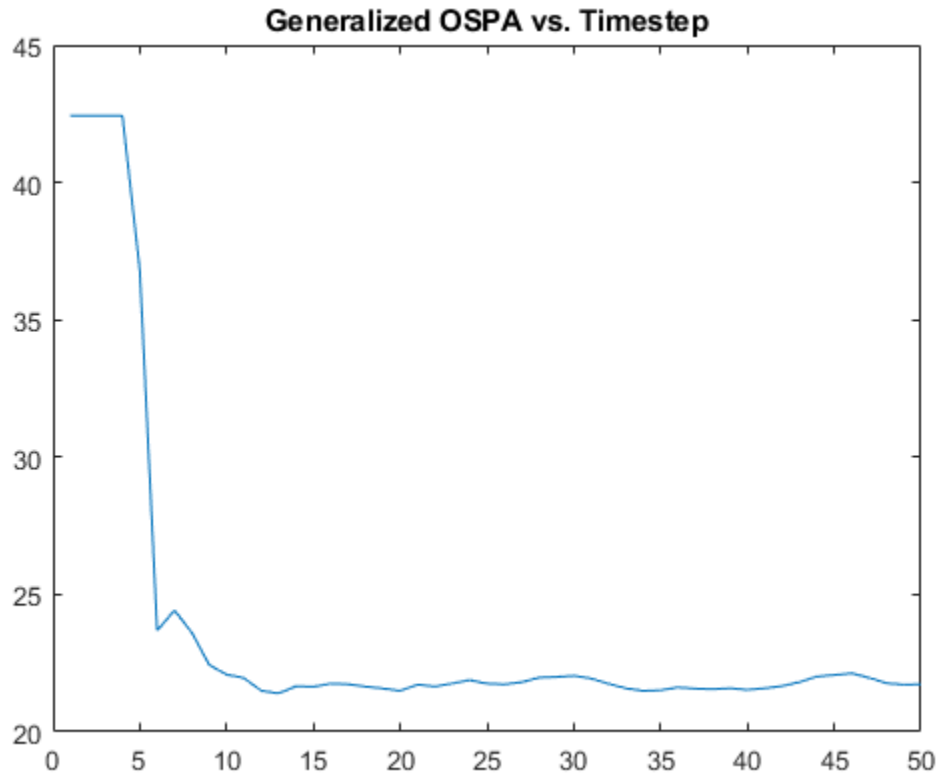
Note that making track confirmation more stringent requires more steps to confirm tracks. As a result, for the first few steps, there are no confirmed tracks and the GOSPA value remains high. You can compare that with the GOSPA graphs in the beginning of this example.

Try Another Tracker

You want to try using another tracker, in this case, the Joint Probabilistic Data Association tracker, `trackerJPDA`. Many of the properties defined for the previous tracker are still valid in the JPDA tracker. You use the `helperGNN2JPDA` to construct the JPDA tracker.

```
jpda = helperGNN2JPDA(tracker);
jpda.ClutterDensity = 1e-3;
rerunScenario(scenario, tracker, tgm, tp);
```

Generalized OSPA vs. Timestep
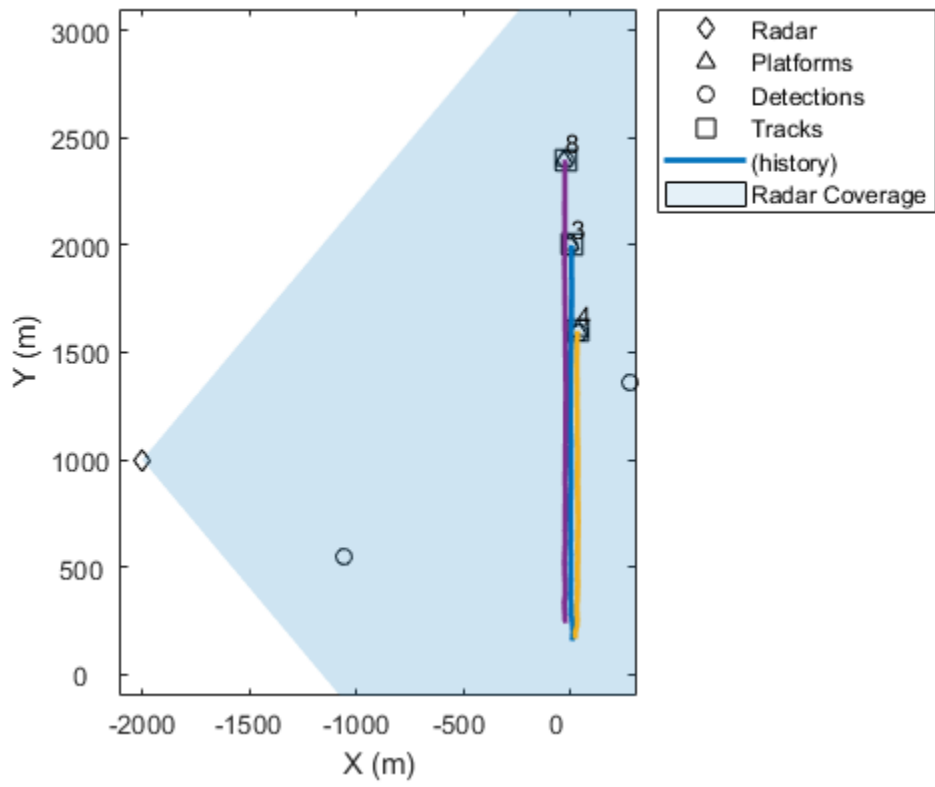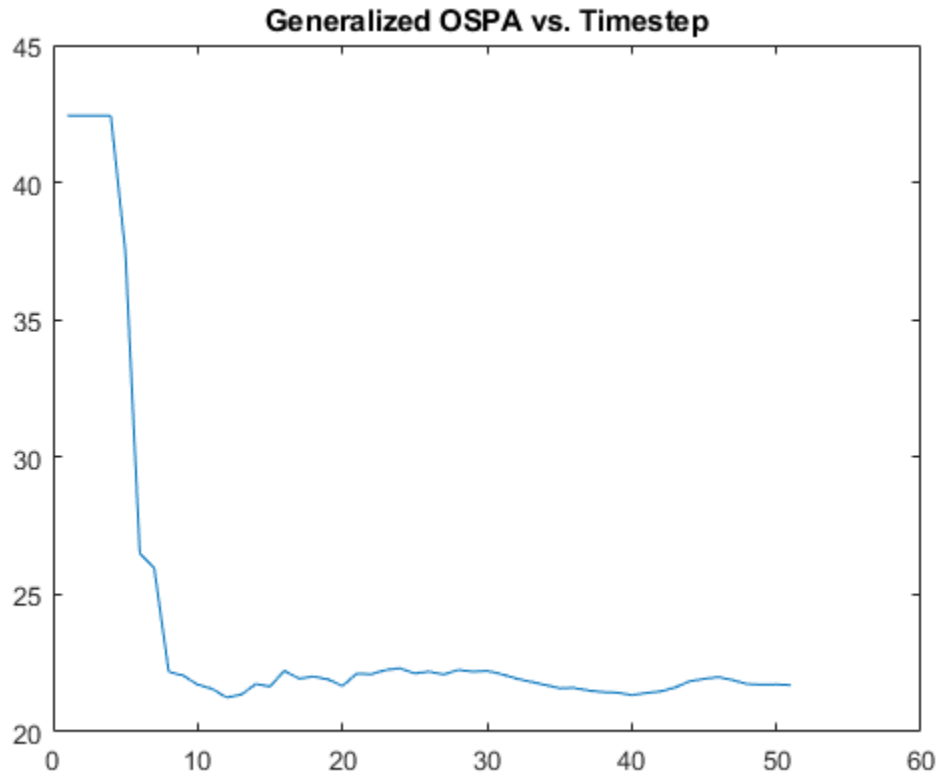
Summary

In this example, you learned to set up multi-object trackers in order to maintain tracks of the real objects in the scene, avoid false tracks, maintain the right number of tracks, and switch between different trackers.

When tuning a multi-object tracker, consider changing the following properties:

FilterInitializationFcn — defines the filter used to initialize a new track with an unassigned detection.

- The function defines the motion and measurement models.
- Check that the state uncertainties are correctly represented by the **StateCovariance** of the filter. For example, a large initial velocity covariance allows tracking fast moving objects.

AssignmentThreshold — maximum normalized distance used in the assignment of detections to tracks.

- Decrease this value to speed up assignment, especially in large scenes with many tracks and to avoid false tracks, redundant tracks, and track swaps. If using a `trackerJPDA`, reducing this value reduces the number of detections that are considered as assigned to a single track. If using a `trackerTOMHT`, reducing the assignment thresholds reduces the number of track branches that are created.
- Increase this value to avoid tracks diverging and breaking even in the presence of new detections.

MaxNumTracks — maximum number of tracks maintained by the tracker.

- Decrease this value to minimize memory usage and speed up tracker initialization.
- Increase this value to avoid new tracks not being initialized because the limit was reached.

ConfirmationThreshold — controls the confirmation of new tracks.

- Decrease this value to confirm more tracks and to confirm tracks more quickly.
- Increase this value to lower the number of false tracks.

DeletionThreshold — controls the coasting and deletion of tracks.

- Decrease this value to delete tracks more quickly (for example when the probability of detection is high).
- Increase this value to compensate for a lower probability of detection (for example coast tracks during occlusions).

You can refer to the following examples for additional details: "How to Generate C Code for a Tracker" on page 5-313, "Introduction to Track Logic" on page 5-76, "Tracking Closely Spaced Targets Under Ambiguity" on page 5-167, and "Track Point Targets in Dense Clutter Using GM-PHD Tracker" on page 5-507.

Utility Functions

updateDisplay

This function updates the display at every step of the simulation.

```
function updateDisplay(rp, pp, dp, trp, covp, scenario, dets, tracks)
    % Plot the platform positions
    poses = platformPoses(scenario);
    pos = reshape([poses(1:3).Position], 3, [])'; % Only the platforms
    plotPlatform(pp, pos);
    radarPos = poses(4).Position; % Only the radar
    plotPlatform(rp, radarPos)

    % Plot the detection positions
    if ~isempty(dets)
        ds = [dets{:}];
        dPos = reshape([ds.Measurement], 3, [])';
    else
        dPos = zeros(0,3);
    end
    plotDetection(dp, dPos);

    % Plot the tracks
    tPos = getTrackPositions(tracks, [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 1 0]);
    tIDs = string([tracks.TrackID]);
    plotTrack(trp, tPos, tIDs);

    % Plot the coverage
    covcon = coverageConfig(scenario);
    plotCoverage(covp, covcon);
end
```

rerunScenario

This function runs the scenario with a given tracker and a track GOSPA metric object. It uses the theater plot object to display the results.

```matlab
function info = rerunScenario(scenario, tracker, tgm, tp)
    % Reset the objects after the previous run
    restart(scenario);
    reset(tracker);
    reset(tgm);
    rp = findPlotter(tp, 'DisplayName', 'Radar');
    pp = findPlotter(tp, 'DisplayName', 'Platforms');
    trp = findPlotter(tp, 'DisplayName', 'Tracks');
    dp = findPlotter(tp, 'DisplayName', 'Detections');
    covp = findPlotter(tp, 'DisplayName', 'Radar Coverage');
    clearPlotterData(tp);

    gospa = zeros(1,51); % number of timesteps is 51
    i = 0;
    s = rng(2019);
    while advance(scenario)
        % Get detections
        dets = detect(scenario);

        % Update the tracker
        if isLocked(tracker) || ~isempty(dets)
            [tracks, ~, ~, info] = tracker(dets, scenario.SimulationTime);
        end

        % Evaluate GOSPA
        i = i + 1;
        truth = platformPoses(scenario);
        gospa(i) = tgm(tracks, truth);

        % Update the display
        updateDisplay(rp, pp, dp, trp, covp, scenario, dets, tracks);
    end
    rng(s)
    figure
    plot(gospa(gospa>0))
    title('Generalized OSPA vs. Timestep')
end
```

initFastCVEKF

This function modifies the default `initcvekf` filter initialization function to allow for more uncertainty in the object speed.

```matlab
function ekf = initFastCVEKF(detection)
    ekf = initcvekf(detection);
    initialCovariance = diag(ekf.StateCovariance);
    initialCovariance([2,4,6]) = 300^2; % Increase the speed covariance
    ekf.StateCovariance = diag(initialCovariance);
end
```

helperGNN2JPDA

This function provides the JPDA tracker equivalent to the GNN tracker given as an input.

```matlab
function jpda = helperGNN2JPDA(gnn)
    jpda = trackerJPDA(...
```

```matlab
        'MaxNumTracks', gnn.MaxNumTracks, ...
        'AssignmentThreshold', gnn.AssignmentThreshold, ...
        'FilterInitializationFcn', gnn.FilterInitializationFcn, ...
        'MaxNumSensors', gnn.MaxNumSensors, ...
        'ConfirmationThreshold', gnn.ConfirmationThreshold, ...
        'DeletionThreshold', gnn.DeletionThreshold, ...
        'HasCostMatrixInput', gnn.HasCostMatrixInput, ...
        'HasDetectableTrackIDsInput', gnn.HasDetectableTrackIDsInput);

    if strcmpi(gnn.TrackLogic, 'History')
        jpda.TrackLogic = 'History';
    else
        jpda.TrackLogic = 'Integrated';
        jpda.DetectionProbability = gnn.DetectionProbability;
        jpda.ClutterDensity = gnn.FalseAlarmRate / gnn.Volume;
    end
end
```

Generate Off-Centered IMU Readings

This example shows how to generate inertial measurement unit (IMU) readings from a sensor that is mounted on a ground vehicle. Depending on the location of the sensor, the IMU accelerations are different.

Create Trajectory

Specify the waypoint trajectory of a vehicle and compute the vehicle poses using `lookupPose`.

```
% Sampling rate.
Fs = 100;

% Waypoints and times of arrival.
waypoints = [1 1 1; 3 1 1; 3 0 0; 0 0 0];
t = [1; 10; 20; 30];

% Create trajectory and compute pose.
traj = waypointTrajectory(waypoints, t, "SampleRate", Fs);
[posVeh, orientVeh, velVeh, accVeh, angvelVeh] = lookupPose(traj, ...
    t(1):1/Fs:t(end));
```

Create Sensor and Define Offset

Create two 9-axis `imuSensor` objects composed of accelerometer, gyroscope, and magnetometer sensors. One `imuSensor` object generates readings of an IMU mounted at the vehicle's origin and the other one generates readings of an IMU mounted at the driver's seat. Next, specify the offset between the vehicle origin and the IMU mounted at the driver's seat. Call `helperPlotIMU` to visualize the locations of the sensors.

```
% IMU at vehicle origin.
imu = imuSensor("accel-gyro-mag", "SampleRate", Fs);

% IMU at driver's seat.
mountedIMU = imuSensor("accel-gyro-mag", "SampleRate", Fs);

% Position and orientation offset of the vehicle and the mounted IMU.
posVeh2IMU = [2.4 0.5 0.4];
orientVeh2IMU = quaternion([0 0 90], "eulerd", "ZYX", "frame");

% Visualization.
helperPlotIMU(posVeh(1,:), orientVeh(1,:), posVeh2IMU, orientVeh2IMU);
```
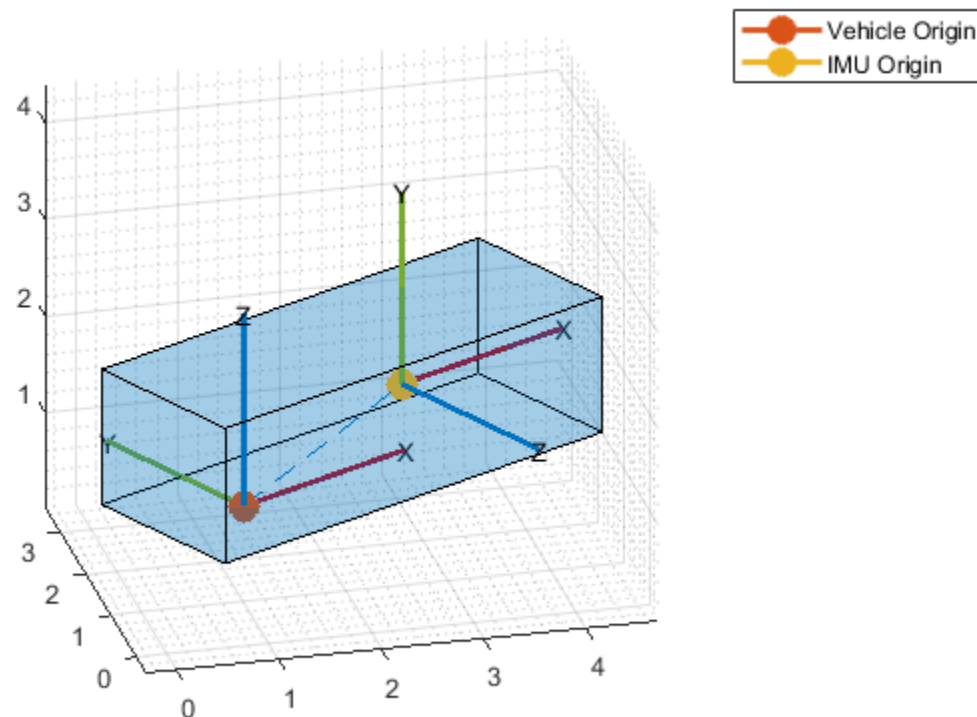
Calculate IMU Trajectory Using Vehicle Trajectory

Compute the ground truth trajectory of the IMU mounted at the driver's seat using the `transformMotion` function. This function uses the position and orientation offsets and the vehicle trajectory to compute the IMU trajectory.

```
[posIMU, orientIMU, velIMU, accIMU, angvelIMU] = transformMotion( ...
    posVeh2IMU, orientVeh2IMU, ...
    posVeh, orientVeh, velVeh, accVeh, angvelVeh);
```

Generate Sensor Readings

Generate the IMU readings for both the IMU mounted at the vehicle origin and the IMU mounted at the driver's seat.

```
% IMU at vehicle origin.
[accel, gyro, mag] = imu(accVeh, angvelVeh, orientVeh);

% IMU at driver's seat.
[accelMounted, gyroMounted, magMounted] = mountedIMU( ...
    accIMU, angvelIMU, orientIMU);
```

Compare Accelerometer Readings

Compare the accelerometer readings of the two IMUs. Notice that the *x*-axis acceleration is different because of the off-center location.

```
figure('Name', 'Accelerometer Comparison')
subplot(3, 1, 1)
plot([accel(:,1), accelMounted(:,1)])
legend('Aligned with Vehicle', 'Off-centered')
title('Accelerometer')
ylabel('x-axis (m/s^2)')
subplot(3, 1, 2)
plot([accel(:,2), accelMounted(:,2)])
ylabel('y-axis (m/s^2)')
subplot(3, 1, 3)
plot([accel(:,3), accelMounted(:,3)])
ylabel('z-axis (m/s^2)')
```
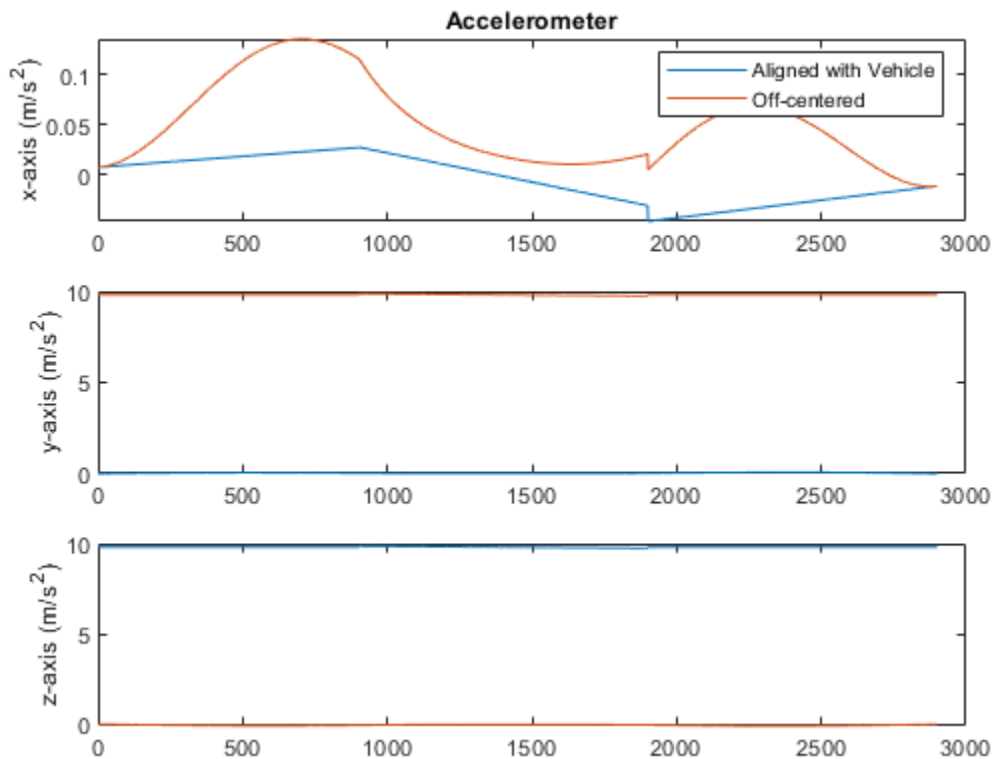

Compare Gyroscope Readings

Compare the gyroscope readings of the two IMUs.

```
figure('Name', 'Gyroscope Comparison')
subplot(3, 1, 1)
plot([gyro(:,1), gyroMounted(:,1)])
ylim([-0.22 0.1])
legend('Aligned with Vehicle', 'Off-centered')
title('Gyroscope')
ylabel('x-axis (rad/s)')
subplot(3, 1, 2)
plot([gyro(:,2), gyroMounted(:,2)])
ylabel('y-axis (rad/s)')
subplot(3, 1, 3)
```

```
plot([gyro(:,3), gyroMounted(:,3)])
ylabel('z-axis (rad/s)')
```
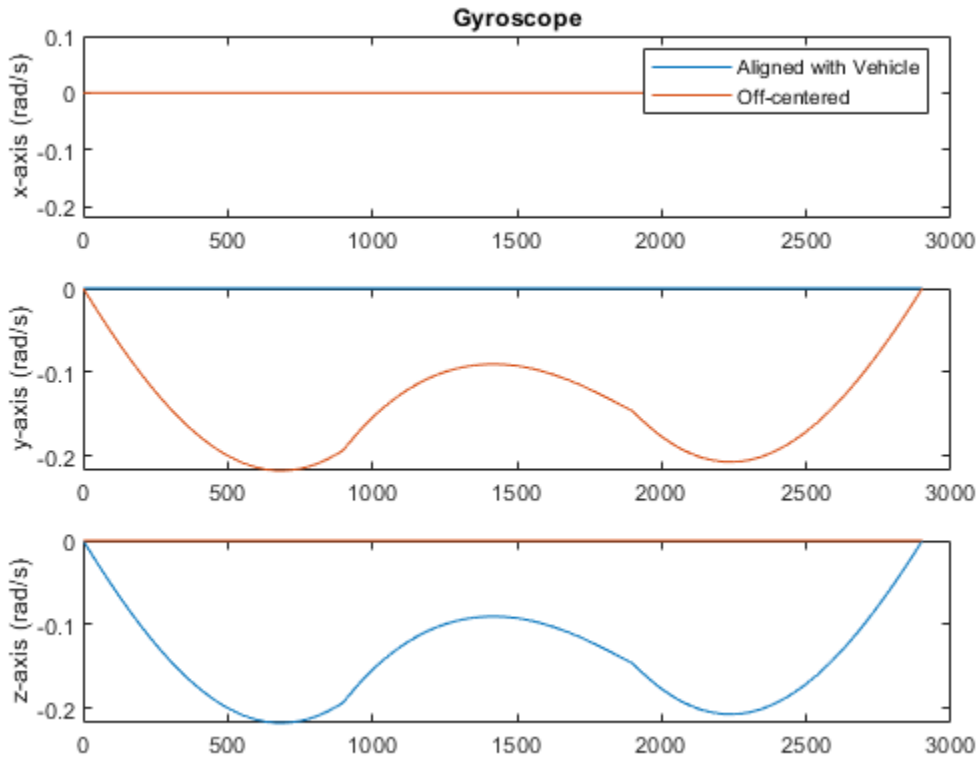

Compare Magnetometer Readings

Compare the magnetometer readings of the two IMUs.

```
figure('Name', 'Magnetometer Comparison')
subplot(3, 1, 1)
plot([mag(:,1), magMounted(:,1)])
legend('Aligned with Vehicle', 'Off-centered')
title('Magnetometer')
ylabel('x-axis (\muT)')
subplot(3, 1, 2)
plot([mag(:,2), magMounted(:,2)])
ylabel('y-axis (\muT)')
subplot(3, 1, 3)
plot([mag(:,3), magMounted(:,3)])
ylabel('z-axis (\muT)')
```
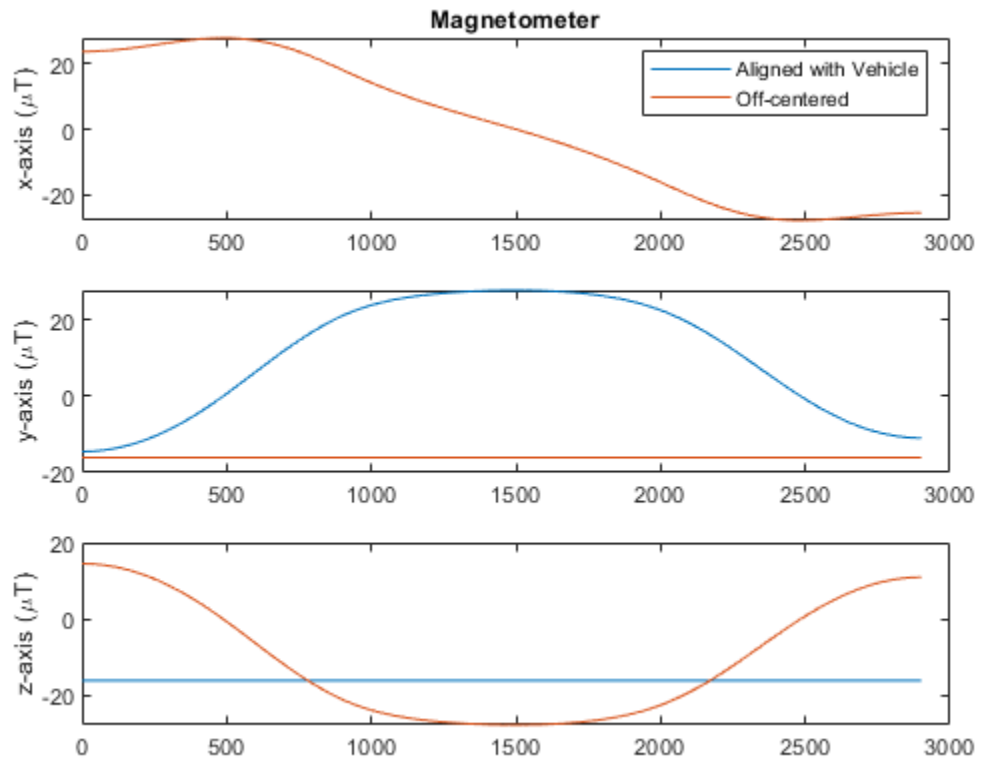
Detect Noise in Sensor Readings with Residual Filtering

This example shows how to use the `residualgps` object function and residual filtering to detect when new sensor measurements may not be consistent with the current filter state.

Load Trajectory and Sensor Data

Load the MAT-file `loggedDataWithMultipath.mat`. This file contains simulated IMU and GPS data as well as the ground truth position and orientation of a circular trajectory. The GPS data contains errors due to multipath errors in one section of the trajectory. These errors were modelled by adding white noise to the GPS data to simulate the effects of an urban canyon.

```
load('loggedDataWithMultipath.mat', ...
    'imuFs', 'accel', 'gyro', ... % IMU readings
    'gpsFs', 'lla', 'gpsVel', ... % GPS readings
    'truePosition', 'trueOrientation', ... % Ground truth pose
    'localOrigin', 'initialState', 'multipathAngles')

% Number of IMU samples per GPS sample.
imuSamplesPerGPS = (imuFs/gpsFs);

% First and last indices corresponding to multipath errors.
multipathIndices = [1850 2020];
```

Fusion Filter

Create two pose estimation filters using the `insfilterNonholonomic` object. Use one filter to process all the sensor readings. Use the other filter to only process the sensor readings that are not considered outliers.

```
% Create filters.

% Use this filter to only process sensor readings that are not detected as
% outliers.
gndFusionWithDetection = insfilterNonholonomic('ReferenceFrame', 'ENU', ...
    'IMUSampleRate', imuFs, ...
    'ReferenceLocation', localOrigin, ...
    'DecimationFactor', 2);
% Use this filter to process all sensor readings, regardless of whether or
% not they are outliers.
gndFusionNoDetection = insfilterNonholonomic('ReferenceFrame', 'ENU', ...
    'IMUSampleRate', imuFs, ...
    'ReferenceLocation', localOrigin, ...
    'DecimationFactor', 2);

% GPS measurement noises.
Rvel = 0.01;
Rpos = 1;

% The dynamic model of the ground vehicle for this filter assumes there is
% no side slip or skid during movement. This means that the velocity is
% constrained to only the forward body axis. The other two velocity axis
% readings are corrected with a zero measurement weighted by the
% |ZeroVelocityConstraintNoise| parameter.
gndFusionWithDetection.ZeroVelocityConstraintNoise = 1e-2;
gndFusionNoDetection.ZeroVelocityConstraintNoise = 1e-2;
```

```
% Process noises.
gndFusionWithDetection.GyroscopeNoise = 4e-6;
gndFusionWithDetection.GyroscopeBiasNoise = 4e-14;
gndFusionWithDetection.AccelerometerNoise = 4.8e-2;
gndFusionWithDetection.AccelerometerBiasNoise = 4e-14;
gndFusionNoDetection.GyroscopeNoise = 4e-6;
gndFusionNoDetection.GyroscopeBiasNoise = 4e-14;
gndFusionNoDetection.AccelerometerNoise = 4.8e-2;
gndFusionNoDetection.AccelerometerBiasNoise = 4e-14;

% Initial filter states.
gndFusionWithDetection.State = initialState;
gndFusionNoDetection.State = initialState;

% Initial error covariance.
gndFusionWithDetection.StateCovariance = 1e-9*ones(16);
gndFusionNoDetection.StateCovariance = 1e-9*ones(16);
```

Initialize Scopes

The `HelperPoseViewer` scope allows a 3-D visualization comparing the filter estimate and ground truth. Using multiple scopes can slow the simulation. To disable the scopes, set the corresponding logical variable to `false`.

```
usePoseView = true;  % Turn on the 3D pose viewer

if usePoseView
    [viewerWithDetection, viewerNoDetection, annoHandle] ...
        = helperCreatePoseViewers(initialState, multipathAngles);
end
```

Simulation Loop

The main simulation loop is a `for` loop with a nested `for` loop. The first loop executes at the `gpsFs`, which is the GPS measurement rate. The nested loop executes at the `imuFs`, which is the IMU sample rate. Each scope is updated at the IMU sample rate.

```
numsamples = numel(trueOrientation);
numGPSSamples = numsamples/imuSamplesPerGPS;

% Log data for final metric computation.
estPositionNoCheck = zeros(numsamples, 3);
estOrientationNoCheck = quaternion.zeros(numsamples, 1);

estPosition = zeros(numsamples, 3);
estOrientation = quaternion.zeros(numsamples, 1);

% Threshold for outlier residuals.
residualThreshold = 6;

idx = 0;
for sampleIdx = 1:numGPSSamples
    % Predict loop at IMU update frequency.
    for i = 1:imuSamplesPerGPS
        idx = idx + 1;

        % Use the predict method to estimate the filter state based
        % on the accelData and gyroData arrays.
```

```matlab
        predict(gndFusionWithDetection, accel(idx,:), gyro(idx,:));

        predict(gndFusionNoDetection, accel(idx,:), gyro(idx,:));

        % Log the estimated orientation and position.
        [estPositionNoCheck(idx,:), estOrientationNoCheck(idx,:)] ...
            = pose(gndFusionWithDetection);

        [estPosition(idx,:), estOrientation(idx,:)] ...
            = pose(gndFusionNoDetection);

        % Update the pose viewer.
        if usePoseView
            viewerWithDetection(estPositionNoCheck(idx,:), ...
                estOrientationNoCheck(idx,:), ...
                truePosition(idx,:), trueOrientation(idx,:));

            viewerNoDetection(estPosition(idx,:), ...
                estOrientation(idx,:), truePosition(idx,:), ...
                trueOrientation(idx,:));
        end
    end

    % This next section of code runs at the GPS sample rate.

    % Update the filter states based on the GPS data.
    fusegps(gndFusionWithDetection, lla(sampleIdx,:), Rpos, ...
        gpsVel(sampleIdx,:), Rvel);

    % Check the normalized residual of the current GPS reading. If the
    % value is too large, it is considered an outlier and disregarded.
    [res, resCov] = residualgps(gndFusionNoDetection, lla(sampleIdx,:), ...
        Rpos, gpsVel(sampleIdx,:), Rvel);
    normalizedRes = res(1:3) ./ sqrt( diag(resCov(1:3,1:3)).' );
    if (all(abs(normalizedRes) <= residualThreshold))
        % Update the filter states based on the GPS data.
        fusegps(gndFusionNoDetection, lla(sampleIdx,:), Rpos, ...
            gpsVel(sampleIdx,:), Rvel);
        if usePoseView
            set(annoHandle, 'String', 'Outlier status: none', ...
                'EdgeColor', 'k');
        end
    else
        if usePoseView
            set(annoHandle, 'String', 'Outlier status: detected', ...
                'EdgeColor', 'r');
        end
    end
end
```
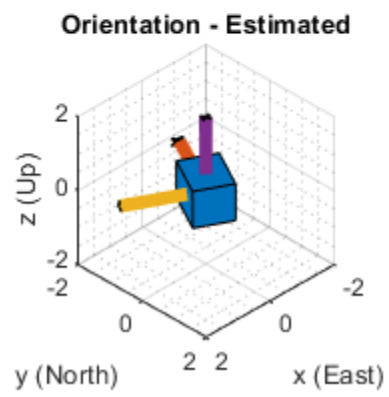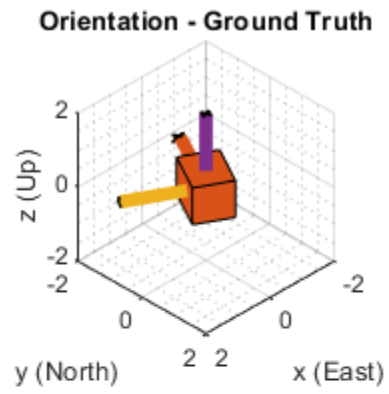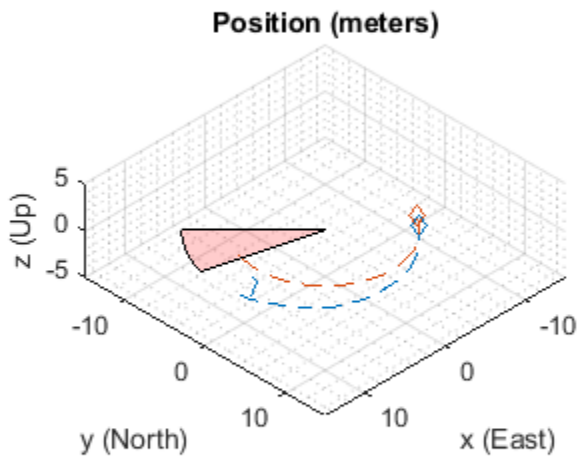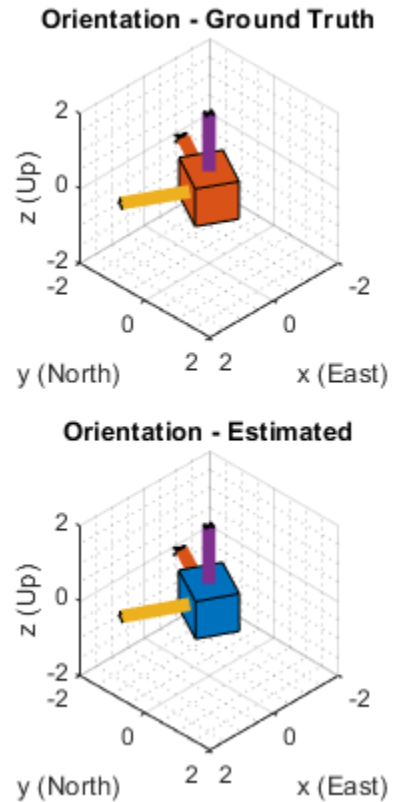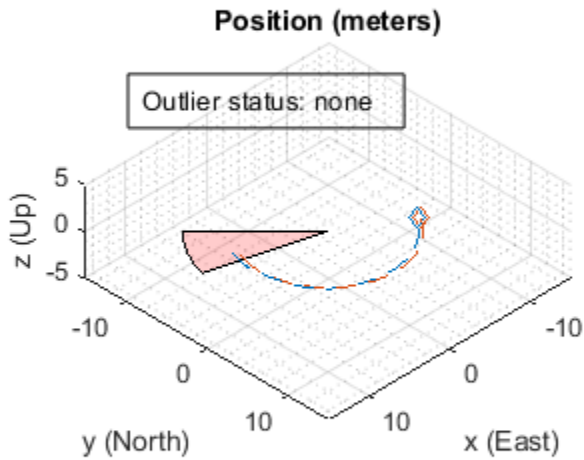
Error Metric Computation

Calculate the position error for both filter estimates. There is an increase in the position error in the filter that does not check for any outliers in the GPS measurements.

```
% Calculate position errors.
posdNoCheck = estPositionNoCheck - truePosition;
posd = estPosition - truePosition;

% Plot results.
t = (0:size(posd,1)-1).' ./ imuFs;
figure('Units', 'normalized', 'Position', [0.2615 0.2833 0.4552 0.3700])
subplot(1, 2, 1)
plot(t, posdNoCheck)
ax = gca;
yLims = get(ax, 'YLim');
hold on
mi = multipathIndices;
fill([t(mi(1)), t(mi(1)), t(mi(2)), t(mi(2))], [7 -5 -5 7], ...
    [1 0 0],'FaceAlpha', 0.2);
set(ax, 'YLim', yLims);
title('Position Error (No outlier removal)')
xlabel('time (s)')
ylabel('error (m)')
legend('x', 'y', 'z', sprintf('outlier\nregion'))

subplot(1, 2, 2)
```

```
plot(t, posd)
ax = gca;
yLims = get(ax, 'YLim');
hold on
mi = multipathIndices;
fill([t(mi(1)), t(mi(1)), t(mi(2)), t(mi(2))], [7 -5 -5 7], ...
    [1 0 0],'FaceAlpha', 0.2);
set(ax, 'YLim', yLims);
title('Position Error (Outlier removal)')
xlabel('time (s)')
ylabel('error (m)')
legend('x', 'y', 'z', sprintf('outlier\nregion'))
```
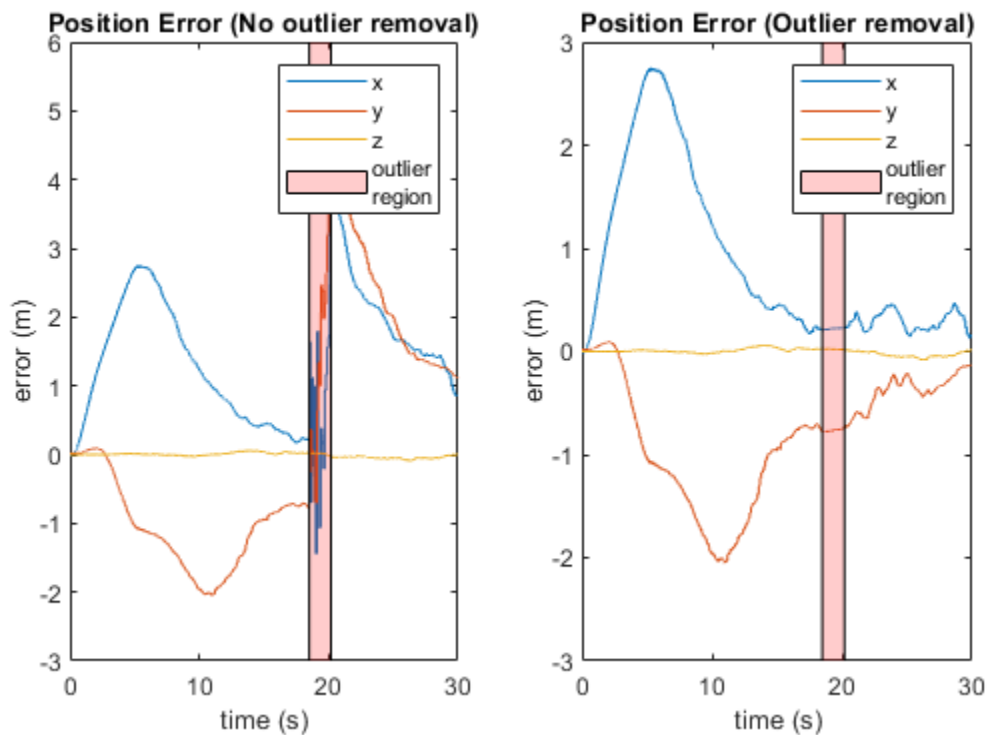

Conclusion

The `residualgps` object function can be used to detect potential outliers in sensor measurements before using them to update the filter states of the `insfilterNonholonomic` object. The other pose estimation filter objects such as, `insfilterMARG`, `insfilterAsync`, and `insfilterErrorState` also have similar object functions to calculate sensor measurement residuals.

IMU Sensor Fusion with Simulink

This example shows how to generate and fuse IMU sensor data using Simulink®. You can accurately model the behavior of an accelerometer, a gyroscope, and a magnetometer and fuse their outputs to compute orientation.

Inertial Measurement Unit

An inertial measurement unit (IMU) is a group of sensors consisting of an accelerometer measuring acceleration and a gyroscope measuring angular velocity. Frequently, a magnetometer is also included to measure the Earth's magnetic field. Each of these three sensors produces a 3-axis measurement, and these three measurements constitute a 9-axis measurement.

Attitude Heading and Reference System

An Attitude Heading and Reference System (AHRS) takes the 9-axis sensor readings and computes the orientation of the device. This orientation is given relative to the NED frame, where N is the Magnetic North direction. The AHRS block in Simulink accomplishes this using an indirect Kalman filter structure.

Simulink System

Open the Simulink model that fuses IMU sensor data

```
open_system('IMUFusionSimulinkModel');
```
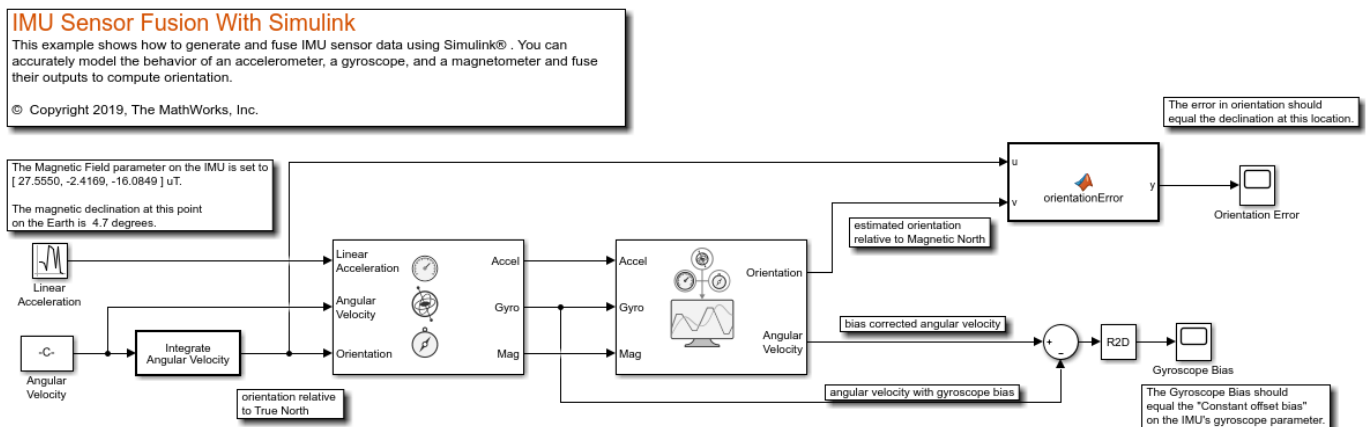

Inputs and Configuration

The inputs to the IMU block are the device's linear acceleration, angular velocity, and the orientation relative to the navigation frame. The orientation is of the form of a quaternion (a 4-by-1 vector in Simulink) or rotation matrix (a 3-by-3 matrix in Simulink) that rotates quantities in the navigation frame to the body frame. In this model, the angular velocity is simply integrated to create an orientation input. The angular velocity is in rad/s and the linear acceleration is in m/s^2. Because the AHRS has only one input related to translation (the accelerometer input), it cannot distinguish between gravity and linear acceleration. Therefore, the AHRS algorithm assumes that linear acceleration is a slowly varying white noise process. This is a common assumption for 9-axis fusion algorithms.

True North vs Magnetic North

Magnetic field parameter on the IMU block dialog can be set to the local magnetic field value. Magnetic field values can be found on the NOAA website or using the `wrldmagm` function in the Aerospace Toolbox™. The magnetic field values on the IMU block dialog correspond the readings of a perfect magnetometer that is orientated to True North. Therefore, the orientation input to the IMU block is relative to the NED frame, where N is the True North direction. However, the AHRS filter navigates towards Magnetic North, which is typical for this type of filter. Therefore, the orientation input to the IMU and the estimated orientation at the output of the AHRS differ by the declination angle between True North and Magnetic North.

This simulation is setup for $0°$ latitude and $0°$ longitude. The magnetic field at this location is set as [27.5550, -2.4169, -16.0849] microtesla in the IMU block. The declination at this location is about $4.7°$

Simulation

Simulate the model. The IMU input orientation and the estimated output orientation of the AHRS are compared using quaternion distance. This is preferable compared to differencing the Euler angle equivalents, considering the Euler angle singularities.

```
sim('IMUFusionSimulinkModel');
```

Estimated Orientation

The difference in estimated vs true orientation should be nearly $4.7°$, which is the declination at this latitude and longitude.

Gyroscope Bias

The second output of the AHRS filter is the bias-corrected gyroscope reading. In the IMU block, the gyroscope was given a bias of 0.0545 rad/s or 3.125 deg/s, which should match the steady state value in the `Gyroscope Bias` scope block.

Further Exercises

By varying the parameters on the IMU, you should see a corresponding change in orientation on the output of the AHRS. You can set the parameters on the IMU block to match a real IMU datasheet and tune the AHRS parameters to meet your requirements.

Track Space Debris Using a Keplerian Motion Model

This example shows how to model earth-centric trajectories using custom motion models within `trackingScenario`, how to configure a monostatic radar sensor to generate synthetic detections of space debris, and how to setup a multi-object tracker to track the simulated targets.

Space debris scenario

There are more than 30,000 large debris objects (with diameter larger than 10cm) and more than 1 million smaller debris objects in Low Earth Orbit (LEO) [1]. This debris can be dangerous for human activities in space, damage operational satellites, and force time sensitive and costly avoidance maneuvers. As space activity increases, reducing and monitoring the space debris becomes crucial.

You can use Sensor Fusion and Tracking Toolbox™ to model the debris trajectories, generate synthetic radar detections of this debris, and obtain position and velocity estimates of each object.

First, create a tracking scenario and set the random seed for repeatable results.

```
s = rng;
rng(2020);
scene = trackingScenario('IsEarthCentered',true);
```

You use the Earth-Centered-Earth-Fixed (ECEF) reference frame. The origin of this frame is at the center of the Earth and the Z axis points toward the north pole. The X axis points towards the intersection of the equator and the Greenwich meridian. The Y axis completes the right-handed system. Platform positions and velocities are defined using Cartesian coordinates in this frame.

Define debris motion model

The `helperMotionTrajectory` class used in this example defines debris object trajectories using a custom motion model function.

Trajectories of space objects rotating around the Earth can be approximated with a Keplerian model, which assumes that Earth is a point-mass body and the objects orbiting around the earth have negligible masses. Higher order effects in Earth gravitational field and environmental disturbances are not accounted for. Since the equation of motion is expressed in ECEF frame which is a non-inertial reference frame, the Coriolis and centripetal forces are accounted for.

The ECEF debris object acceleration vector is

$$\vec{a} = \frac{-\mu}{r^3}\vec{r} - 2\vec{\Omega} \times \frac{\mathrm{d}}{\mathrm{d}t}\vec{r} - \vec{\Omega} \times \left(\vec{\Omega} \times \vec{r}\right),$$

where μ is the standard gravitational parameter of the Earth, $\vec{r}$ is the ECEF debris object position vector, r is the norm of the position vector, and $\vec{\Omega}$ is the Earth rotation vector.

The function `keplerorbit` provided below uses a 4th order Runge-Kutta numerical integration of this equation to propagate the position and velocity in time.

First, we create initial positions and velocities for the space debris objects. This is done by obtaining the traditional orbital elements (semi-major axis, eccentricity, inclination, longitude of the ascending node, argument of periapsis, and true anomaly angles) of these objects from random distributions. Then convert these orbital elements to position and velocity vectors by using the supporting function `oe2rv`.

```
% Generate a population of debris
numDebris = 100;

range = 7e6 + 1e5*randn(numDebris,1);
ecc = 0.015 + 0.005*randn(numDebris,1);
inc = 80 + 10*rand(numDebris,1);
lan = 360*rand(numDebris,1);
w = 360*rand(numDebris,1);
nu = 360*rand(numDebris,1);

% Convert to initial position and velocity
for i = 1:numDebris
    [r,v] = oe2rv(range(i),ecc(i),inc(i),lan(i),w(i),nu(i));
    data(i).InitialPosition = r; %#ok<SAGROW>
    data(i).InitialVelocity = v; %#ok<SAGROW>
end

% Create platforms and assign them trajectories using the keplerorbit motion model
for i=1:numDebris
    debris(i) = platform(scene); %#ok<SAGROW>
    debris(i).Trajectory = helperMotionTrajectory(@keplerorbit,...
        'SampleRate',0.1,... % integration step 10sec
        'Position',data(i).InitialPosition,...
        'Velocity',data(i).InitialVelocity); %#ok<SAGROW>
end
```

Model space surveillance radars

In this example, we define four antipodal stations with fan-shaped radar beams looking into space. The fans cut through the orbits of debris objects to maximize the number of object detections. A pair of stations are located in the Pacific ocean and in the Atlantic ocean, whereas a second pair of surveillance stations are located near the poles. Having four dispersed radars allows for the re-detection of space debris to correct their position estimates and also acquiring new debris detections.

```
% Create a space surveillance station in the Pacific ocean
station1 = platform(scene,'Position',[10 180 0]);

% Create a second surveillance station in the Atlantic ocean
station2 = platform(scene,'Position',[0 -20 0]);

% Near the North Pole, create a third surveillance station in Iceland
station3 = platform(scene,'Position',[65 -20 0]);

% Create a fourth surveillance station near the south pole
station4 = platform(scene,'Position',[-90 0 0]);
```

Each station is equipped with a radar modeled with a `monostaticRadarSensor` object. In order to detect debris objects in the LEO range, the radar has the following requirements:

- Detecting a 10 dBsm object up to 2000 km away
- Resolving objects horizontally and vertically with a precision of 100 m at 2000 km range
- Having a fan-shaped field of view of 120 degrees in azimuth and 30 degrees in elevation
- Looking up into space based on its geo-location

```
% Create fan-shaped monostatic radars to monitor space debris objects
```

```matlab
radar1 = monostaticRadarSensor(1,...
    'UpdateRate',0.1,... 10 sec
    'ScanMode','No scanning',...
    'MountingAngles',[0 90 0],... look up
    'FieldOfView',[120;30],... degrees
    'ReferenceRange',2000000,... m
    'ReferenceRCS', 10,... dBsm
    'HasFalseAlarms',false,...
    'HasElevation',true,...
    'AzimuthResolution',0.01,... degrees
    'ElevationResolution',0.01,... degrees
    'RangeResolution',100,... m
    'HasINS',true,...
    'DetectionCoordinates','Scenario');
station1.Sensors = radar1;

radar2 = clone(radar1);
radar2.SensorIndex = 2;
station2.Sensors = radar2;

radar3 = clone(radar1);
radar3.SensorIndex = 3;
station3.Sensors = radar3;

radar4 = clone(radar1);
radar4.SensorIndex = 4;
station4.Sensors = radar4;
```

Visualize the ground truth on a virtual globe

In this example, `helperScenarioGlobeViewer` provides a virtual globe used to visualize all the elements defined in the tracking scenario: individual debris objects and their trajectories, radar fans, radar detections, and tracks.

```matlab
globeDisplay = helperScenarioGlobeViewer;

% Show radar beams on the globe
covcon = coverageConfig(scene);
plotCoverage(globeDisplay,covcon);

% Set TargetHistoryLength to visualize the full trajectory of the debris objects
globeDisplay.TargetHistoryLength = 1000;

scene.StopTime = 3600;
scene.UpdateRate = 0.1;

while advance(scene)
    time = scene.SimulationTime;
    updateDisplay(globeDisplay,time,debris);
end
snap(globeDisplay);
```
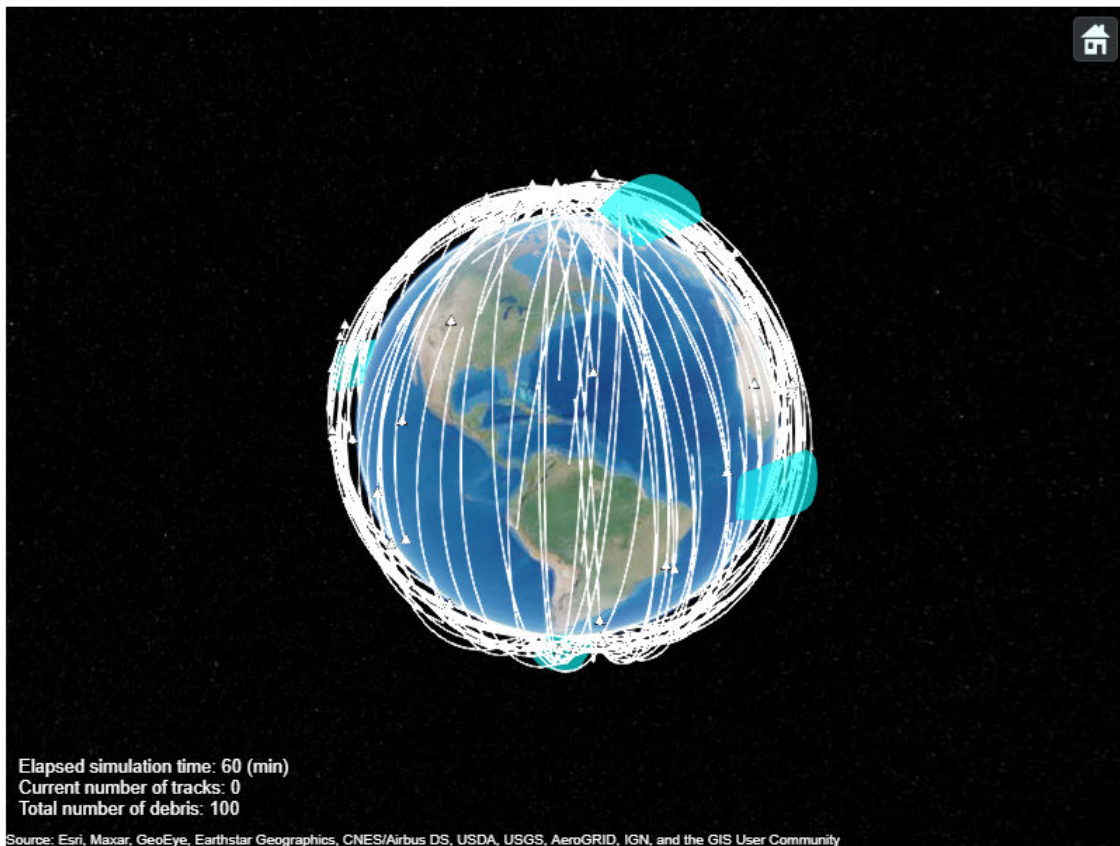
On the virtual globe, you can see the space debris represented by white dots with individual trailing trajectories shown by white lines. Most of the generated debris objects are on orbits with high inclination angles close to 80 deg.

The trajectories are plotted in ECEF coordinates, and therefore the entire trajectory rotates towards the west due to Earth rotation. After several orbit periods, all space debris pass through the surveillance beams of the radars.

Simulate synthetic detections and track space debris

The sensor models use the ground truth to generate synthetic detections. Call the `detect` method on the tracking scenario to obtain all the detections in the scene.

A multi-object tracker `trackerJPDA` is used to create new tracks, associate detections to existing tracks, estimate their state, and delete divergent tracks. Setting the property `HasDetectableTrackIDsInput` to true allows the tracker to accept an input that indicates whether a tracked object is detectable in the surveillance region. This is important for not penalizing tracks that are propagated outside of the radar surveillance areas. The utility function `isDetectable` calculates which tracks are detectable at each simulation step.

Additionally, a utility function `deleteBadTracks` is used to delete divergent tracks faster.

```matlab
% Define Tracker
tracker = trackerJPDA('FilterInitializationFcn',@initKeplerUKF,...
    'HasDetectableTrackIDsInput',true,...
    'ClutterDensity',1e-20,...
    'AssignmentThreshold',1e4,...
    'DeletionThreshold',[7 10]);

% Reset scenario, seed, and globe display
restart(scene);
scene.StopTime = 1800; % 30 min
clear(globeDisplay);
globeDisplay.TargetHistoryLength = 2;
plotCoverage(globeDisplay,covcon);

% Initialize tracks

confTracks = objectTrack.empty(0,1);
while advance(scene)
    time = scene.SimulationTime;

    % Generate detections
    detections = detect(scene);

    % Generate and update tracks
    detectableInput = isDetectable(tracker,time, covcon);
    if ~(isempty(detections) && ~isLocked(tracker))
        [confTracks, ~, allTracks,info] = tracker(detections,time,detectableInput);
        confTracks = deleteBadTracks(tracker,confTracks);
    end

    % Update globe display
    updateDisplay(globeDisplay,time,debris,detections,[],confTracks);

    % Move camera during simulation and take snapshots
    switch time
        case 100
            setCamera(globeDisplay,[90 150 5e6],[0 -65 345]);
            im1 = snap(globeDisplay);
        case 270
            setCamera(globeDisplay,[60 -120 2.6e6],[20 -45 20]);
        case 380
            setCamera(globeDisplay,[60 -120 2.6e6],[20 -45 20]);
            im2 = snap(globeDisplay);
        case 400
            % reset
            setCamera(globeDisplay,[17.3 -67.2 2.400e7], [360 -90 0]);
        case 1500
            setCamera(globeDisplay,[54 2.3 6.09e6], [0 -73 348]);
        case 1560
            im3 = snap(globeDisplay);
    end

end

% Restore random seed
rng(s);

imshow(im1);
```
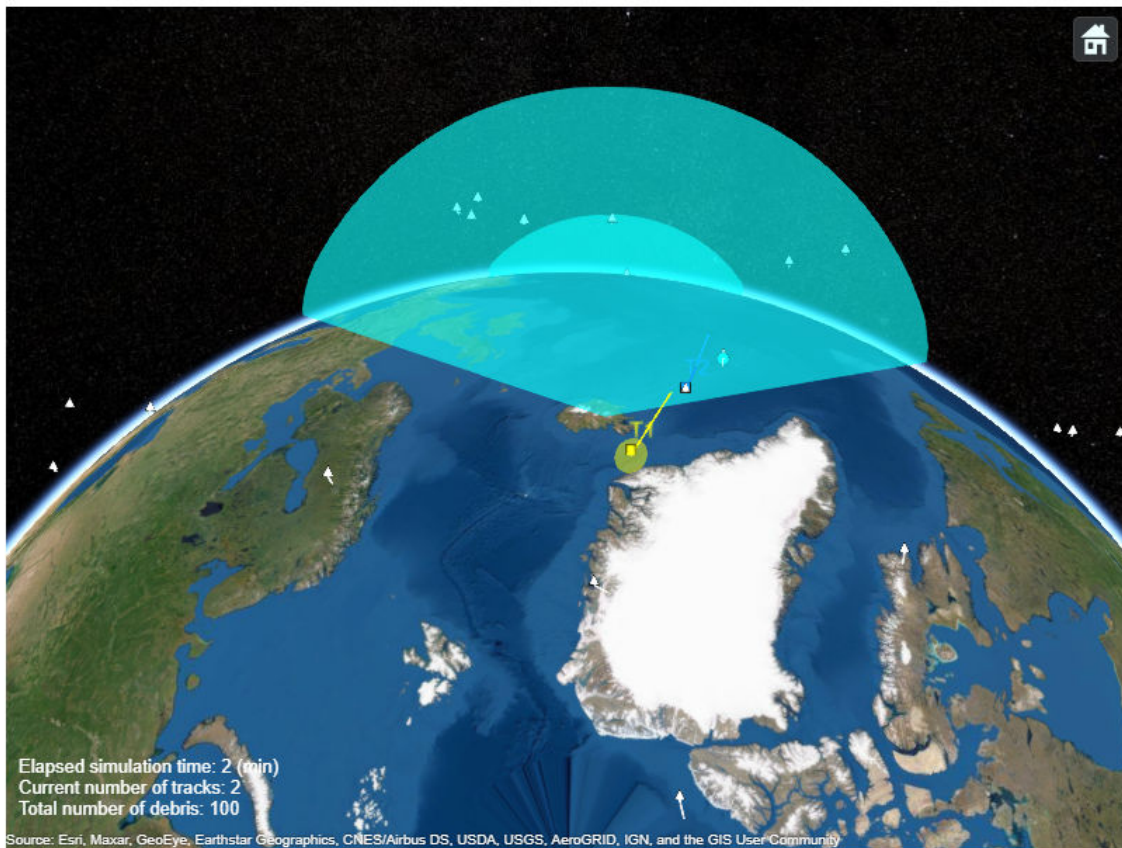
On the first snapshot, you can see an object being tracked as track T1 in yellow. This object was only detected twice, which was not enough to reduce the uncertainty of the track. Therefore, the size of its covariance ellipse is relatively large. You can also observe another track T2 in blue, which is detected by the sensor several times. As a result, its corresponding covariance ellipse is much smaller since more detections were used to correct the state estimate.

```
imshow(im2);
```
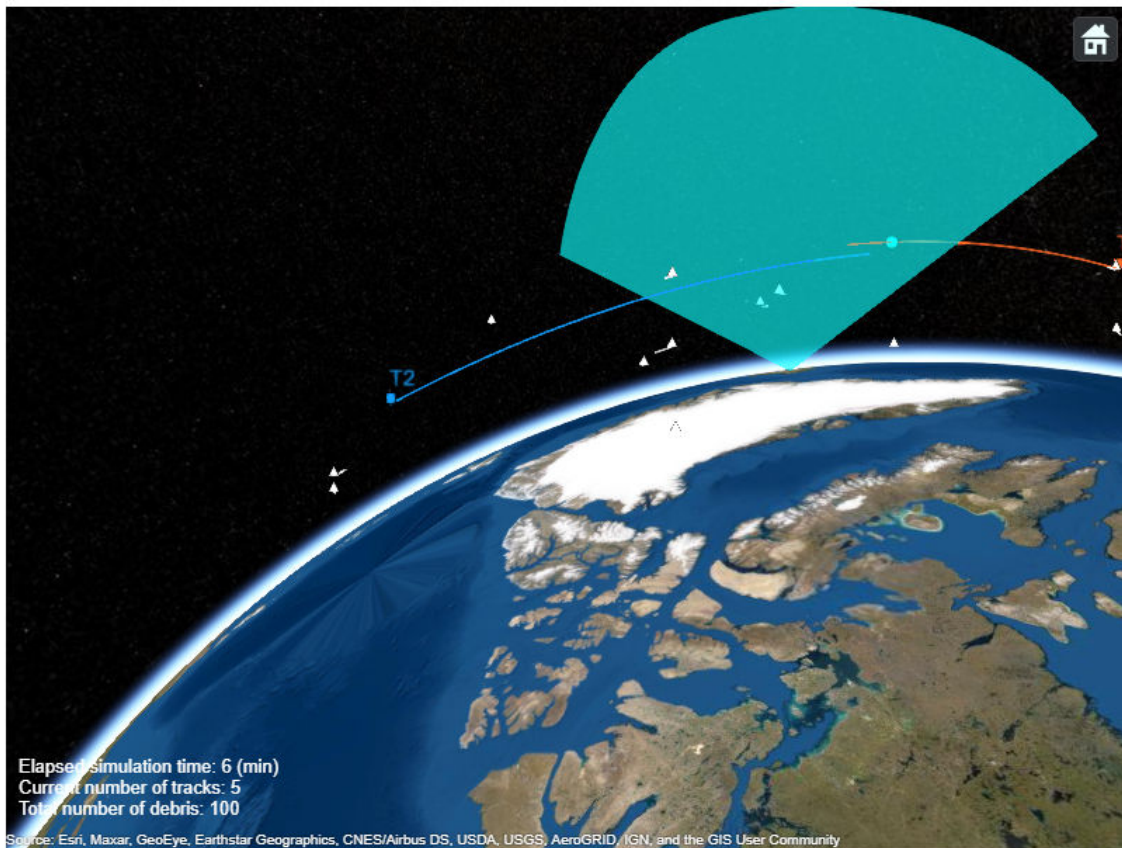
A few minutes later, as seen on the snapshot above, T1 was deleted because the uncertainty of the track has grown too large without detections. On the other hand, the second track T2 survived due to the additional detections.

```
imshow(im3)
```
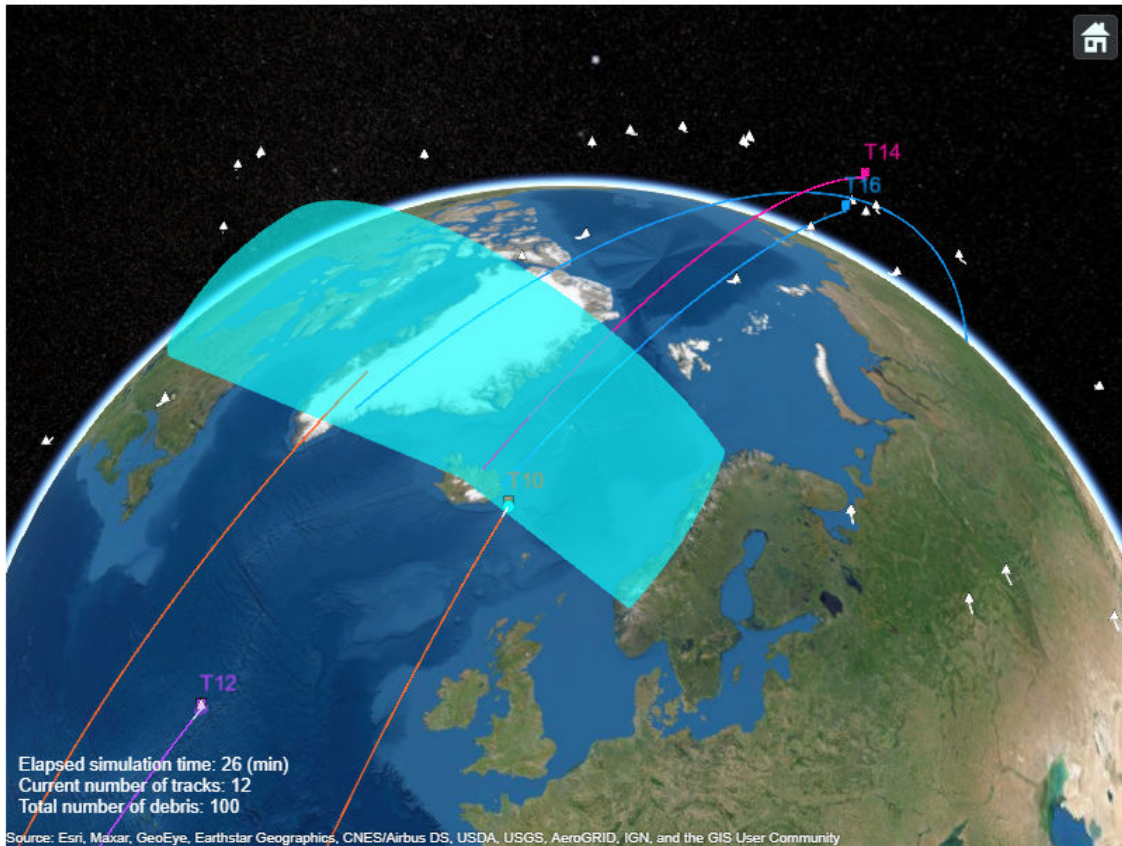
In the screenshot above, you can see that track T12 (in purple) is about to enter the radar surveillance area. Track T10 (in orange) was just updated with a detection, which reduced the uncertainty of its estimated position and velocity. With radar station configuration, after 30 minutes of surveillance, 15 tracks were initialized and confirmed out of the 100 debris objects. If you increase the simulation time, the radars will cover 360 degrees in space and eventually more debris can be tracked. Different radar station locations and configurations could be explored to increase the number of tracked objects.

Summary

In this example you have learned how to specify your own motion model to move platforms in a tracking scenario and how to use them to setup a tracker. This enables you to apply sensor fusion and tracking techniques offered in this toolbox to a wider range of applications, such as the problem of modelling and tracking space debris in an Earth-Centered-Earth-Fixed coordinate frame as shown in this example.

Supporting functions

The motion model used in this example is presented below. The state is the ECEF positions and velocities of the object `[x; vx; y; vy; z; vz]`.

```matlab
function state = keplerorbit(state,dt)
% keplerorbit performs numerical integration to predict the state of
% keplerian bodies. The state is [x;vx;y;vy;z;vz]

% Runge-Kutta 4 integration method:
k1 = kepler(state);
k2 = kepler(state + dt*k1/2);
k3 = kepler(state + dt*k2/2);
k4 = kepler(state + dt*k3);

state = state + dt*(k1+2*k2+2*k3+k4)/6;

    function dstate=kepler(state)
        x =state(1,:);
        vx = state(2,:);
        y=state(3,:);
        vy = state(4,:);
        z=state(5,:);
        vz = state(6,:);

        mu = 398600.4405*1e9; % m^3 s^-2
        omega = 7.292115e-5; % rad/s

        r = norm([x y z]);
        g = mu/r^2;

        % Coordinates are in a non-intertial frame, account for Coriolis
        % and centripetal acceleration
        ax = -g*x/r + 2*omega*vy + omega^2*x;
        ay = -g*y/r - 2*omega*vx + omega^2*y;
        az = -g*z/r;
        dstate = [vx;ax;vy;ay;vz;az];
    end
end
```

initKeplerUKF initializes a tracking filter with your own motion model. In this example, we use the same motion model that establishes ground truth, `keplerorbit`.

```matlab
function filter = initKeplerUKF(detection)

% assumes radar returns [x y z]
measmodel= @(x,varargin) x([1 3 5],:);
detNoise = detection.MeasurementNoise;
sigpos = 0.4;% m
sigvel = 0.4;% m/s^2
meas = detection.Measurement;
initState = [meas(1); 0; meas(2); 0; meas(3);0];
filter = trackingUKF(@keplerorbit,measmodel,initState,...
    'StateCovariance', diag(repmat([10, 10000].^2,1,3)),...
    'ProcessNoise',diag(repmat([sigpos, sigvel].^2,1,3)),...
    'MeasurementNoise',detNoise);

end
```

oe2rv converts a set of 6 traditional orbital elements to position and velocity vectors.

```matlab
function [r,v] = oe2rv(a,e,i,lan,w,nu)
% Reference: Bate, Mueller & White, Fundamentals of Astrodynamics Sec 2.5
```

```
mu = 398600.4405*1e9; % m^3 s^-2

% Express r and v in perifocal system
cnu = cosd(nu);
snu = sind(nu);
p = a*(1 - e^2);
r = p/(1 + e*cnu);
r_peri = [r*cnu ; r*snu ; 0];
v_peri = sqrt(mu/p)*[-snu ; e + cnu ; 0];

% Tranform into Geocentric Equatorial frame
clan = cosd(lan);
slan = sind(lan);
cw = cosd(w);
sw = sind(w);
ci = cosd(i);
si = sind(i);
R = [ clan*cw-slan*sw*ci  ,  -clan*sw-slan*cw*ci  ,     slan*si; ...
      slan*cw+clan*sw*ci  ,   -slan*sw+clan*cw*ci  ,    -clan*si; ...
      sw*si                         ,   cw*si                      ,      ci];
r = R*r_peri;
v = R*v_peri;
end
```

isDetectable is used in the example to determine which tracks are detectable at a given time.

```
function detectInput = isDetectable(tracker,time,covcon)

if ~isLocked(tracker)
    detectInput = zeros(0,1,'uint32');
    return
end
tracks = tracker.predictTracksToTime('all',time);
if isempty(tracks)
    detectInput = zeros(0,1,'uint32');
else
    alltrackid = [tracks.TrackID];
    isDetectable = zeros(numel(tracks),numel(covcon),'logical');
    for i = 1:numel(tracks)
        track = tracks(i);
        pos_scene = track.State([1 3 5]);
        for j=1:numel(covcon)
            config = covcon(j);
            % rotate position to sensor frame:
            d_scene = pos_scene(:) - config.Position(:);
            scene2sens = rotmat(config.Orientation,'frame');
            d_sens = scene2sens*d_scene(:);
            [az,el] = cart2sph(d_sens(1),d_sens(2),d_sens(3));
            if abs(rad2deg(az)) <= config.FieldOfView(1)/2 && abs(rad2deg(el)) < config.FieldOfV:
                isDetectable(i,j) = true;
            else
                isDetectable(i,j) = false;
            end
        end
    end

    detectInput = alltrackid(any(isDetectable,2))';
```

```
end
end
```

deleteBadTracks is used to remove tracks that obviously diverged. Specifically, diverged tracks in this example are tracks whose current position has fallen on the surface of the earth and tracks whose covariance has become too large.

```
function tracks = deleteBadTracks(tracker,tracks)
% remove divergent tracks:
%   - tracks with covariance > 4*1e8 (20 km standard deviation)
%   - tracks with estimated position outside of LEO bounds
n = numel(tracks);
toDelete = zeros(1,n,'logical');
for i=1:numel(tracks)
    [pos, cov] = getTrackPositions(tracks(i),[ 1 0 0 0 0 0 ; 0 0 1 0 0 0; 0 0 0 0 1 0]);
    if norm(pos) < 6500*1e3 || norm(pos) > 8500*1e3 || max(cov,[],'all') > 4*1e8
        deleteTrack(tracker,tracks(i).TrackID);
        toDelete(i) =true;
    end
end
tracks(toDelete) = [];
end
```

References

[1] https://www.esa.int/Safety_Security/Space_Debris/Space_debris_by_the_numbers

Simulate and Track En-Route Aircraft in Earth-Centered Scenarios

This example shows you how to use an Earth-Centered `trackingScenario` and a `geoTrajectory` object to model a flight trajectory that spans thousands of kilometers. You use two different models to generate synthetic detections of the airplane: a monostatic radar and ADS-B reports. You use a multi-object tracker to estimate the plane trajectory, compare the tracking performance, and explore the impact that ADS-B provides on the overall tracking quality.

In the United-States, the Federal Aviation Administration (FAA) is responsible for the regulation of thousands of flights everyday across the entire national airspace. Commercial flights are usually tracked at all time, from their departure airport to arrival. An air traffic control system is a complex multilevel system. Airport control towers are responsible for monitoring and handling the immediate vicinity of the airport while Air Route Traffic Control Centers (ARTCC) are responsible for long range en-route surveillance across various regions that compose the national airspace.

The capability as well as complexity of air traffic/surveillance radars have increased significantly over the past decades. The addition of transponders on aircraft adds a two-way communication between the radar facility and airplanes which allows for very accurate position estimation and benefits decision making at control centers. In 2020, all airplanes flying above 10,000 feet are required to be equipped with an Automatic Dependent Surveillance Broadcast (ADS-B) transponder to broadcast their on-board estimated position. This message is received and processed by air traffic controllers.

Create an En-Route Air Traffic scenario

You start by creating an Earth-centered scenario.

```
% Save current random generator state
s = rng;
% Set random seed for predictable results
rng(2020);
% Create scenario
scene = trackingScenario('IsEarthCentered',true,'UpdateRate',1);
```

Generate the true airplane trajectory

The matfile `flightwaypoints` attached in this example contains synthesized coordinates and time information of a flight trajectory from Wichita to Chicago. You use a `geoTrajectory` object to create the flight trajectory.

```
load('flightwaypoints.mat')
flightRoute = geoTrajectory(lla,timeofarrival);
airplane = platform(scene,'Trajectory',flightRoute);
```

Nowadays, commercial planes are all equipped with GPS receivers as part of their onboard instruments. To model the flight instruments, you set the `PoseEstimator` property of the airplane platform. As the backbone of ADS-B, the accuracy of onboard GPS can be set to conform with ADS-B requirements. The Navigation Accuracy Category used in ADS-B, for position and velocity are referred to as NACp and NACv. Per FAA regulation[1], the NACp must be less than 0.05 nautical miles and the NACv must be less than 10 meters per second. In this example, you use an `insSensor` model with a position accuracy of 100 m and a velocity accuracy of 10 m/s.

```
onboardINS = insSensor('PositionAccuracy',100,'VelocityAccuracy',10,...
    "RandomStream","mt19937ar with seed");
```

```
airplane.PoseEstimator = onboardINS;
load('737rcs.mat');
airplane.Signatures{1} = boeing737rcs;
```

Add surveillance stations along the route

There are several models of long-range surveillance radars used by the FAA. The Air Route
Surveillance Radar 4 (ARSR-4) is a radar introduced in the 1990s which can provide 3D returns of
any 1 square-meter object at long range of 250 nautical miles (463 kilometers). Most of ARSR-4
radars are located along the borders of the continental United-States, while slightly shorter range
radars are mostly located at FAA radar sites on the continent. In this example, a single radar type is
modeled following common specifications of an ARSR-4 as listed below:

- Update rate: 12 sec
- Maximum range (1 meter-square target): 463 km
- Range Resolution: 323 m
- Range Accuracy: 116 m
- Azimuth field of view: 360 deg
- Azimuth Resolution: 1.4 deg
- Azimuth Accuracy: 0.176 deg
- Height Accuracy: 900 m

A platform is added to the scenario for each radar site. The RCS signature of those platforms is set to
be -50 decibel to avoid creating unwanted radar returns.

By default, the radar detections are reported in the radar mounting platform body frame, which in
this case is the local NED frame at the position of each radar site. However, in this example you set
the `DetectionCoordinates` property to `Scenario` in order to output detections in the ECEF
frame, which allows the tracker to process all the detections from different radar sites in a common
frame.

```
% Model an ARSR-4 radar
updaterate = 1/12;
fov = [360;30.001];
elAccuracy = atan2d(0.9,463); % 900m accuracy @ max range
elBiasFraction = 0.1;

arsr4 = monostaticRadarSensor(1,'UpdateRate',updaterate,...
    'FieldOfView',fov,...,
    'HasElevation',true,...
    'ScanMode','Mechanical',...
    'MechanicalScanLimits',[0 360; -30 0],...
    'HasINS',true,...
    'HasRangeRate',true,...
    'HasFalseAlarms',false,...
    'ReferenceRange',463000,...
    'ReferenceRCS',0,...
    'AzimuthResolution',1.4,...
    'AzimuthBiasFraction',0.176/1.4,...
    'ElevationResolution',elAccuracy/elBiasFraction,...
    'ElevationBiasFraction',elBiasFraction,...
    'RangeResolution', 323,...
    'RangeBiasFraction',116/323,... Accuracy / Resolution
    'RangeRateResolution',100,...
```

```
        'DetectionCoordinates','Scenario');

% Add ARSR-4 radars at each ARTCC site
radarsitesLLA = [41.4228  -88.0583  0;...
    40.6989  -89.8258  0;...
    39.2219  -95.2461  0];

for i=1:3
    radar = clone(arsr4);
    radar.SensorIndex = i;
    platform(scene,'Position',radarsitesLLA(i,:),...
        'Signatures',rcsSignature('Pattern',-50),'Sensors',{radar});
end
```

Define a central tracker

Typically, one ARTCC maintains tracks for all objects within its surveillance zone and passes the tracking information to the next ARTCC as the object flies into a new zone. In this example, you define a single centralized tracker for all the radars. You use a `trackerGNN` object to fuse the radar detections of the plane from multiple radars with other sensor information such as ADS-B.

```
tracker = trackerGNN('FilterInitializationFcn',@initfilter,...
    'ConfirmationThreshold',[3 5],...
    'DeletionThreshold',[5 5],...
    'AssignmentThreshold',[1000 Inf]);
```

Visualize the scene

You use the `helperGlobeViewer` object attached in this example to display platforms, trajectories, detections, and tracks on the Earth.

```
gl = helperGlobeViewer;
setCamera(gl,[28.9176  -95.3388  5.8e5],[0 -30 10]);

% Show radar sites
plotPlatform(gl,scene.Platforms(2:end),'d');
% Show radar coverage
covcon = coverageConfig(scene);
plotCoverage(gl,covcon);
% Show flight route
plotTrajectory(gl,airplane);
% Take a snapshot
snap(gl);
```
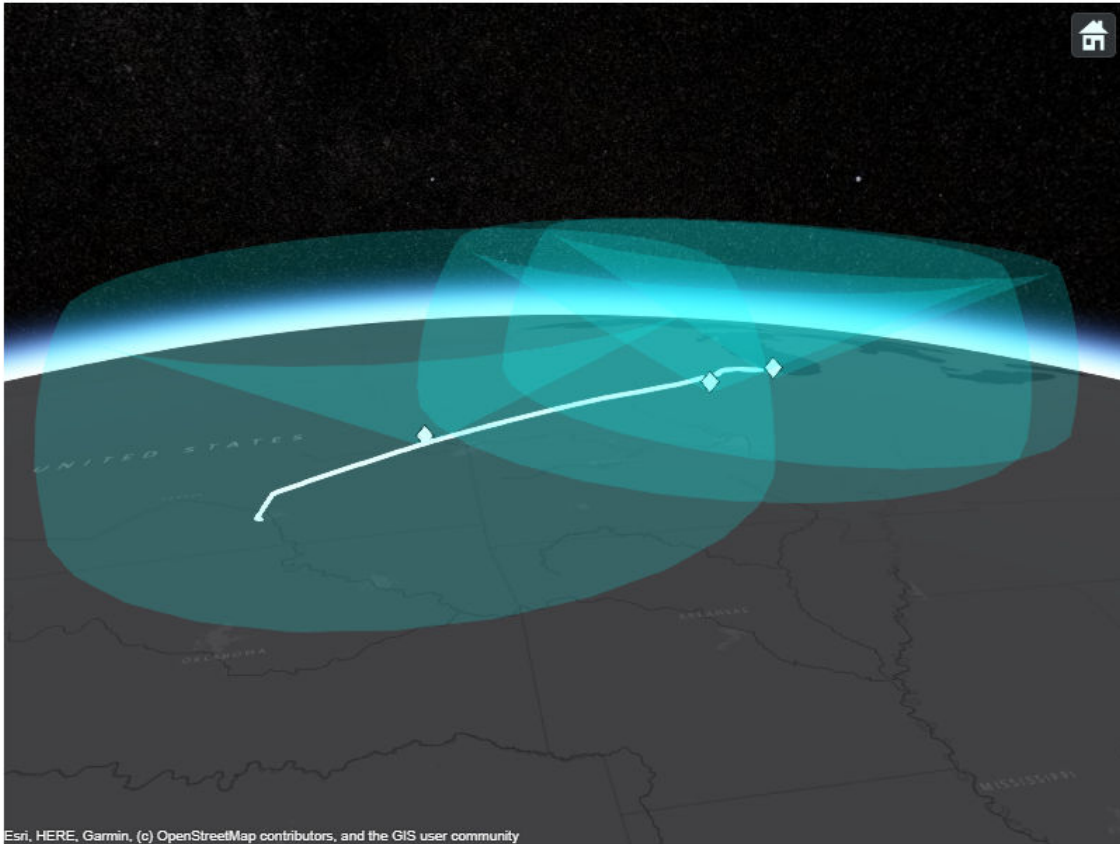
Track the flight using radars only

In this first simulation, you will simulate the scene and track the plane based solely on the long range radars.

```
useADSB = false;
snapTimes = [300 1600];

% Declare loop variables
detBuffer = {};
tLLA = [];
tHeading = [];
tSpeed = [];
tTime = [];
images = {};

% Set updata rates in seconds
trackupdatetime = 12;
adsbupdatetime  = 1;
arsrupdatetime  = 12;

while advance(scene)
    time = scene.SimulationTime;
```

```
% Update position of airplane on the globe
plotTarget(gl,'airplane',airplane,'^','full');

% Generate radar detections at the defined rate
if mod(time,arsrupdatetime) == 0
    % Generate synthetic radar detections
    dets = detect(scene);
    dets = removeBelowGround(dets);
    detBuffer = [detBuffer; dets]; %#ok<AGROW>
end

% Generate ADS-B detections at the defined rate
if useADSB && mod(time,adsbupdatetime) == 0
    adsb = adsbDetect(time,airplane);
    detBuffer = [detBuffer; adsb]; %#ok<AGROW>
end

% Fuse detections in tracker and update tracks on the globe
if mod(time,trackupdatetime) == 0 && time > 0
    % Update detections on the globe
    plotDetection(gl,detBuffer);
    % Tracker needs detections for first call i.e.
    cond = ~isempty(detBuffer) || ~isLocked(tracker);
    if cond
        tracks = tracker(detBuffer,time);
        if ~isempty(tracks)
            % Record the estimated airplane data
            [tLLA, tSpeed, tHeading, tTime] = ...
                helperNavigationData(tracks,tLLA, tSpeed, tHeading, tTime);
        end
        plotTrack(gl,tracks);
        detBuffer = {};
    end
end

% Move camera and take snapshots
images = moveCamera(gl,time,snapTimes,images);
end
```

Note that ARSR detections are relatively inaccurate in altitude, which is generally acceptable as air traffic controller separate airplanes horizontally rather than vertically. The least accurate detections are generated by radar sites located at longer ranges. These detections are nonetheless associated to the track. In the figure, the white line representes the true trajectory and the yellow line the trajectory estimated by the tracker.

```
for i=1:numel(images)
    figure
    imshow(images{i});
end
```
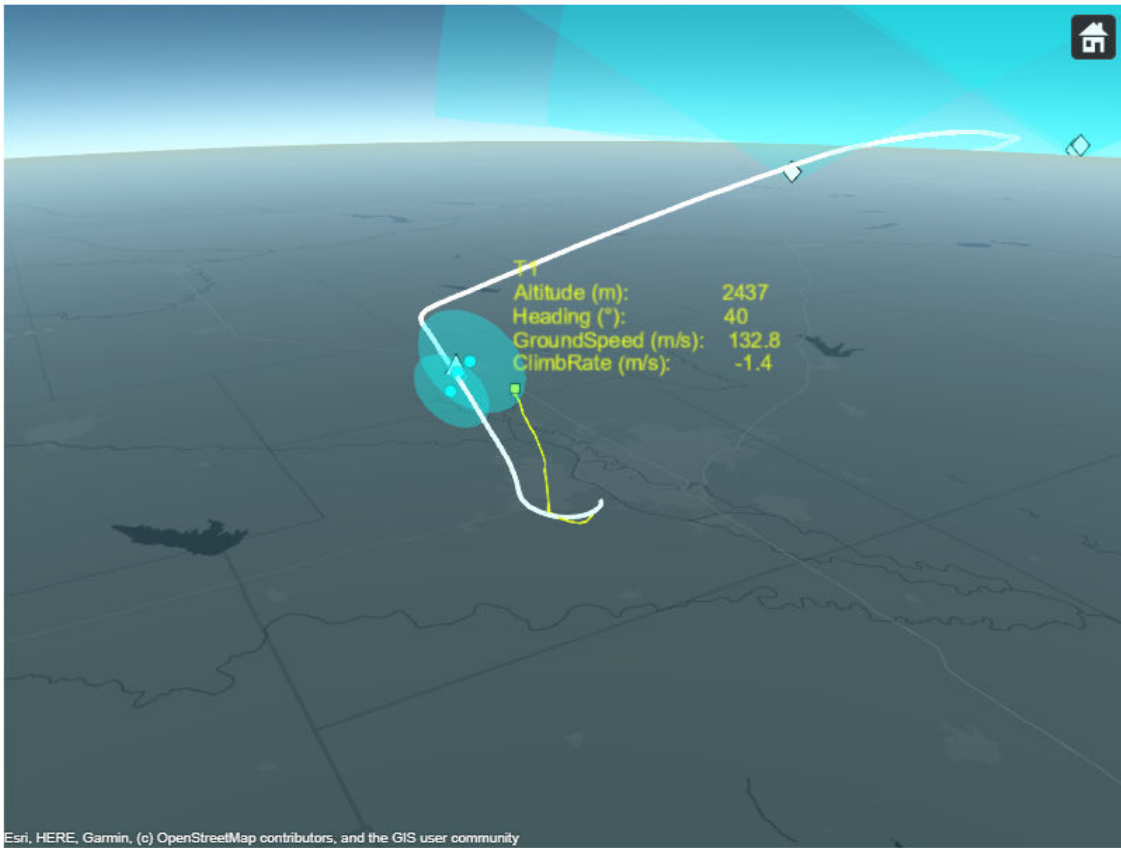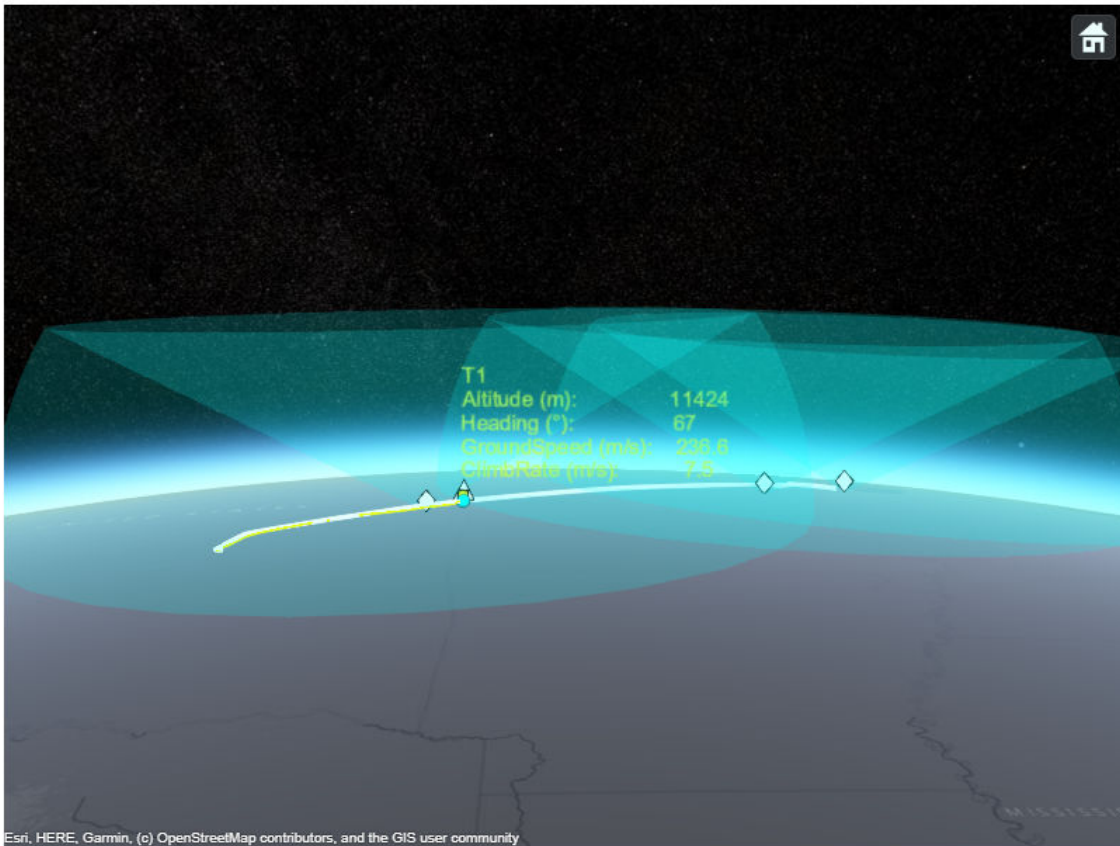
The first leg of the flight is far from any radar and detections have high measurement noise. Additionally, the constant velocity motion model does not model the motion well during the initial flight turn after take-off. As the airplane enters the coverage area of the second surveillance radar, additional detections correct the track shown on the second snapshot. The track (in yellow) follows the truth (in white) much closely as it enters the coverage area of the second radar.

Track the flight using radars and ADS-B

The utility function `adsbDetect` uses the estimated position measured by the airplane own navigation instrument modeled by the `PoseEstimator` property of the platform. The estimated position is usually encoded and broadcasted on 1090 MHz channel for nearby ADS-B receivers to pick up. in the United States, there is almost a full coverage of commercial flights cruising at altitudes of 10,000 meters or lower.

```
useADSB = true;
snapTimes = [200 1244];
[trackLLA2, trackSpeed2, trackHeading2, trackTime2, images] = ...
    simulateScene(gl,scene,tracker,useADSB,snapTimes);

% Reset random seed
rng(s);
```

Show snapshots at the same simulation time are shown below as in the previous section.

```
for i=1:numel(images)
    figure
    imshow(images{i});
end
```
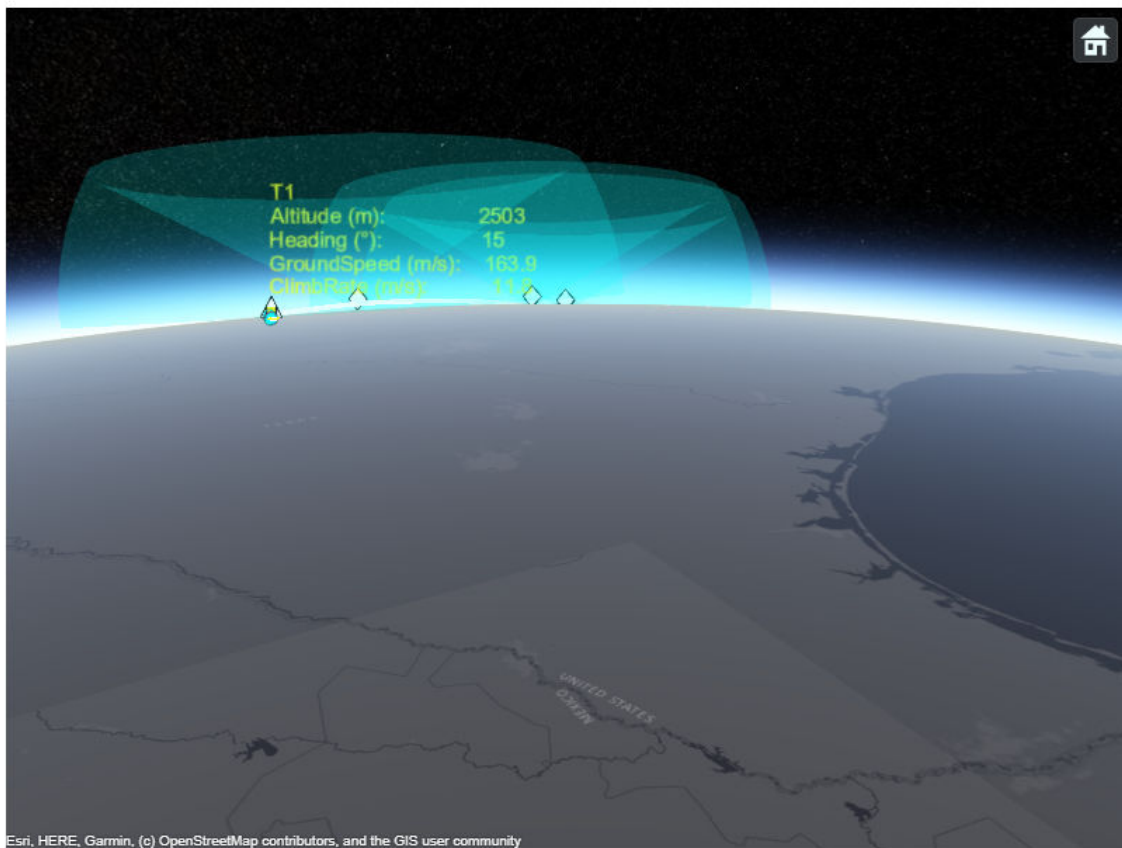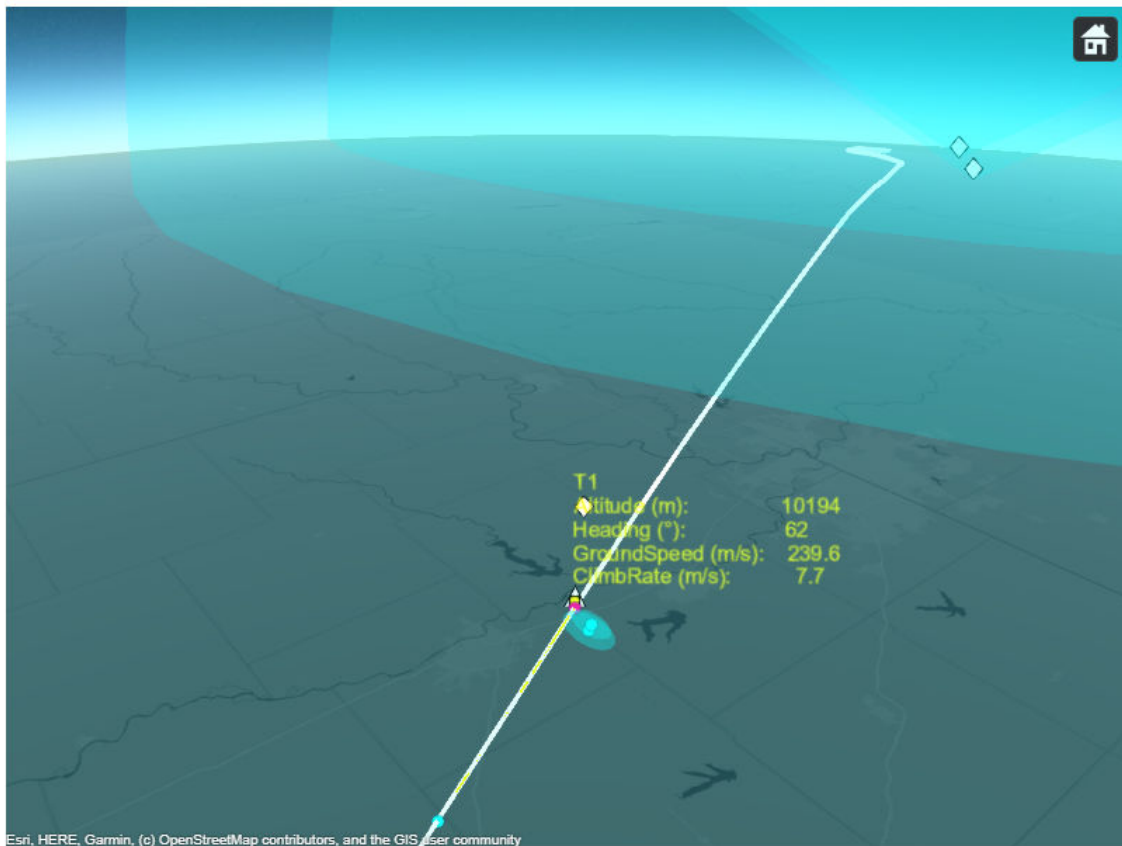
ADS-B detections are represented in purple color on the globe. The track matches more closely with the ground truth though no significant improvement is noticed while entering the middle radar coverage area.

Results

You compare the recorded track data in both simulations. The true position and velocity are available from the `geoTrajectory`. In air traffic control, dynamic states of interests are positions, represented by latitude, longitude, and altitude, heading, and groundspeed.

```
% Query truth at the recorded timestamps for both runs
[trueLLA, ~, trueVel] = lookupPose(flightRoute,tTime);
[trueLLAadsb, ~, trueVeladsb] = lookupPose(flightRoute,trackTime2);

figure
tiledlayout(3,1);
nexttile
hold on
plot(tTime/60,trueLLA(:,3),'LineWidth',2,'LineStyle','--','DisplayName','Truth');
plot(tTime/60,tLLA(:,3),'DisplayName','Surveillance radar only');
plot(trackTime2/60,trackLLA2(:,3),'DisplayName','Surveillance radar + ADS-B');
legend('Location','northeastoutside')
xlabel('Time (minute)');
```

```
ylabel('Altitude (meter)')
title('Altitude')

nexttile
hold on
plot(tTime/60,vecnorm(trueVel(:,(1:2)),2,2),'LineWidth',2,'LineStyle','--');
plot(tTime/60,tSpeed);
plot(trackTime2/60,trackSpeed2);
xlabel('Time (minute)');
ylabel('Speed (m/s)')
title('Groundspeed')

nexttile
hold on
plot(tTime/60,atan2d(trueVel(:,2),trueVel(:,1)),'LineWidth',2,'LineStyle','--');
plot(tTime/60,tHeading);
plot(trackTime2/60,trackHeading2);
xlabel('Time (minute)');
ylabel('Heading (degree)')
title('Heading')
```
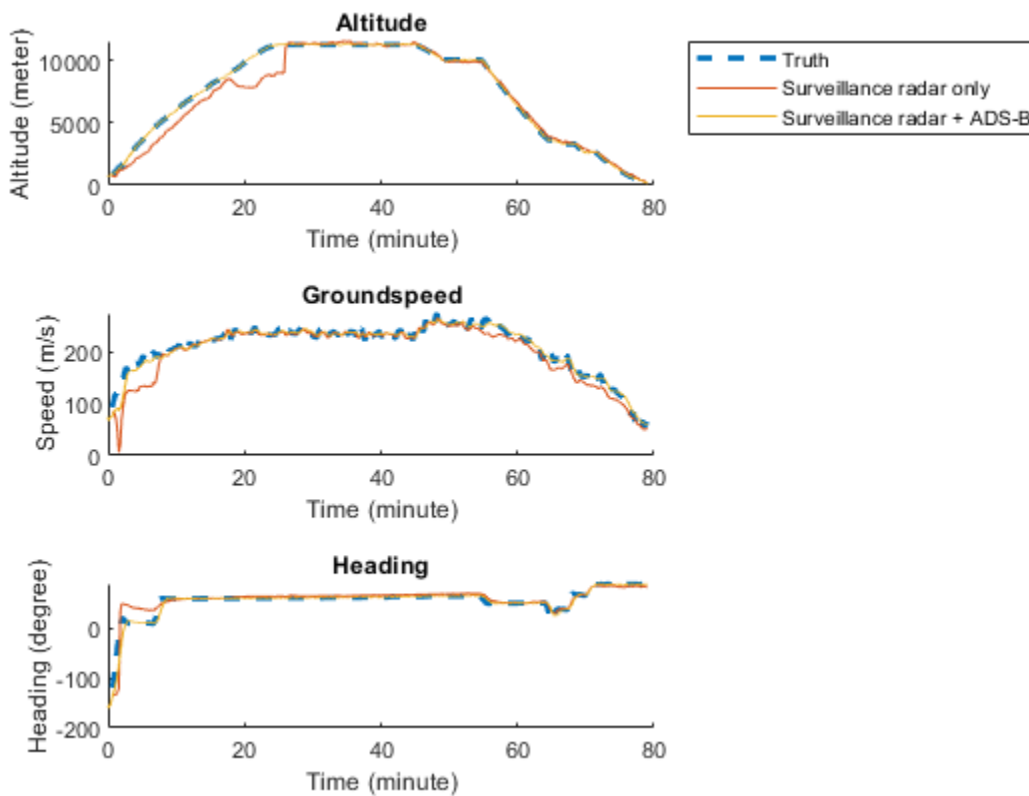

The results show that the additional data provided by ADS-B has a significant effect on the altitude estimation as altitude is the least accurate information reported by the radars. Meanwhile, the groundspeed and heading estimate is also improved with the benefit of more frequent updates (every second vs every 12 seconds).

Summary

In this example you have learned how to create an Earth-centered scenario and define trajectories using geodetic coordinates. You also learned how to model Air Route Surveillance Radars and generate synthetic detections. You feed these detections to a multi-object tracker and estimate the position, velocity, and heading of the plane. The tracking performance is improved by adding and fusing of ADS-B information. You used a simple approach to model ADS-B reports and integrate them to tracking solution. In this example, only a single flight was modeled. However, the benefit of ADS-B can be further manifested when modeling multiple flights in scenario where safe separation distances must be maintained by air traffic controllers.

Supporting functions

simulateScene contains the simulation loop which can be run with or without ADS-B. The scenario is updated every second. ADS-B detections are generated every second and ARSR detections are available at the end of every 12 seconds scan. The function outputs the estimated position of the plane as latitude (deg), longitude (deg), and WGS84 altitude (m), the estimated ground speed (m/s), the estimated heading (deg), and the corresponding simulation time (s).

```
function [tLLA,tSpeed,tHeading,tTime,images] = simulateScene(gl,scene,tracker,useADSB,snapTimes)
% Reset the scenario, globe, and seed
rng(2020);
restart(scene);
airplane = scene.Platforms{1};
release(tracker);
clear(gl);
setCamera(gl,[39.4563 -90.1187 2.356e6]);

% Display initial state of the scene
covcon = coverageConfig(scene);
plotPlatform(gl,scene.Platforms(2:end),'d');
plotCoverage(gl,covcon);

% Declare loop variables
detBuffer = {};
tLLA = [];
tHeading = [];
tSpeed = [];
tTime = [];
images = {};

% Set update rates in seconds
if useADSB
    trackupdatetime = 1;
else
    trackupdatetime = 12;
end
adsbupdatetime  = 1;
arsrupdatetime  = 12;

while advance(scene)
    time = scene.SimulationTime;

    % Update position of airplane on the globe
    plotTarget(gl,'airplane',airplane,'^','full');

    % Generate radar detections at the defined rate
```

```matlab
        if mod(time,arsrupdatetime) == 0 && time > 0
            % Generate synthetic radar detections
            dets = detect(scene);
            dets = removeBelowGround(dets);
            detBuffer = [detBuffer; dets]; %#ok<AGROW>
        end

        % Generate ADS-B detections at the defined rate
        if useADSB && mod(time,adsbupdatetime) == 0
            adsb = adsbDetect(time,airplane);
            detBuffer = [detBuffer; adsb]; %#ok<AGROW>
        end

        % Update detections on the globe
        plotDetection(gl,detBuffer);

        % Fuse detections in tracker and update tracks on the globe
        if mod(time,trackupdatetime) == 0 && time > 0
            % Tracker needs detections for first call
            cond = ~(isempty(detBuffer) && ~isLocked(tracker));
            if cond
                tracks = tracker(detBuffer,time);
                if ~isempty(tracks)
                    % Record the estimated airplane data
                    [tLLA, tSpeed, tHeading, tTime] = ...
                        helperNavigationData(tracks,tLLA, tSpeed, tHeading, tTime);
                end
                plotTrack(gl,tracks);
                detBuffer = {};
            end
        end

        % Move camera and take snapshots
        images = moveCamera(gl,time,snapTimes,images);

    end
end
```

adsbDetect generates detection of the airplane based on its on-board sensor suite as modeled by the `insSensor`.

```matlab
function detection = adsbDetect(time,airplane,~)
%adsbDetect generate ADS-B detections

estimpose = pose(airplane,'CoordinateSystem','Cartesian');
meas = [estimpose.Position(:) ; estimpose.Velocity(:)];
measnoise = diag([100 100 100 10 10 10].^2);

mparams = struct('Frame','Rectangular',...
    'OriginPositin',zeros(3,1),...
    'OriginVelocity',zeros(3,1),...
    'Orientation',eye(3),... ADS-B does not report orientation
    'IsParentToChild', 1,...
    'HasAzimuth', 1,...
    'HasElevation', 1,...
    'HasRange', 1, ...
    'HasVelocity', 1);
```

```
attr ={ struct('TargetIndex',airplane.PlatformID,'SNR',10)};

detection = {objectDetection(time, meas, 'SensorIndex',20,...
    'MeasurementNoise',measnoise,...
    'MeasurementParameters',mparams,...
    'ObjectAttributes',attr)};
end
```

initfilter defines the extended kalman filter used by `trackerGNN`. Airplane motion is well approximated by a constant velocity motion model. Therefore a rather small process noise will give more weight to the dynamics compared to the measurements which are expected to be quite noisy at long ranges.

```
function filter = initfilter(detection)
filter = initcvekf(detection);
filter.StateCovariance = 4*filter.StateCovariance; % initcvekf uses measurement noise as the defa
filter.ProcessNoise = 0.02*eye(3);
end
```

removeBelowGround removes radar detections that lie below the surface of the Earth as these cannot be created by a real radar.

```
function detsout = removeBelowGround(detsin)
n = numel(detsin);
keep = zeros(1,n,'logical');
for i=1:n
    meas = detsin{i}.Measurement(1:3)';
    lla = fusion.internal.frames.ecef2lla(meas);
    if lla(3)>0
        keep(i) = true;
    else
        keep(i) = false;
    end
end
detsout = detsin(keep);
end
```

moveCamera specifies new camera positions to follow the airplane and take snapshots.

```
function images = moveCamera(gl, time, snapTimes,images)

if time == 120
    setCamera(gl,[37, -97.935, 4.328e4],[0 -23 30]);
end

if time == 1244
    setCamera(gl,[38.8693 -95.6109 1.214e4],[0 -26.8 38.7]);
end

if time == 1445
    setCamera(gl,[37.995, -94.60 6.87e4],[0 -15 2]);
end

if time == 2100
    setCamera(gl, [38.9747 -94.6385 1.3e5], [0 -20 55]);
end

if time == 3000
```

```
    setCamera(gl,[41.636 -87.011 6.33e4],[0 -25 270]);
end

% Snaps
if any(time == snapTimes)
    img = snap(gl);
    images = [ images, {img}];
end
end
```

Reference

[1] FAR §91.227 Automatic Dependent Surveillance-Broadcast (ADS-B) Out equipment performance requirements

Simulate, Detect, and Track Anomalies in a Landing Approach

The example shows how to automatically detect deviations and anomalies in aircraft making final approaches to an airport runway. In this example, you will model an ideal landing approach trajectory and generate variants from it, simulate radar tracks, and issue warnings as soon as the tracks deviate from safe landing rules.

Introduction

Landing is a safety critical stage of flight. An aircraft in final approach to landing must align itself with the runway, gradually descend to the ground, and reduce its ground speed while keeping it safely above stall speed. All these steps are done to ensure that the aircraft touches the ground softly to reduce the risk to passengers and to avoid physical damage to the aircraft or the runway. These rules can be easily defined by an aviation professional or they can be inferred from tracking data using machine learning [1]. In this example, you assume that the rules are already defined.

Major airports usually have multiple runways oriented in different directions. Approaching aircraft are guided by the air traffic controllers in the airport tower to land on one of the runways that is best aligned against the direction of wind at that time. During the approach, controllers monitor the aircraft based on tracking systems. Airport traffic has increased over the last decades and, with it, the workload on air traffic controllers has increased as well. As a result, there is a need to automatically and reliably alert air traffic controllers to aircraft that approach the landing point in an unsafe manner: not aligned well with the runway, descending too quickly or too slowly, or approaching too quickly or too slowly.

Generate and Label Truths

You define a landing approach trajectory into Logan International Airport in Boston, MA using a `geoTrajectory` object. The trajectory waypoints are aligned with runway 22L, which runs from north-east to south-west, and a glide slope of 3 degrees. The time of arrival and climb rate are defined to slow the approaching aircraft to a safe speed and a smooth touchdown. Note that the positive value of climb rate is used for a descending trajectory. You use `helperScenarioGlobeViewer` (see the supporting file) to visualize the trajectory on the map.

```
baselineApproachTrajectory = geoTrajectory([42.7069 -70.8395 1500; 42.5403 -70.9203 950; 42.3736
    'ClimbRate', [6; 3.75; 3.75]);

viewer = helperScenarioGlobeViewer;
viewer.TargetHistoryLength = 0;
viewer.TrackHistoryLength = 0;
viewer.TrackLabelScale = 0.75;
positionCamera(viewer, [42.3072 -70.8463 12455], [0 -34 335]);
plotTrajectory(viewer, baselineApproachTrajectory, 'Color', [15 255 255]/255, "Width", 1);
```

For a trajectory coming to land at Logan airport on runway 22L to be safe, the trajectory must satisfy the following rules:

- The trajectory must be closely aligned with the runway direction.
- The glide slope must be between 2.5 and 4 degrees in the last 20963 meters. At distances above 20963 meters, the altitude must be at least 3000 ft.
- The speed must be between 120 knots and 180 knots at the landing point. The upper speed bound can increase linearly with distance from the landing point.

You define these rules using the `helperTrajectoryValidationRule` (see the supporting file) and visualize the rules on the map.

```
% Define and show trajectory rules.
trajRules = defineTrajectoryRules();
showRules(viewer, trajRules)
snap(viewer)
```
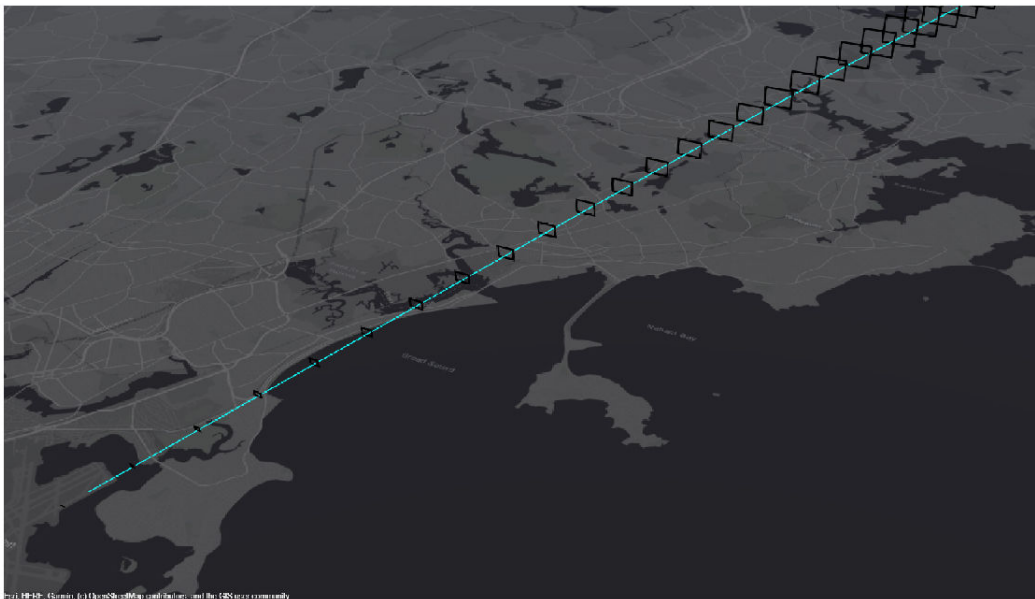

You use the `perturbations` object function to define a normal distribution around the baseline trajectory, `approachTrajectory`. Each waypoint in the trajectory is perturbed with a zero-mean normal distribution and a standard deviation that gets smaller from the first waypoint to the last (the landing point). At the first waypoint, the standard deviation is 5e-3 degrees in longitude and 300 meters in altitude. The standard deviation decreases to 1e-3 degrees in longitude and 150 meters in altitude in the mid-point and then to 1e-4 degrees in longitude and 0 in altitude at the end-point on the ground.

```
% Define perturbation to the approach trajectory
perturbations(baselineApproachTrajectory, 'Waypoints', 'Normal', zeros(3,3), [0 5e-3 300; 0 1e-3
```

To create 20 trajectories that are perturbed from the baseline trajectory, first `clone` the trajectory and then `perturb` it.

```
% Generate perturbed trajectories
s = rng(2021, 'twister'); % Set random noise generator for repeatable results
numTrajectories = 20;
trajectories = cell(1,numTrajectories);
for i = 1:numTrajectories
    trajectories{i} = clone(baselineApproachTrajectory);
    perturb(trajectories{i});
end
```

To see which perturbed trajectories satisfy the rules of a safe approach to landing, you use the helper function `validateTrajectory`, provided at the bottom of this page. The function declares a trajectory to be anomalous if at least 1% of the points sampled from it violate any trajectory rule.

```
[truthAnomalyFlags, truthPercentAnomalousSteps] = validateTrajectory(trajectories, trajRules);
```

Plot the trajectories in yellow for anomalous trajectories and cyan for safe approaches. Overall, there are 7 anomalous trajectories out of the 20 generated trajectories.

```
plotTrajectory(viewer, trajectories(truthAnomalyFlags), 'Color', [255 255 17]/255, "Width", 1);
plotTrajectory(viewer, trajectories(~truthAnomalyFlags), 'Color', [15 255 255]/255, "Width", 1);
positionCamera(viewer, [42.4808 -70.916 1136], [0 0 340]);
snap(viewer)
```
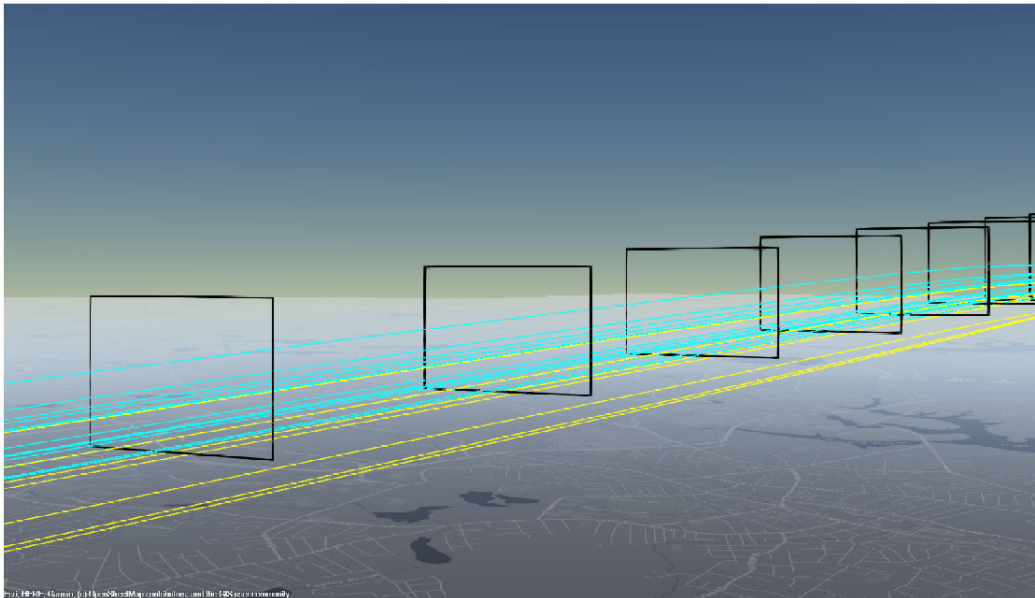

Define a Scenario

Detecting anomalies in real time based on tracking data is a challenge for several reasons. First, since the tracking data is imperfect with noise, the tracking results are uncertain. As a result, some tolerances must be provided to avoid issuing false warnings. Second, the sensors report false detections, and the tracking system must be careful not to confirm tracks based on these false detections. The careful confirmation requires that the tracking system takes more time to confirm the tracks. To avoid excessive warnings on false tracks, you issue warnings only after a track is confirmed.

You define an Earth-centered tracking scenario.

```
% Create an Earth-centered tracking scenario
scenario = trackingScenario('UpdateRate', 1, 'IsEarthCentered', true);
```

Aircraft approaching landing in an airport are scheduled to avoid aerodynamic impact from one aircraft on the one following in its wake. The minimum safe time difference between two aircraft is one minute.

You use the `perturbations` and `perturb` object functions again to perturb the `TimeOfArrival` of each trajectory and to make sure no additional perturbations are applied to the `Waypoints`. Then you attach each trajectory to a new platform. To perturb the entire scenario, you use the `perturb` object function.

```
% Schedule the trajectories and attach each to a platform.
for i = 1:numTrajectories
    perturbations(trajectories{i}, 'TimeOfArrival', 'Uniform',(i-1)*60, (i-1)*60+10);
    perturbations(trajectories{i}, 'Waypoints', 'None');
    platform(scenario, 'Trajectory', trajectories{i});
end
perturb(scenario);
```

Like other major airports in the USA, Logan uses an Airport Surface Detection Equipment - Model X (ASDE-X) to track aircraft during final approach and on the ground [2]. ASDE-X relies on an airport surveillance radar, automatic dependent surveillance–broadcast (ADS–B) reports from the approaching aircraft, and other methods to provide accurate tracking which is updated every second (for more details, see [1]).

To simplify the model of this tracking system, you use a statistical radar model, `monostaticRadarSensor`, attached to the airport tower, and connect the sensor to a `trackerGNN` object. You configure the tracker to be conservative about confirming tracks by setting the `ConfirmationThreshold` to confirm if a track receives 4-out-of-5 updates.

```
asdex = monostaticRadarSensor(1, ...
    'ScanMode', 'No Scanning', ...
    'MountingAngles', [0 0 0], ...
    'FieldOfView', [360;20], ...
    'UpdateRate', 1, ...
    'ReferenceRange', 40000,...
    'RangeResolution', 100, ...
    'HasElevation', true, ...
    'HasINS', true, ...
    'DetectionCoordinates', 'Scenario', ...
    'FalseAlarmRate', 1e-7, ...
    'ElevationResolution', 0.4, ...
    'AzimuthResolution', 0.4);
p = platform(scenario, 'Position', [42.3606 -71.011 0], 'Sensors', asdex);
tracker = trackerGNN("AssignmentThreshold", [100 2000], "ConfirmationThreshold", [4 5]);
tam = trackAssignmentMetrics('AssignmentThreshold', 100, 'DivergenceThreshold', 200);
```

Run the Scenario and Detect Anomalous Tracks

In the following lines, you simulate the scenario and track the approaching aircraft. You use the `validateTracks` helper function to generate anomaly warnings for the tracks. You can see the code for the function at the bottom of this page.

Tracks that violate the safe approach rules are shown in yellow while tracks that follow these rules are shown in cyan. Note that the warning is issued immediately when the track violates any rule and is removed when it satisfies all the rules.

```
% Clean the display and prepare it for the simulation.
clear(viewer)
```

```
positionCamera(viewer, [42.3072 -70.8463 12455], [0 -34 335]);
showRules(viewer, trajRules)
snaptime = 930;
clear validateTracks

% Main loop
while advance(scenario)
    % Collect detections
    dets = detect(scenario);

    % Update the tracker and output tracks.
    if ~isempty(dets) || isLocked(tracker)
        tracks = tracker(dets, scenario.SimulationTime);
    else
        tracks = objectTrack.empty;
    end

    % Get platform poses and assignment between tracks and truths.
    poses = platformPoses(scenario,"Quaternion","CoordinateSystem","Cartesian");
    tam(tracks,poses);
    [assignedTrackIDs, assignedTruthIDs] = currentAssignment(tam);

    % Validate the tracks with rules to find anomalous tracks.
    [tracks, trackAnomalyHistory] = validateTracks(tracks, trajRules, assignedTrackIDs, assignedT

    % Visualize
    updateDisplay(viewer,scenario.SimulationTime,[scenario.Platforms{:}],dets,[],tracks);
end
rng(s); % Reset the random number generator to its value before running the example.
```

The following gif was taken when for a minute of simulation from the 900 seconds to the 960 seconds. It shows tracks identified as safe in cyan and tracks identified as anomalous in yellow. This identification is done at every simulation step as can be seen for track 2489.
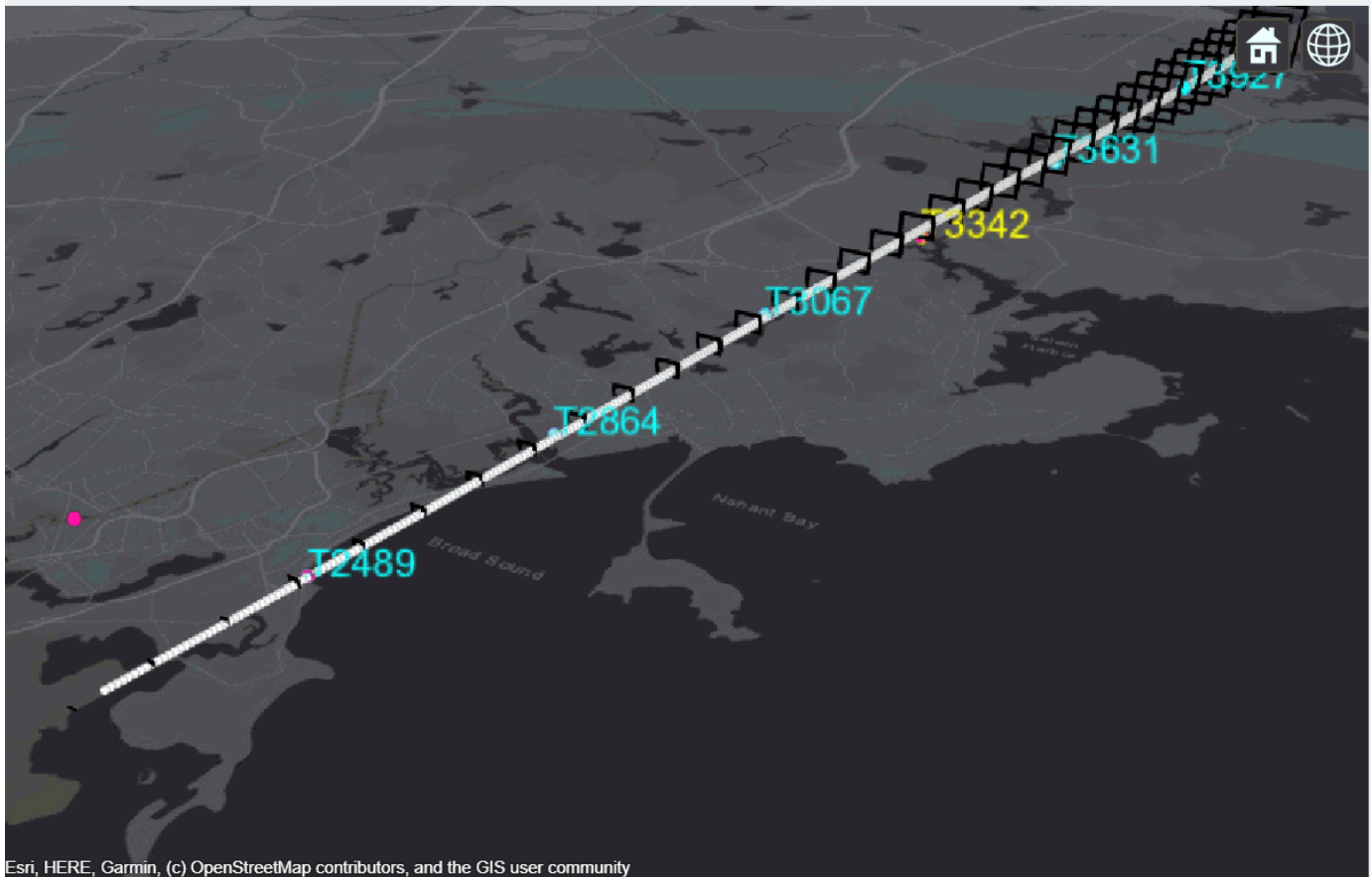
Compare Track Anomaly Reports to Truth

To verify that the anomaly warnings were issued for the right tracks, you use the `analyze` helper function, shown at the bottom of this page.

The function uses the `trackAnomalyHistory` collected during the simulation and compares it to the `truthPercentAnomalousSteps` calculated for each trajectory. Similar to truth, tracks are assigned anomaly flags if they were declared anomalous at least 1% of the time steps. You can see that the anomalies are issued correctly for the seven trajectories that were found to be anomalous.

```
comparisonTable = analyze(trackAnomalyHistory ,truthPercentAnomalousSteps);
disp(comparisonTable)
```

TruthID	Truth Anomaly Flag	Track Anomaly Flag
1	false	false
2	false	false
3	false	false
4	false	false
5	false	false
6	false	false
7	true	true
8	true	true
9	false	false
10	true	true

```
11              false               false
12              false               false
13              true                true
14              true                true
15              false               false
16              false               false
17              false               false
18              false               false
19              true                true
20              true                true
```

Summary

In this example, you learned how to use tracking data to generate real-time warnings for anomalies like an unsafe landing approach.

You used `geoTrajectory` to define an ideal landing approach trajectory in geographic coordinates. You then used `perturbations` and `perturb` to create 20 trajectories that deviate from the ideal landing approach trajectory and to schedule the trajectories one after the other in the `trackingScenario`. To model an airport tracking system, you simplified the system model using a statistic radar model, by the `monostaticRadarSensor` System object, and a tracker, by the `trackerGNN` System object.

References

1 Raj Deshmukh and Inseok Hwang, "Anomaly Detection Using Temporal Logic Based Learning for Terminal Airspace Operations", AIAA SciTech Forum, 2019.

2 Federal Aviation Administration, "Fact Sheet - Airport Surface Detection Equipment, Model X (ASDE-X)". https://www.faa.gov/news/fact_sheets/news_story.cfm?newsId=17095 retrieved May 2020.

Supporting Functions

defineTrajectoryRules Define trajectory rules

```
function trajRules = defineTrajectoryRules
% This function defines rules for safe approach to landing on runway 22L at
% Logan International Airport in Boston, MA.

% The function uses the helperTrajectoryValidationRule attached as a
% supporting file to this example

% The trajectory must be closely aligned with the runway direction.
longitudeRule = helperTrajectoryValidationRule([42.37 42.71], [0.4587, -90.4379], [0.5128, -92.7

% The glide slope must be between 2.5 and 4 degrees in the last 20963
% meters. At distances above 20963 meters, the altitude must be at least
% 3000 ft. The rules are relative to range from the runway landing point.
altitudeRule1 = helperTrajectoryValidationRule([100 20963], [sind(2.5) 0], [sind(4) 0]);
altitudeRule2 = helperTrajectoryValidationRule([20963 40000], 3000 * 0.3048, [sind(4) 0]);

% The speed must be between 120 knots and 180 knots at the landing point.
% The upper speed bound can increase linearly with distance from the
% landing point.
speedRule = helperTrajectoryValidationRule([0 40000], 61.733, [1e-3 100]);

% Collect all the rules.
```

```matlab
trajRules = [longitudeRule;altitudeRule1;altitudeRule2;speedRule];
end
```

validateTrajectory Validate each trajectory

```matlab
function [truthAnomalyFlags, percentAnomalousSteps] = validateTrajectory(trajectories, rules)
numTrajectories = numel(trajectories);
numAnomalousSteps = zeros(1, numTrajectories);
for tr = 1:numTrajectories
    if iscell(trajectories)
        traj = trajectories{tr};
    elseif numTrajectories == 1
        traj = trajectories;
    else
        traj = trajectories(tr);
    end
    timesamples = (traj.TimeOfArrival(1):traj.TimeOfArrival(end));
    [pos,~,vel] = lookupPose(traj, timesamples);
    posECEF = lookupPose(traj, timesamples, 'ECEF');
    landingPoint = [1.536321  -4.462053   4.276352]*1e6;

    for i = 1:numel(timesamples)
        distance = norm(posECEF(i,:) - landingPoint);
        isLongitudeValid = validate(rules(1),pos(i,1),pos(i,2));
        isAltitudeValid = (validate(rules(2),distance,pos(i,3)) && validate(rules(3),distance,pos
        isSpeedValid = validate(rules(4),distance,norm(vel(i,:)));
        isValid = isLongitudeValid && isAltitudeValid && isSpeedValid;
        numAnomalousSteps(tr) = numAnomalousSteps(tr) + ~isValid;
    end
end
percentAnomalousSteps = numAnomalousSteps ./ numel(timesamples) * 100;
truthAnomalyFlags = (percentAnomalousSteps > 1);
end
```

validateTrack Validate tracks vs anomaly rules

```matlab
function [tracks, history] = validateTracks(tracks, rules, assignedTrackIDs, assignedTruthIDs)
persistent trackAnomalyHistory
if isempty(trackAnomalyHistory)
    trackAnomalyHistory = repmat(struct('TrackID', 0, 'AssignedTruthID', 0, 'NumSteps', 0, 'NumAn
end

posECEF = getTrackPositions(tracks, [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0  1 0]);
pos = fusion.internal.frames.ecef2lla(posECEF);
vel = getTrackVelocities(tracks, [0 1 0 0 0 0; 0 0 0 1 0 0; 0 0 0 0 0 1]);
numTracks = numel(tracks);
landingPoint = [1.536321  -4.462053   4.276352]*1e6;

trackIDs = [trackAnomalyHistory.TrackID];

for tr = 1:numTracks
    % Only validate tracks if altitude is greater than 0
    if pos(tr,3) > 15
        distance = norm(posECEF(tr,:) - landingPoint);
        isLongitudeValid = validate(rules(1),pos(tr,1),pos(tr,2));
        isAltitudeValid = (validate(rules(2),distance,pos(tr,3)) && validate(rules(3),distance,po
        isSpeedValid = validate(rules(4),distance,norm(vel(tr,:)));
        isValid = isLongitudeValid && isAltitudeValid && isSpeedValid;
```

5-625

```
        else
            isValid = true;
        end

        tracks(tr).ObjectClassID = uint8(~isValid) + uint8(isValid)*6; % To get the right color for t

        % Update anomlay history
        inHistory = (tracks(tr).TrackID == trackIDs);
        if any(inHistory)
            trackAnomalyHistory(inHistory).NumSteps = trackAnomalyHistory(inHistory).NumSteps + 1;
            trackAnomalyHistory(inHistory).NumAnomalousSteps = trackAnomalyHistory(inHistory).NumAnon
        else
            ind = find(trackIDs == 0, 1, 'first');
            trackAnomalyHistory(ind).AssignedTruthID = assignedTruthIDs(tracks(tr).TrackID == assigne
            trackAnomalyHistory(ind).TrackID = tracks(tr).TrackID;
            trackAnomalyHistory(ind).NumSteps = 1;
            trackAnomalyHistory(ind).NumAnomalousSteps = ~isValid;
        end
    end
    history = trackAnomalyHistory;
end
```

analyze - Analyze the track anomaly history and compare it to the truth anomaly percentage

```
function comparisonTable = analyze(trackAnomalyHistory, percentTruthAnomalous)
trackAnomalyHistory = trackAnomalyHistory([trackAnomalyHistory.TrackID] > 0);
anomalousSteps = [trackAnomalyHistory.NumAnomalousSteps];
numSteps = [trackAnomalyHistory.NumSteps];
trackAssignedTruths = [trackAnomalyHistory.AssignedTruthID];
assignedTruths = unique(trackAssignedTruths);
numTrackAnomalousSteps = zeros(numel(assignedTruths),1);
numTrackSteps = zeros(numel(assignedTruths),1);

for i = 1:numel(assignedTruths)
    inds = (assignedTruths(i) == trackAssignedTruths);
    numTrackAnomalousSteps(i) = sum(anomalousSteps(inds));
    numTrackSteps(i) = sum(numSteps(inds));
end
percentTrackAnomalous = numTrackAnomalousSteps ./ numTrackSteps * 100;
trueAnomaly = (percentTruthAnomalous > 1)';
anomalyFlags = (percentTrackAnomalous > 1);
comparisonTable = table((1:20)',trueAnomaly, anomalyFlags,...
    'VariableNames', {'TruthID', 'Truth Anomaly Flag', 'Track Anomaly Flag'});
end
```

Generate Code for a Track Fuser with Heterogeneous Source Tracks

This example shows how to generate code for a track-level fusion algorithm in a scenario where the tracks originate from heterogeneous sources with different state definitions. This example is based on the "Track-Level Fusion of Radar and Lidar Data" on page 5-536 example, in which the state spaces of the tracks generated from lidar and radar sources are different.

Define a Track Fuser for Code Generation

You can generate code for a `trackFuser` using MATLAB® Coder™. To do so, you must modify your code to comply with the following limitations:

Code Generation Entry Function

Follow the instructions on how to use "System Objects in MATLAB Code Generation" (MATLAB Coder). For code generation, you must first define an entry-level function, in which the object is defined. Also, the function cannot use arrays of objects as inputs or outputs. In this example, you define the entry-level function as the heterogeneousInputsFuser function. The function must be on the path when you generate code for it. Therefore, it cannot be part of this live script and is attached in this example. The function accepts local tracks and current time as input and outputs central tracks.

To preserve the state of the fuser between calls to the function, you define the fuser as a `persistent` variable. On the first call, you must define the fuser variable because it is empty. The rest of the following code steps the `trackFuser` and returns the fused tracks.

```
function tracks = heterogeneousInputsFuser(localTracks,time)
%#codegen

persistent fuser
if isempty(fuser)
    % Define the radar source configuration
    radarConfig = fuserSourceConfiguration('SourceIndex',1,...
        'IsInitializingCentralTracks',true,...
        'CentralToLocalTransformFcn',@central2local,...
        'LocalToCentralTransformFcn',@local2central);

    % Define the lidar source configuration
    lidarConfig = fuserSourceConfiguration('SourceIndex',2,...
        'IsInitializingCentralTracks',true,...
        'CentralToLocalTransformFcn',@central2local,...
        'LocalToCentralTransformFcn',@local2central);

    % Create a trackFuser object
    fuser = trackFuser(...
        'MaxNumSources', 2, ...
        'SourceConfigurations',{radarConfig;lidarConfig},...
        'StateTransitionFcn',@helperctcuboid,...
        'StateTransitionJacobianFcn',@helperctcuboidjac,...
        'ProcessNoise',diag([1 3 1]),...
        'HasAdditiveProcessNoise',false,...
        'AssignmentThreshold',[250 inf],...
        'ConfirmationThreshold',[3 5],...
        'DeletionThreshold',[5 5],...
```

```
        'StateFusion','Custom',...
        'CustomStateFusionFcn',@helperRadarLidarFusionFcn);
end

tracks = fuser(localTracks, time);
end
```

Homogeneous Source Configurations

In this example, you define the radar and lidar source configurations differently than in the original "Track-Level Fusion of Radar and Lidar Data" on page 5-536 example. In the original example, the `CentralToLocalTransformFcn` and `LocalToCentralTransformFcn` properties of the two source configurations are different because they use different function handles. This makes the source configurations a heterogeneous cell array. Such a definition is correct and valid when executing in MATLAB. However, in code generation, all source configurations must use the same function handles. To avoid the different function handles, you define one function to transform tracks from central (fuser) definition to local (source) definition and one function to transform from local to central. Each of these functions switches between the transform functions defined for the individual sources in the original example. Both functions are part of the heterogeneousInputsFuser function.

Here is the code for the `local2central` function, which uses the `SourceIndex` property to determine the correct function to use. Since the two types of local tracks transform to the same definition of central track, there is no need to predefine the central track.

```
function centralTrack = local2central(localTrack)
switch localTrack.SourceIndex
    case 1 % radar
        centralTrack = radar2central(localTrack);
    otherwise % lidar
        centralTrack = lidar2central(localTrack);
end
end
```

The function `central2local` transforms the central track into a radar track if `SourceIndex` is 1 or into a lidar track if `SourceIndex` is 2. Since the two tracks have a different definition of `State`, `StateCovariance`, and `TrackLogicState`, you must first predefine the output. Here is the code snippet for the function:

```
function localTrack = central2local(centralTrack)
state = 0;
stateCov = 1;
coder.varsize('state', [10, 1], [1 0]);
coder.varsize('stateCov', [10 10], [1 1]);
localTrack = objectTrack('State', state, 'StateCovariance', stateCov);

switch centralTrack.SourceIndex
    case 1
        localTrack = central2radar(centralTrack);
    case 2
        localTrack = central2lidar(centralTrack);
    otherwise
        % This branch is never reached but is necessary to force code
        % generation to use the predefined localTrack.
end
end
```

The functions `radar2central` and `central2radar` are the same as in the original example but moved from the live script to the heterogeneousInputsFuser function. You also add the `lidar2central` and `central2lidar` functions to the heterogeneousInputsFuser function. These two functions convert from the track definition that the fuser uses to the lidar track definition.

Run the Example in MATLAB

Before generating code, make sure that the example still runs after all the changes made to the fuser. The file `lidarRadarData.mat` contains the same scenario as in the original example. It also contains a set of radar and lidar tracks recorded at each step of that example. You also use a similar display to visualize the example and define the same `trackGOSPAMetric` objects to evaluate the tracking performance.

```matlab
% Load the scenario and recorded local tracks
load('lidarRadarData.mat','scenario','localTracksCollection')
display = helperTrackFusionCodegenDisplay('FollowActorID',3);
showLegend(display,scenario);

% Radar GOSPA
gospaRadar = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperRadarDistance,...
    'CutoffDistance',25);

% Lidar GOSPA
gospaLidar = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperLidarDistance,...
    'CutoffDistance',25);

% Central/Fused GOSPA
gospaCentral = trackGOSPAMetric('Distance','custom',...
    'DistanceFcn',@helperLidarDistance,... % State space is same as lidar
    'CutoffDistance',25);


gospa = zeros(3,0);
missedTargets = zeros(3,0);
falseTracks = zeros(3,0);
% Ground truth for metrics. This variable updates every time step
% automatically, because it is a handle to the actors.
groundTruth = scenario.Actors(2:end);

fuserStepped = false;
fusedTracks = objectTrack.empty;
idx = 1;
clear heterogeneousInputsFuser
while advance(scenario)
    time = scenario.SimulationTime;
    localTracks = localTracksCollection{idx};

    if ~isempty(localTracks) || fuserStepped
        fusedTracks = heterogeneousInputsFuser(localTracks,time);
        fuserStepped = true;
    end

    radarTracks = localTracks([localTracks.SourceIndex]==1);
    lidarTracks = localTracks([localTracks.SourceIndex]==2);
```

```
% Capture GOSPA and its components for all trackers
[gospa(1,idx),~,~,~,missedTargets(1,idx),falseTracks(1,idx)] = gospaRadar(radarTracks, ground
[gospa(2,idx),~,~,~,missedTargets(2,idx),falseTracks(2,idx)] = gospaLidar(lidarTracks, ground
[gospa(3,idx),~,~,~,missedTargets(3,idx),falseTracks(3,idx)] = gospaCentral(fusedTracks, grou

% Update the display
display(scenario,[],[], radarTracks,...
    [],[],[],[], lidarTracks, fusedTracks);

idx = idx + 1;
end
```
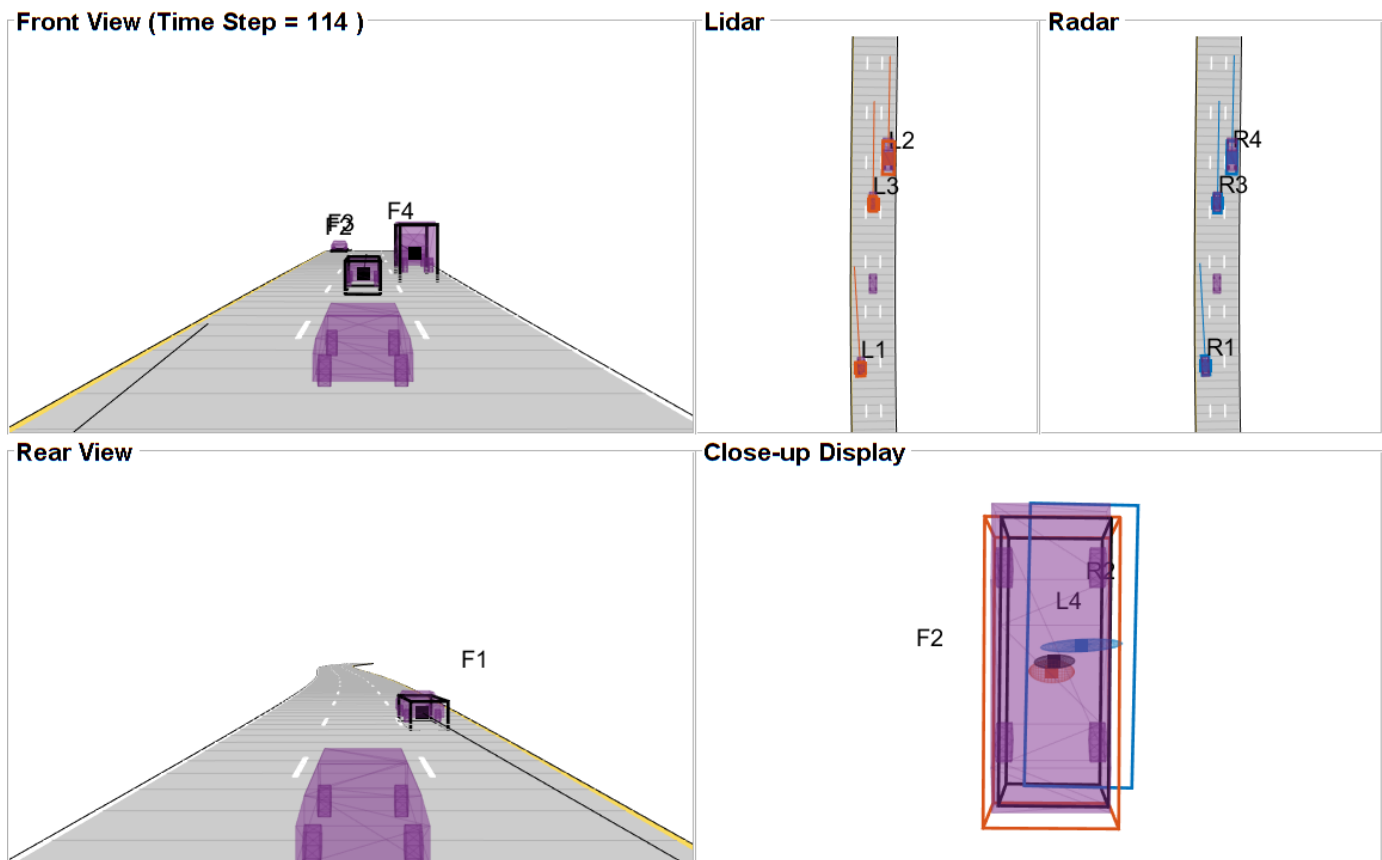

Generate Code for the Track Fuser

To generate code, you must define the input types for both the radar and lidar tracks and the timestamp. In both the original script and in the previous section, the radar and lidar tracks are defined as arrays of `objectTrack` objects. In code generation, the entry-level function cannot use an array of objects. Instead, you define an array of structures.

You use the struct `oneLocalTrack` to define the inputs coming from radar and lidar tracks. In code generation, the specific data types of each field in the struct must be defined exactly the same as the types defined for the corresponding properties in the recorded tracks. Furthermore, the size of each field must be defined correctly. You use the `coder.typeof` (MATLAB Coder) function to specify fields that have variable size: `State`, `StateCovariance`, and `TrackLogicState`. You define the `localTracks` input using the `oneLocalTrack` struct and the `coder.typeof` function, because the

number of input tracks varies from zero to eight in each step. You use the function `codegen` (MATLAB Coder) to generate the code.

Notes:

1   If the input tracks use different types for the `State` and `StateCovariance` properties, you must decide which type to use, double or single. In this example, all tracks use double precision and there is no need for this step.

2   If the input tracks use different definitions of `StateParameters`, you must first create a superset of all `StateParameters` and use that superset in the `StateParameters` field. A similar process must be done for the `ObjectAttributes` field. In this example, all tracks use the same definition of `StateParameters` and `ObjectAttributes`.

```
% Define the inputs to fuserHeterogeneousInputs for code generation
oneLocalTrack = struct(...
    'TrackID', uint32(0), ...
    'BranchID', uint32(0), ...
    'SourceIndex', uint32(0), ...
    'UpdateTime', double(0), ...
    'Age', uint32(0), ...
    'State', coder.typeof(1, [10 1], [1 0]), ...
    'StateCovariance', coder.typeof(1, [10 10], [1 1]), ...
    'StateParameters', struct, ...
    'ObjectClassID', double(0), ...
    'TrackLogic', 'History', ...
    'TrackLogicState', coder.typeof(false, [1 10], [0 1]), ...
    'IsConfirmed', false, ...
    'IsCoasted', false, ...
    'IsSelfReported', false, ...
    'ObjectAttributes', struct);

localTracks = coder.typeof(oneLocalTrack, [8 1], [1 0]);
fuserInputArguments = {localTracks, time};

codegen heterogeneousInputsFuser -args fuserInputArguments;
```

Run the Example with the Generated Code

You run the generated code like you ran the MATLAB code, but first you must reinitialize the scenario, the GOSPA objects, and the display.

You use the `toStruct` object function to convert the input tracks to arrays of structures.

Notes:

1   If the input tracks use different data types for the `State` and `StateCovariance` properties, make sure to cast the `State` and `StateCovariance` of all the tracks to the data type you chose when you defined the `oneLocalTrack` structure above.

2   If the input tracks required a superset structure for the fields `StateParameters` or `ObjectAttributes`, make sure to populate these structures correctly before calling the `mex` file.

You use the `gospaCG` variable to keep the GOSPA metrics for this run so that you can compare them to the GOSPA values from the MATLAB run.

```
% Rerun the scenario with the generated code
fuserStepped = false;
```

```
fusedTracks = objectTrack.empty;
gospaCG = zeros(3,0);
missedTargetsCG = zeros(3,0);
falseTracksCG = zeros(3,0);

idx = 1;
clear heterogeneousInputsFuser_mex
reset(display);
reset(gospaRadar);
reset(gospaLidar);
reset(gospaCentral);
restart(scenario);
while advance(scenario)
    time = scenario.SimulationTime;
    localTracks = localTracksCollection{idx};

    if ~isempty(localTracks) || fuserStepped
        fusedTracks = heterogeneousInputsFuser_mex(toStruct(localTracks),time);
        fuserStepped = true;
    end

    radarTracks = localTracks([localTracks.SourceIndex]==1);
    lidarTracks = localTracks([localTracks.SourceIndex]==2);

    % Capture GOSPA and its components for all trackers
    [gospaCG(1,idx),~,~,~,missedTargetsCG(1,idx),falseTracksCG(1,idx)] = gospaRadar(radarTracks,
    [gospaCG(2,idx),~,~,~,missedTargetsCG(2,idx),falseTracksCG(2,idx)] = gospaLidar(lidarTracks,
    [gospaCG(3,idx),~,~,~,missedTargetsCG(3,idx),falseTracksCG(3,idx)] = gospaCentral(fusedTracks

    % Update the display
    display(scenario,[],[], radarTracks,...
        [],[],[],[], lidarTracks, fusedTracks);

    idx = idx + 1;
end
```
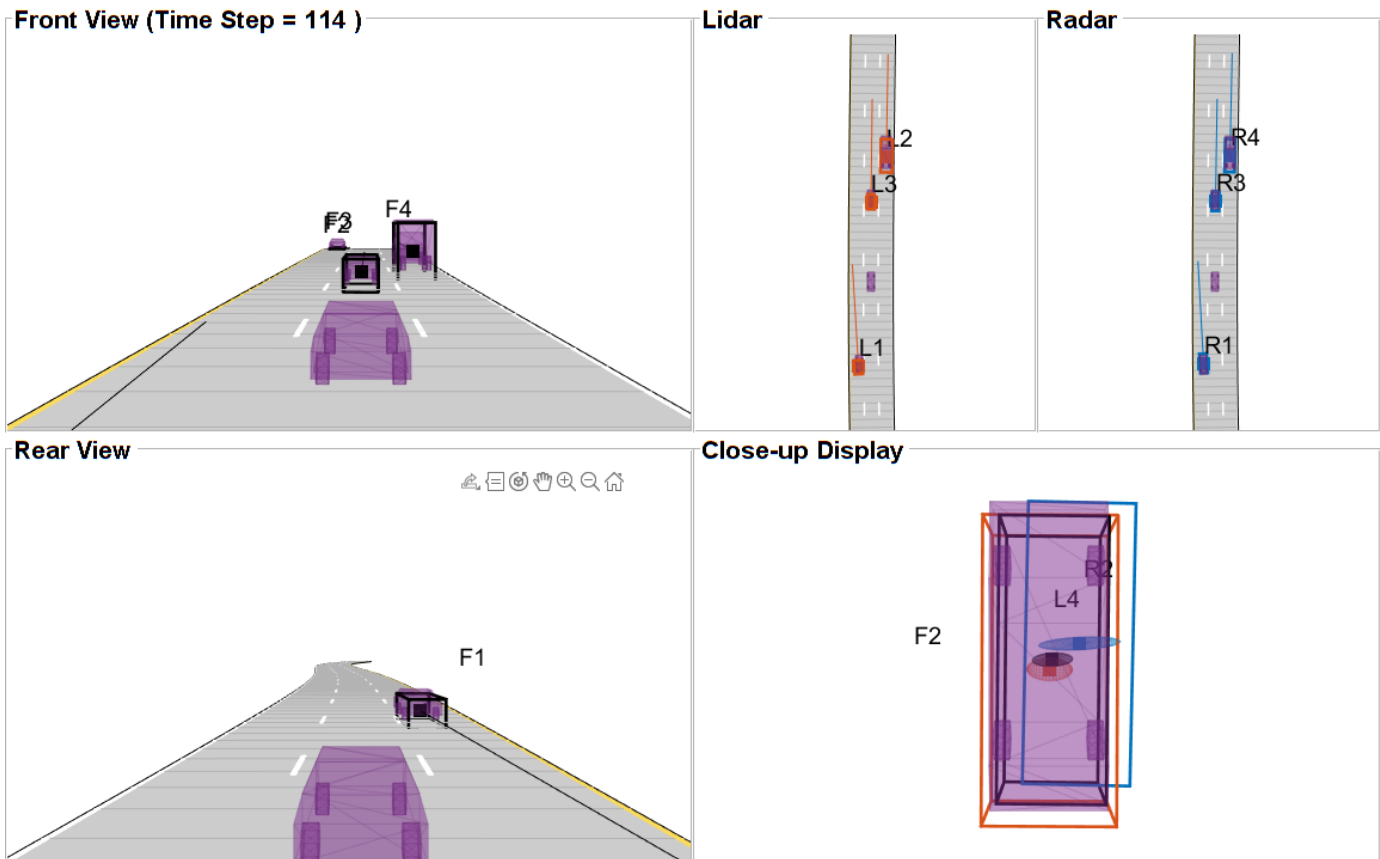
At the end of the run, you want to verify that the generated code provided the same results as the MATLAB code. Using the GOSPA metrics you collected in both runs, you can compare the results at the high level. Due to numerical roundoffs, there may be small differences in the results of the generated code relative to the MATLAB code. To compare the results, you use the absolute differences between GOSPA values and check if they are all smaller than 1e-10. The results show that the differences are very small.

```
% Compare the GOSPA values from MATLAB run and generated code
areGOSPAValuesEqual = all(abs(gospa-gospaCG)<1e-10,'all');
disp("Are GOSPA values equal up to the 10th decimal (true/false)? " + string(areGOSPAValuesEqual))
```

```
Are GOSPA values equal up to the 10th decimal (true/false)? true
```

Summary

In this example, you learned how to generate code for a track-level fusion algorithm when the input tracks are heterogeneous. You learned how to define the `trackFuser` and its `SourceConfigurations` property to support heterogeneous sources. You also learned how to define the input in compilation time and how to pass it to the mex file in runtime.

Supporting Functions

The following functions are used by the GOSPA metric.

helperLidarDistance

Function to calculate a normalized distance between the estimate of a track in radar state-space and the assigned ground truth.

```
function dist = helperLidarDistance(track, truth)

% Calculate the actual values of the states estimated by the tracker

% Center is different than origin and the trackers estimate the center
rOriginToCenter = -truth.OriginOffset(:) + [0;0;truth.Height/2];
rot = quaternion([truth.Yaw truth.Pitch truth.Roll],'eulerd','ZYX','frame');
actPos = truth.Position(:) + rotatepoint(rot,rOriginToCenter')';

% Actual speed and z-rate
actVel = [norm(truth.Velocity(1:2));truth.Velocity(3)];

% Actual yaw
actYaw = truth.Yaw;

% Actual dimensions.
actDim = [truth.Length;truth.Width;truth.Height];

% Actual yaw rate
actYawRate = truth.AngularVelocity(3);

% Calculate error in each estimate weighted by the "requirements" of the
% system. The distance specified using Mahalanobis distance in each aspect
% of the estimate, where covariance is defined by the "requirements". This
% helps to avoid skewed distances when tracks under/over report their
% uncertainty because of inaccuracies in state/measurement models.

% Positional error.
estPos = track.State([1 2 6]);
reqPosCov = 0.1*eye(3);
e = estPos - actPos;
d1 = sqrt(e'/reqPosCov*e);

% Velocity error
estVel = track.State([3 7]);
reqVelCov = 5*eye(2);
e = estVel - actVel;
d2 = sqrt(e'/reqVelCov*e);

% Yaw error
estYaw = track.State(4);
reqYawCov = 5;
e = estYaw - actYaw;
d3 = sqrt(e'/reqYawCov*e);

% Yaw-rate error
estYawRate = track.State(5);
reqYawRateCov = 1;
e = estYawRate - actYawRate;
d4 = sqrt(e'/reqYawRateCov*e);

% Dimension error
estDim = track.State([8 9 10]);
reqDimCov = eye(3);
e = estDim - actDim;
```

```matlab
d5 = sqrt(e'/reqDimCov*e);

% Total distance
dist = d1 + d2 + d3 + d4 + d5;
end
```

helperRadarDistance

Function to calculate a normalized distance between the estimate of a track in radar state-space and the assigned ground truth.

```matlab
function dist = helperRadarDistance(track, truth)
% Calculate the actual values of the states estimated by the tracker

% Center is different than origin and the trackers estimate the center
rOriginToCenter = -truth.OriginOffset(:) + [0;0;truth.Height/2];
rot = quaternion([truth.Yaw truth.Pitch truth.Roll],'eulerd','ZYX','frame');
actPos = truth.Position(:) + rotatepoint(rot,rOriginToCenter')';
actPos = actPos(1:2); % Only 2-D

% Actual speed
actVel = norm(truth.Velocity(1:2));

% Actual yaw
actYaw = truth.Yaw;

% Actual dimensions. Only 2-D for radar
actDim = [truth.Length;truth.Width];

% Actual yaw rate
actYawRate = truth.AngularVelocity(3);

% Calculate error in each estimate weighted by the "requirements" of the
% system. The distance specified using Mahalanobis distance in each aspect
% of the estimate, where covariance is defined by the "requirements". This
% helps to avoid skewed distances when tracks under/over report their
% uncertainty because of inaccuracies in state/measurement models.

% Positional error
estPos = track.State([1 2]);
reqPosCov = 0.1*eye(2);
e = estPos - actPos;
d1 = sqrt(e'/reqPosCov*e);

% Speed error
estVel = track.State(3);
reqVelCov = 5;
e = estVel - actVel;
d2 = sqrt(e'/reqVelCov*e);

% Yaw error
estYaw = track.State(4);
reqYawCov = 5;
e = estYaw - actYaw;
d3 = sqrt(e'/reqYawCov*e);

% Yaw-rate error
```

```matlab
estYawRate = track.State(5);
reqYawRateCov = 1;
e = estYawRate - actYawRate;
d4 = sqrt(e'/reqYawRateCov*e);

% Dimension error
estDim = track.State([6 7]);
reqDimCov = eye(2);
e = estDim - actDim;
d5 = sqrt(e'/reqDimCov*e);

% Total distance
dist = d1 + d2 + d3 + d4 + d5;

% A constant penalty for not measuring 3-D state
dist = dist + 3;
end
```

Extended Object Tracking with Lidar for Airport Ground Surveillance

An apron is a defined area at the airport intended to accommodate aircraft for purposes of loading or unloading passengers, mail or cargo, fueling, parking or maintenance [1]. Airport aprons are usually highly dynamic and heterogeneous environments where apron personnel and vehicles operate in close proximity to each other. Due to such nature of the aprons, it presents a higher risk for ground handling accidents involving aircraft as well as ground personnel. Lidar-based surveillance systems at aprons have been proposed as an effective method to improve the situation picture and to serve as a measure to mitigate high risk at the aprons [2].

This example shows you how to simulate Lidar data for an apron traffic scene and track ground traffic using a GGIW-PHD (Gamma Gaussian Inverse Wishart PHD) extended object tracker.

Setup Scenario

In this example, you simulate a scenario where an aircraft is entering into the gate area. The aircraft is guided into its parking spot by three marshallers, one on each side of the aircraft and one in front of it. You simulate ground traffic near the parking spot of the aircraft. After the aircraft is parked, the ground traffic as well as the marshallers start moving towards the aircraft. In addition to the aircraft entering into the gate area, you also simulate two aircraft already parked at the gate. The scenario used in this example was created by using the Tracking Scenario Designer and exported to a MATLAB® function to connect it with downstream functionalities. The exported function was modified in MATLAB to specify the `Mesh` property for each platform and to complete trajectories to the scenario end time. The `Mesh` property of the `Platform` allows you to define a geometry of a platform for lidar simulation. In this example, you specify the geometry for aircraft as `tracking.scenario.airplaneMesh`. The other objects in the scene are represented using cuboids. For more details on the how to create the scenario, refer to the `createScenario` function at the end of this example.

```
scenario = createScenario;
```

To perceive the scene, you use a 360-degree field of view lidar sensor using the `monostaticLidarSensor` System object™. The sensor has 64 elevation channels and is mounted at the terminal at a height of 8 meters from the ground. The azimuth resolution of the sensor is defined as 0.32 degrees. Under this configuration, the sensor produces a total of 72,000 points per scan. This high resolution set of data points is commonly termed as the point cloud. The data from the sensor is simulated at 5 Hz by specifying the `UpdateRate` of the sensor.

```
% Create a lidar sensor
lidar = monostaticLidarSensor( ...
    'SensorIndex',1,...
    'UpdateRate',5,...
    'HasINS',true,...
    'AzimuthLimits',[-180 180],...
    'ElevationLimits',[-10 27],...
    'AzimuthResolution',0.32,...
    'ElevationResolution',0.5781,...
    'DetectionCoordinates','scenario',...
    'MountingLocation',[50 -15 -8],...
    'MountingAngles',[-90 0 0],...
    'MaxRange',150,...
    'RangeAccuracy',0.002);
```

```
% Get access to terminal platform
terminal = scenario.Platforms{1};

% Mount the lidar on the platform
terminal.Sensors = lidar;
```

You can observe the scene and simulated lidar data from birds-eye view as well as from the roof of the terminal in the animation below. The point cloud is denoted by colored points, where color changes from blue to yellow with height. Notice that the lidar returns contain reflections from environment such as the ground.
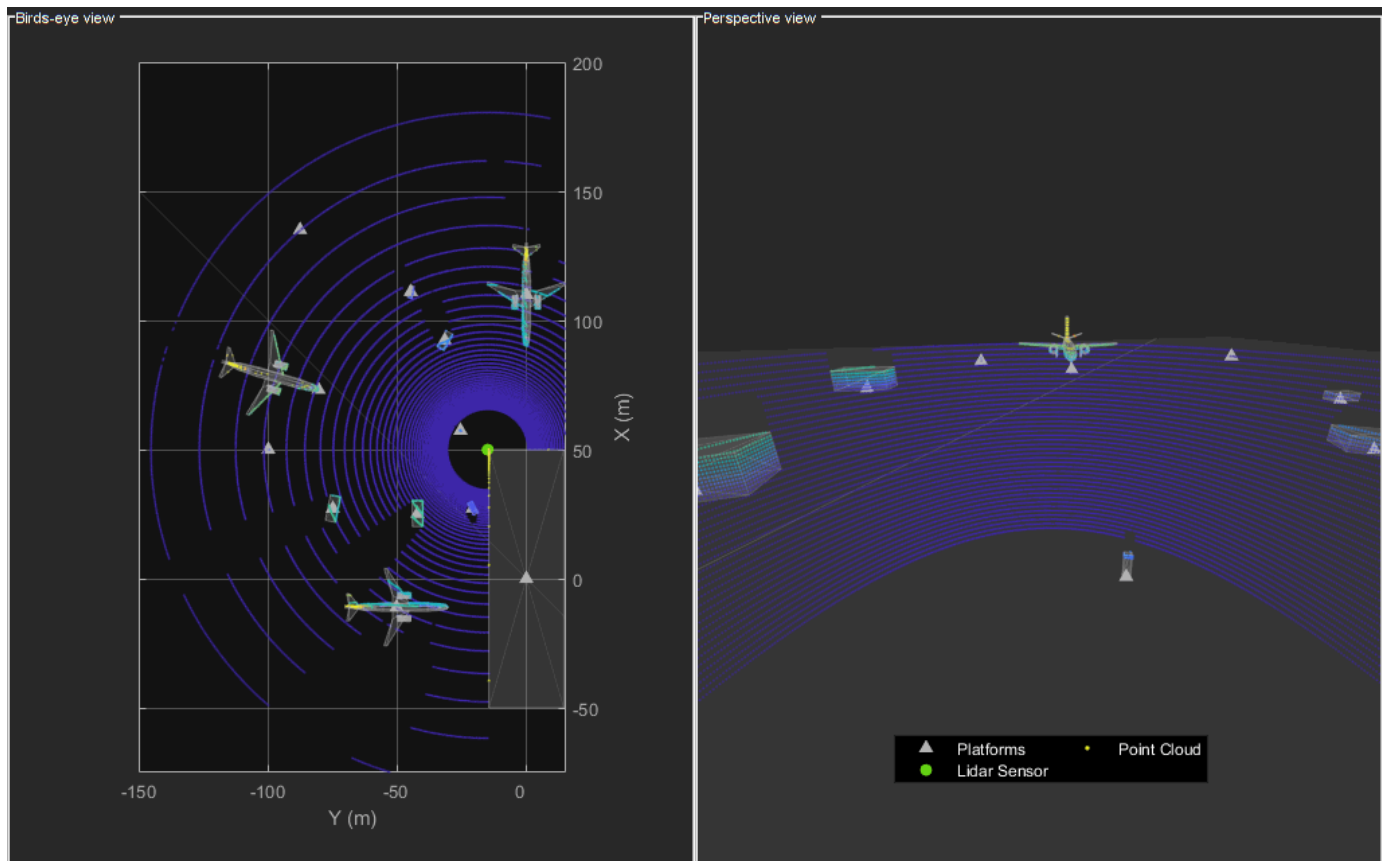
You can also observe in the animation that the lidar sensor returns multiple measurements per object. Conventional multi-object trackers such as `trackerGNN` (GNN) tracker, `trackerJPDA` (JPDA) tracker assumes that the each sensor reports one measurement per object. To use conventional trackers for tracking objects using lidar, point cloud from potential objects is typically preprocessed and clustered into new type of measurements. These new measurements typically define positional as well as dimensional aspects of the object, for example, a bounding box measurement. For a complete workflow on using conventional trackers with lidar data, refer to the "Track Vehicles Using Lidar: From Point Cloud to Track List" on page 5-369 example. This workflow of segmenting lidar data and clustering into bounding boxes is prone to clustering imperfections, when objects get too close. This is especially true in the case of apron environments. An alternative way to track objects using lidar data is to use extended object trackers. In contrast to conventional trackers, extended object trackers are designed to track objects which produce more than one measurement per sensor.

Setup Extended Object Tracker and Performance Metrics

Extended Object Tracker

In this example, you use a GGIW (Gamma Gaussian Inverse Wishart) implementation of a probability hypothesis density (PHD) tracker. This GGIW model uses three distributions to specify the target model: A Gaussian distribution to describe the kinematics of the target's motion center such as its position and velocity; An Inverse Wishart (IW) distribution to describe an ellipsoidal extent of the target; A Gamma distribution to describe the expected number of measurements from the target. For more details on the GGIW-PHD filter, refer to [3].

An extended object PHD tracker uses a partitioning algorithm to process the input measurement set. A partitioning algorithm is responsible for specifying multiple possible segmentation hypothesis for sensor measurements. As the number of total hypothesis for segmentation is typically very large, approximation techniques like Distance Partitioning (see `partitionDetections`), Prediction Partitioning and Expectation Maximization (EM) are used [3]. The distance-partitioning algorithm works similar to distance-based clustering techniques with the difference that it produces multiple possible partitions. The prediction-partitioning and EM algorithm uses predictions from the tracker about the objects to assist the partitioning of the measurements into multiple possible clusters. This technique to use predictions from the tracker is essential when objects are spatially close to each other such as in an apron environment. In this example, you use a helper class `helperPartitioningAlgorithm` to partition the measurements set using predictions from the tracker. You obtain these predictions by utilizing the `predictTracksToTime` function of `trackerPHD`.

```
% A handle object to pass predicted tracks into partitioning algorithm
partitioner = helperPartitioningAlgorithm;

% A function which takes detections as inputs and return partitions
partitioningFcn = @(detections)partitionDetections(partitioner,detections);
```

To setup a PHD tracker, you first define the configuration of the sensor using a `trackingSensorConfiguration` object. You use the properties of the simulated lidar sensor to specify properties of the configuration. You also define a function initFilter on page 5-0 , which initializes a constant-velocity `ggiwphd` filter. This function wraps around `initcvggiwphd` and increases the certainty in Gamma distribution and Inverse Wishart distribution of the target. It also configures the filter to work a large number of detections by specifying the `MaxNumDetections` property of ggiwphd filter.

```
% Sensor's field of view defined in same order as returned
% by sensor's transform function
sensorLimits = [lidar.AzimuthLimits;lidar.ElevationLimits;0 lidar.MaxRange];

% Sensor's resolution defined in same order as limits
sensorResolution = [lidar.AzimuthResolution;lidar.ElevationResolution;lidar.MaxRange];

% Parameters to transform state of the track by transform function.
trackToSensorTransform = struct('Frame','spherical',...
    'OriginPosition',lidar.MountingLocation(:),...
    'Orientation',rotmat(quaternion(lidar.MountingAngles,'eulerd','ZYX','frame'),'frame'),...
    'IsParentToChild',true,...
    'HasVelocity',false);

% A function to initiate a PHD filter by this sensor
filterInitFcn = @initFilter;
```

```
config = trackingSensorConfiguration(1,...
    'IsValidTime',true,...% update with sensor on each step call
    'FilterInitializationFcn',filterInitFcn,...% Function to initialize a PHD filter
    'SensorTransformFcn',@cvmeas,...% Transformation function from state to az,el,r
    'SensorTransformParameters',trackToSensorTransform,...% Parameters for transform function
    'SensorLimits',sensorLimits,...
    'SensorResolution',sensorResolution,...
    'DetectionProbability',0.95... % Probability of detecting the target
    );
```

Next, you assemble this information and create an extended object PHD tracker using the `trackerPHD` System object™.

```
tracker = trackerPHD('SensorConfigurations',config,...
    'PartitioningFcn',partitioningFcn,...
    'AssignmentThreshold',30,...% -log-likelihood beyond which a measurment cell initializes new
    'ConfirmationThreshold',0.9,...% Threshold to call a component as confirmed track
    'ExtractionThreshold',0.75,...% Threshold to call a component as track
    'MergingThreshold',25,...% Threshold to merge components with same Label
    'LabelingThresholds',[1.1 0.1 0.05]); % Prevents track-splitting
```

Metrics

Next, you set up a GOSPA metric calculator using the `trackGOSPAMetric` class to evaluate the performance of the tracker. GOSPA metric aims to analyze the performance of a tracker by providing a single cost value. A lower value of the cost represents better tracking performance. To use GOSPA metric, a distance function is defined between a track and a truth. This distance function calculates the cost to assign a track and truth to each other. It is also used to represent the localization or track-level accuracy of the estimate. The distance between a track and truth in this example is defined using a `'Custom'` distance function trackTruthDistance on page 5-0 , included in the Supporting Functions on page 5-0 below. As the GGIW-PHD tracker estimates the geometric center, the custom distance function uses the position of the platform's geometric center to define the positional error.

```
gospaObj = trackGOSPAMetric('Distance','custom','DistanceFcn',@trackTruthDistance);
```

You use a helper class `helperAirportTrackingDisplay` to visualize the ground truth, simulated data and estimated tracks.

```
display = helperAirportTrackingDisplay;
```

Run Scenario and Tracker

Next, you run the scenario, simulate returns from the lidar sensor and process them using the tracker. You also calculate the GOSPA metric for the tracks using the available ground truth. This example assumes that preprocessing of lidar data to remove lidar returns from environment such as ground and terminal is already performed. Therefore, returns from the environment are removed using the available ground truth information from simulation about lidar returns.

```
% Initialize tracks
confTracks = struct.empty(0,1);

% Ground truth for metric
platforms = scenario.Platforms;
trackablePlatforms = platforms(cellfun(@(x)x.ClassID,platforms) > 0);

% Initialize metric values
```

```
gospaMetrics = zeros(0,4);

% Number of tracks recording for MATLAB vs MEX
numTrackML = zeros(0,1);

% Advance scenario
while advance(scenario)
    time = scenario.SimulationTime;

    % Call lidarDetect on terminal to simulate point cloud returns.
    [ptCloud, ~, groundTruthIdx] = lidarDetect(terminal,time);

    % Pack returns as objectDetection. Included in Supporting Functions
    detections = packAsObjectDetection(ptCloud, groundTruthIdx, time);

    % Provide predicted tracks for prediction partitioning
    if isLocked(tracker)
        predictedTracks = predictTracksToTime(tracker,'all',time);
        partitioner.PredictedTrackList = predictedTracks;
    end

    % Call tracker step using detections and time
    confTracks = tracker(detections, time);

    % Calculate OSPA metric
    truthStruct = cellfun(@(x)x.pose,trackablePlatforms);
    [~,gospa,~,loc,missT,falseT] = gospaObj(confTracks, trackablePlatforms);
    gospaMetrics(end+1,:) = [gospa loc missT falseT]; %#ok<SAGROW>

    % Update the display
    display(scenario, ptCloud, confTracks);
end
```

Results

Next, you evaluate the performance of each tracker using visualization as well as quantitative metrics. In the animation below, the tracks are represented by blue squares annotated by their identities (TrackID). The ellipsoid around the object describes the extent estimated by the GGIW-PHD filter.
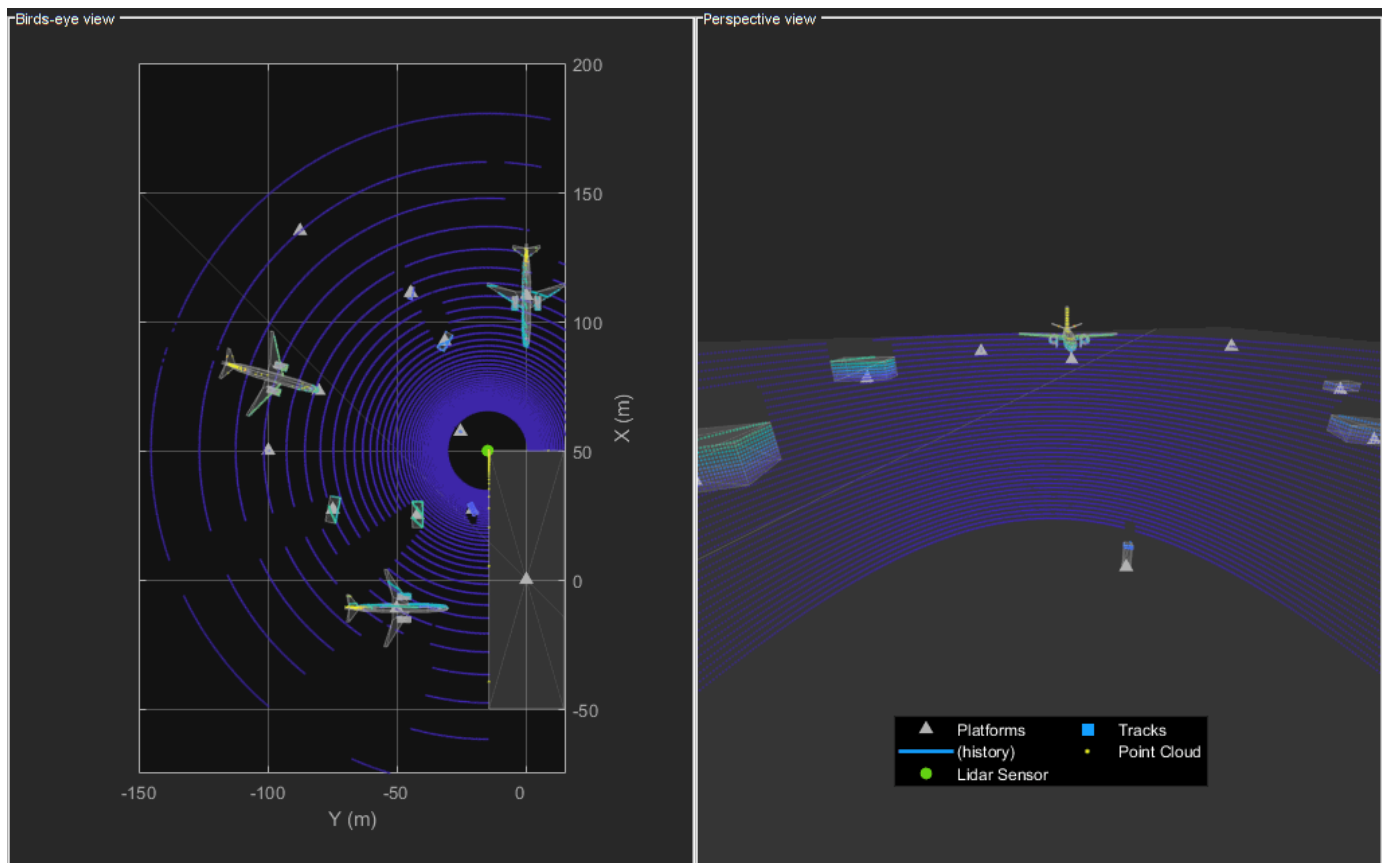
Track Maintenance

Till the aircraft is parked, the tracker is able to track all objects, except the marshaller on the right of the aircraft. This is because the marshaller remained undetected by the sensor at its initial position. After the aircraft is parked, the tracker is able to react to the motion of the objects to estimate their position as well as velocity. The tracker is able to maintain a track on all objects up to about 34 seconds. After that, the marshaller on the left gets occluded behind the aircraft jet engines and its track is dropped by the tracker due to multiple misses. As the ground traffic approaches the aircraft, the tracker is able to track them as separate objects and did not confuse their measurements with one other. This was possible because the predictions of the tracker about objects assisted in partitioning the measurement set properly.

You can also quantitatively analyze this performance of the tracker using the GOSPA metric and its components. Notice that the false track component of the metric remains zero, which indicates that no false tracks were confirmed. The non-zero missed target component represents the undetected marshaller. Notice that as soon as the tracker establishes a track on that object, the component drops down to zero. The missed target component rises to a non-zero value again, which represents the track drop due to object occlusion.

```
figure;
order = ["Total GOSPA","Localization","Missed Target","False Tracks"];
for i = 1:4
    ax = subplot(2,2,i);
    plot(gospaMetrics(:,i),'LineWidth',2);
    title(order(i));
```

```
    xlabel('Time'); ylabel('Metric');
    grid on;
end
```
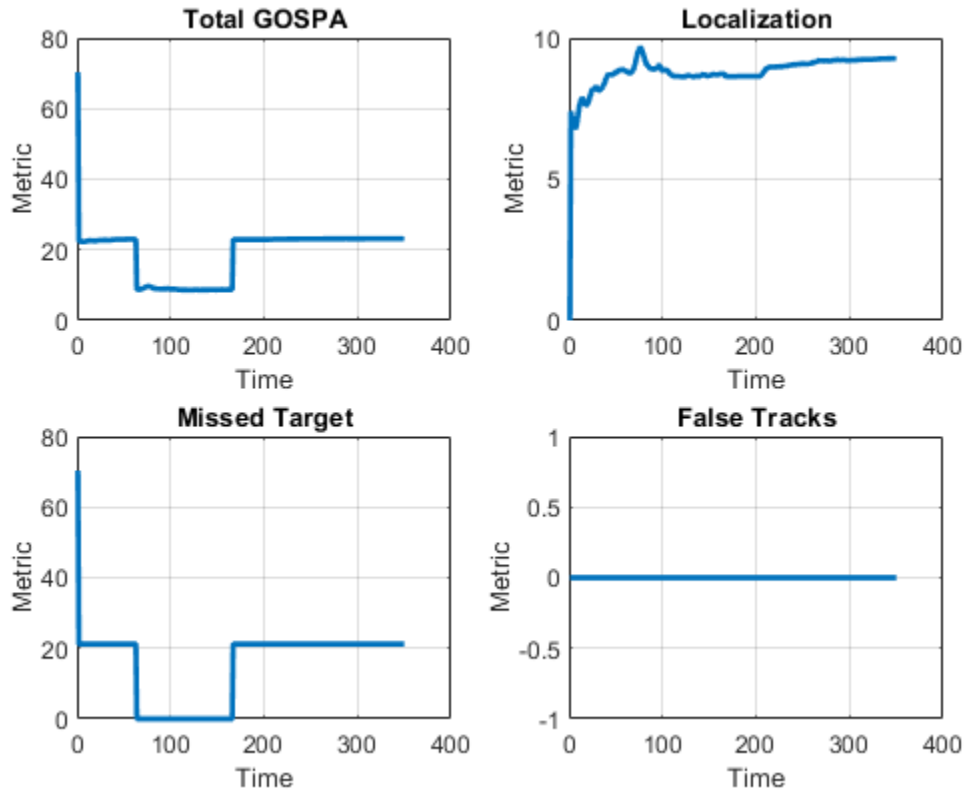

Track-level accuracy

Track-level accuracy refers to the accuracy in estimating the state of each object such as its position, velocity and dimensions. You can observe that the that tracks lie on the ground truth of the objects. When the object starts moving, a velocity vector is plotted, representing the direction and magnitude the velocity. Notice that the estimated velocity is in the same direction as the object motion. The GGIW-PHD filter assumes that the measurements from the target are distributed in the target's estimated ellipsoidal extent. This results in the filter estimating the size of the object using observable or non-occluded regions of the objects. Notice that the tracks on the parked ground vehicles have an offset in their position towards the observable sides.

The track-level inaccuracies can also be quantitatively estimated using the localization component of the GOSPA metric. The total GOSPA metric is a combined metric of all its components and hence captures the effect of localization accuracies, missed targets, and false tracks.

Summary

In this example, you learned how to simulate lidar data for an apron scenario and process it with an extended object GGIW-PHD tracker. You also learned how to evaluate the performance of the tracker using the GOSPA metric and its associated components.

Supporting Functions

packAsObjectDetection

```
function detections = packAsObjectDetection(ptCloud, groundTruthIdx, time)
% Get x,y,z locations from the cell array
loc = ptCloud{1}';
% Points which are nan reflect those rays which did not intersect with any
% object
nanPoints = any(isnan(loc),1);

% groundTruthIdx{1} represents the ground truth of the points. The first
% column represents PlatformID and second column represents ClassID.
% ClassID for environment is specified as 0.
envPoints = groundTruthIdx{1}(:,2) == 0;

% nanPoints or envPoints are remove
toRemove = nanPoints | envPoints';

% Keep valid return
loc = loc(:,~toRemove);

% Pack each return as objectDetection.
n = sum(~toRemove);

% Allocate memory
detections = repmat({objectDetection(time,[0;0;0],'MeasurementNoise',1e-2*eye(3))},n,1);

% Fill measurements
for i = 1:n
    detections{i}.Measurement = loc(:,i);
end

end
```

createScenario

```
function scenario = createScenario

% Create Scenario
scenario = trackingScenario;
scenario.UpdateRate = 0;

planeMesh = tracking.scenario.airplaneMesh;

% Create platforms
Terminal = platform(scenario,'ClassID',0);
Terminal.Dimensions = struct( ...
    'Length', 100, ...
    'Width', 29, ...
    'Height', 20, ...
    'OriginOffset', [0 0 10]);

Aircraft = platform(scenario,'ClassID',1,'Mesh',planeMesh);
Aircraft.Dimensions = struct( ...
    'Length', 39, ...
    'Width', 34, ...
    'Height', 13.4, ...
```

```
    'OriginOffset', [39/2 0 13.4/2]);
Aircraft.Signatures{1} = ...
    rcsSignature(...
    'Pattern', [20 20;20 20], ...
    'Azimuth', [-180 180], ...
    'Elevation', [-90;90], ...
    'Frequency', [0 1e+20]);

Aircraft.Trajectory = waypointTrajectory(...
    [73.2829 -80.5669 0;60.7973 -37.3337 0],...
    [0;11.25],...
    'GroundSpeed',[8;0],...
    'Course',[106.1085;106.1085],...
    'Orientation',quaternion([106.1085 0 0;106.1085 0 0],'eulerd','ZYX','frame'));

delayTrajectory(Aircraft,0.1,70);

Ground = platform(scenario,'ClassID',0);
Ground.Trajectory.Position = [0 0 0];
Ground.Dimensions = struct('Length',500,...
    'Width',500,...
    'Height',1,...
    'OriginOffset',[0 0 -0.5]);

Plane = platform(scenario,'ClassID',1,'Mesh',planeMesh);
Plane.Dimensions = struct( ...
    'Length', 40, ...
    'Width', 30, ...
    'Height', 10, ...
    'OriginOffset', [0 0 5]);
Plane.Signatures{1} = ...
    rcsSignature(...
    'Pattern', [20 20;20 20], ...
    'Azimuth', [-180 180], ...
    'Elevation', [-90;90], ...
    'Frequency', [0 1e+20]);
Plane.Trajectory.Position = [-11 -50.3 0];
Plane.Trajectory.Orientation = quaternion([90 0 0], 'eulerd','zyx','frame');

Marshall = platform(scenario,'ClassID',6);
Marshall.Dimensions = struct( ...
    'Length', 0.24, ...
    'Width', 0.45, ...
    'Height', 1.7, ...
    'OriginOffset', [0 0 0.85]);

s = [57.3868 -25.5244 0;59.41 -32.53 0];
s2 = [57.3868 -25.5244 0;60.5429 -36.4531 0];

Marshall.Trajectory = waypointTrajectory( ...
    s2, ...
    [0;8], ...
    'Course', [-75;-75], ...
    'GroundSpeed', [0;0], ...
    'ClimbRate', [0;0], ...
    'Orientation', quaternion([0.793353340291235 0 0 -0.608761429008721;0.793353340291235 0 0 -0

delayTrajectory(Marshall, 12, 70);
```

```
Marshall1 = platform(scenario,'ClassID',6);
Marshall1.Dimensions = struct( ...
    'Length', 0.24, ...
    'Width', 0.45, ...
    'Height', 1.7, ...
    'OriginOffset', [0 0 0.85]);
Marshall1.Trajectory = waypointTrajectory( ...
    [50 -100 0;56.5 -89.3 0;56.8 -73.6 0;60.9 -59.5 0], ...
    [0;6.28880678506828;14.2336975651715;38.5523666710762], ...
    'Course', [52.5692412453125;71.0447132029417;101.247131543705;107.567935050594], ...
    'GroundSpeed', [2;2;2;0], ...
    'ClimbRate', [0;0;0;0], ...
    'Orientation', quaternion([0.896605327597113 0 0 0.442830539286162;0.813888868238491 0 0 0.58
delayTrajectory(Marshall1, 12, 70);

Marshall2 = platform(scenario,'ClassID',6);
Marshall2.Dimensions = struct( ...
    'Length', 0.24, ...
    'Width', 0.45, ...
    'Height', 1.7, ...
    'OriginOffset', [0 0 0.85]);
Marshall2.Trajectory = waypointTrajectory( ...
    [135.1 -87.7 0;118.3 -87.7 0;112.2 -73.3 0;80.8 -47.6 0], ...
    [0;10.3490919743252;18.1872070129823;42], ...
    'Course', [155.957629971433;124.12848387907;114.734795167842;153.606988602699], ...
    'GroundSpeed', [2;2;2;0], ...
    'ClimbRate', [0;0;0;0], ...
    'Orientation', quaternion([0.896605327597113 0 0 0.442830539286162;0.813888868238491 0 0 0.58
delayTrajectory(Marshall2, 12, 70);

Loader = platform(scenario,'ClassID',2);
Loader.Dimensions = struct( ...
    'Length', 10, ...
    'Width', 4, ...
    'Height', 4, ...
    'OriginOffset', [0 0 2]);
Loader.Trajectory = waypointTrajectory( ...
    [25.2 -42.2 0;39.4500 -42.3000 0;53.7 -42.4 0], ...
    [0;5;10], ...
    'Course',[-0.4021;-0.4021;-0.4021],...
    'Orientation',[quaternion([-0.4021 0 0],'eulerd','ZYX','frame');quaternion([-0.4021 0 0],'eu
    'GroundSpeed',[0;5.8;0],...
    'ClimbRate',[0;0;0],...
    'AutoPitch', false, ...
    'AutoBank', false);
delayTrajectory(Loader, 12, 70);

Loader2 = platform(scenario,'ClassID',2);
Loader2.Dimensions = struct( ...
    'Length', 10, ...
    'Width', 4, ...
    'Height', 4, ...
    'OriginOffset', [0 0 2]);
Loader2.Trajectory = waypointTrajectory( ...
    [27.142823040363762 -75 0;42.5614 -71.9000 0;57.98 -68.8 0], ...
    [0;5;10], ...
    'Course', [11.368107148451321;11.368107148451321;11.368107148451321], ...
```

```
        'Orientation',quaternion([11.368107148451321 0 0;11.368107148451321 0 0;11.368107148451321 0
        'GroundSpeed', [0;5.824096001626275;0], ...
        'ClimbRate', [0;0;0], ...
        'AutoPitch', false, ...
        'AutoBank', false);
delayTrajectory(Loader2, 50, 70);

Power = platform(scenario,'ClassID',2);
Power.Dimensions = struct( ...
        'Length', 5, ...
        'Width', 2, ...
        'Height', 1, ...
        'OriginOffset', [0 0 0.5]);
Power.Trajectory = waypointTrajectory( ...
        [27.2312963703295 -20.7687036296705 0;40.1656 -27.6344 0;53.1 -34.5 0], ...
        [0;5;10], ...
        'Course', [-27.9596882700088;-27.9596882700088;-27.9596882700088], ...
        'Orientation',[quaternion([-27.9596882700088 0 0],'eulerd','ZYX','frame');quaternion([-27.959
        'GroundSpeed', [0;5.8574;0], ...
        'ClimbRate', [0;0;0], ...
        'AutoPitch', false, ...
        'AutoBank', false);
delayTrajectory(Power, 20, 70);

Refueller = platform(scenario,'ClassID',2);
Refueller.Dimensions = struct( ...
        'Length', 7, ...
        'Width', 3, ...
        'Height', 2, ...
        'OriginOffset', [0 0 1]);
Refueller.Trajectory = waypointTrajectory( ...
        [92.3 -31.6 0;83.3 -36.7 0;74.3 -41.8 0], ...
        [0;5;10], ...
        'Course', [-150.4612;-150.4612;-150.4612], ...
        'Orientation',[quaternion([-150.4612 0 0],'eulerd','ZYX','frame');quaternion([-150.4612 0 0]
        'GroundSpeed', [0;4.137825515896000;0], ...
        'ClimbRate', [0;0;0], ...
        'AutoPitch', false, ...
        'AutoBank', false);
delayTrajectory(Refueller, 20, 70);

Car = platform(scenario,'ClassID',2);
Car.Dimensions = struct( ...
        'Length', 4.7, ...
        'Width', 1.8, ...
        'Height', 1.4, ...
        'OriginOffset', [0 0 0.7]);
Car.Trajectory = waypointTrajectory( ...
        [111.1 -44.8 0;93.8100 -48.1725 0;76.5200 -51.5450 0], ...
        [0;7.046336423986581;14.092672847973162], ...
        'Course', [-169.250904745059;-169.250904745059;-169.250904745059], ...
        'Orientation',[quaternion([-169.250904745059 0 0],'eulerd','ZYX','frame');quaternion([-169.25
        'GroundSpeed', [0;5;0], ...
        'ClimbRate', [0;0;0], ...
        'AutoPitch', false, ...
        'AutoBank', false);
delayTrajectory(Car, 20, 70);
```

```matlab
Plane1 = platform(scenario,'ClassID',1,'Mesh',planeMesh);
Plane1.Dimensions = struct( ...
    'Length', 40, ...
    'Width', 30, ...
    'Height', 10, ...
    'OriginOffset', [0 0 5]);
Plane1.Signatures{1} = ...
    rcsSignature(...
    'Pattern', [20 20;20 20], ...
    'Azimuth', [-180 180], ...
    'Elevation', [-90;90], ...
    'Frequency', [0 1e+20]);
Plane1.Trajectory.Position = [110 0 0];
Plane1.Trajectory.Orientation = quaternion([180 0 0], 'eulerd','zyx','frame');

    function delayTrajectory(plat, tOffset, tEnd)
        traj = plat.Trajectory;
        wp = traj.Waypoints;
        toa = traj.TimeOfArrival;
        g = traj.GroundSpeed;
        c = traj.Course;
        cr = traj.ClimbRate;
        o = traj.Orientation;

        tEnd = max(toa(end),tEnd);

        wp = [repmat(wp(1,:),1,1);wp;repmat(wp(end,:),1,1)];
        toa = [0;toa + tOffset;tEnd];
        g = [0;g;0];
        c = [c(1);c;c(end)];
        cr = [0;cr;0];

        o = [repmat(o(1),1,1);o;repmat(o(end),1,1)];

        newTraj = waypointTrajectory(wp,toa,...;
            'Course',c,...
            'GroundSpeed',g,...
            'ClimbRate',cr,...
            'Orientation',o);
        plat.Trajectory = newTraj;
    end
end
```

initFilter

```matlab
function filter = initFilter (varargin)
% This function interacts with the tracker in two signatures
% filter = initFilter() is called during prediction stages to add
% components to PHD.
%
% filter = initFilter(detections) is called during correction stages to add
% components to the PHD from detection sets which have a low-likelihood
% against existing tracks. All detections in the input have the assumption
% to belong to the same target.

% Adds zero components during prediction stages
filter = initcvggiwphd(varargin{:});
```

```
% If called with a detection
if nargin == 1
    % Get expected size
    expSize = filter.ScaleMatrices/(filter.DegreesOfFreedom - 4);

    % Higher the dof, higher the certainty in dimensions
    dof = 50;
    filter.DegreesOfFreedom = dof;
    filter.ScaleMatrices = (dof-4)*expSize;

    % Get expected number of detections
    expNumDets = filter.Shapes/filter.Rates;

    % shape/rate^2 = uncertainty;
    uncertainty = (expNumDets/4)^2;
    filter.Shapes = expNumDets^2/uncertainty;
    filter.Rates = expNumDets/uncertainty;
end

% GammaForgettingFactors acts like process noise
% for measurement rate
filter.GammaForgettingFactors(:) = 1.03;

% Temporal Decay acts like process noise for
% dimensions. Lower the decay, higher the variance increase
filter.TemporalDecay = 500;

% Specify MaxNumDetections
filter.MaxNumDetections = 10000;

end
```

trackTruthDistance

```
function dist = trackTruthDistance(track, truth)
% The tracks estimate the center of the object. Compute the origin position
rOriginToCenter = -truth{1}.Dimensions.OriginOffset(:);
rot = quaternion([truth{1}.Orientation],'eulerd','ZYX','frame');
actPos = truth{1}.Position(:) + rotatepoint(rot,rOriginToCenter')';

% Estimated track position
estPos = track.State(1:2:end);

% Error distance is between position.
dist = norm(actPos(:) - estPos);
end
```

References

[1] Weber, Ludwig. *International Civil Aviation Organization*. Kluwer Law International BV, 2017.

[2] Mund, Johannes, Lothar Meyer, and Hartmut Fricke. "LiDAR Performance Requirements and Optimized Sensor Positioning for Point Cloud-based Risk Mitigation at Airport Aprons." *Proceedings of the 6th International Conference on Research in Air Transportation*. 2014.

[3] Granström, Karl, et al. "On extended target tracking using PHD filters." (2012).

Track Multiple Lane Boundaries with a Global Nearest Neighbor Tracker

This example shows how to design and test a multiple lane tracking algorithm. The algorithm is tested in a driving scenario with probabilistic lane detections.

Introduction

An automated lane change maneuver (LCM) system enables the ego vehicle to automatically move from one lane to another lane. To successfully change lanes, the system requires localization of the ego vehicle with respect to stationary features, such as lane markings. A lane detection algorithm typically provides offset and curvature information about the current and adjacent lane boundaries. During the lane change, a discontinuity in the lateral offset is introduced into the lane detections, because the lateral offset is always with respect to the current lane in which the vehicle is traveling. The discontinuity and the jump in offset values may cause the LCM system to become unstable. One technique to compensate for this discontinuity issue is to use a multi-lane tracker.

Detect Lanes in a Lane Change Scenario

You load a `drivingScenario` (Automated Driving Toolbox) object, `scenario`, that contains an ego vehicle and its sensors from the `LaneTrackingScenario.mat` file. You use a `visionDetectionGenerator` (Automated Driving Toolbox) object to detect lanes in the scenario.

```
load('LaneTrackingScenario.mat','scenario','egoVehicle','sensors');
laneDetector = sensors{1};
```

To visualize the scenario in a Driving Scenario Designer, use:

```
drivingScenarioDesigner(scenario)
```

In this scenario, the ego vehicle is driving along a curved road with multiple lanes. The ego vehicle is equipped with a lane detector that detects lane boundaries and reports two lane boundaries on each side of the ego vehicle. To pass a slower moving vehicle traveling in the ego lane, the ego vehicle changes lanes from its original lane to the one on its left. The measurement reported by the lane detector contains the offset, the heading, and the curvature of the lane.

The following block of code runs the scenario and display the results of the lane detections.

```
% Setup plotting area
egoCarBEP = createDisplay(scenario,egoVehicle);

% Setup data logs
timeHistory = 0:scenario.SampleTime:(scenario.StopTime-scenario.SampleTime);
laneDetectionOffsetHistory = NaN(5,length(timeHistory));
timeStep = 1;
restart(scenario)
running = true;
while running
    % Simulation time
    simTime  = scenario.SimulationTime;

    % Get the ground truth lane boundaries in the ego vehicle frame
    groundTruthLanes = laneBoundaries(egoVehicle,'XDistance',0:5:70,'AllBoundaries',true);
    [egoBoundaries,egoBoundaryExist] = findEgoBoundaries(groundTruthLanes); %% ego lane and adjac
    laneBoundaryDetections = laneDetector(egoBoundaries(egoBoundaryExist),simTime); %% ego lane a
```

```
    laneObjectDetections = packLanesAsObjectDetections(laneBoundaryDetections); %% convert lane |

    % Log lane detections
    for laneIDX = 1:length(laneObjectDetections)
        laneDetectionOffsetHistory(laneIDX,timeStep) = laneObjectDetections{laneIDX}.Measurement
    end

    % Visualization road and lane ground truth in ego frame
    updateDisplay(egoCarBEP,egoVehicle,egoBoundaries);

    % Advance to the next step
    timeStep = timeStep + 1;
    running = advance(scenario);
end
```

Since the lane detector always reports the two lane markings on each side of the ego vehicle, the lane change causes it to report discontinuous lane markings. You can observe it in the graph below.

```
f=figure;
plot(timeHistory,laneDetectionOffsetHistory(1:4,:),'LineWidth',2)
xlabel('Time (s)')
ylabel('Lane detections lateral offset (m)')
legend('Adjacent left', 'Left', 'Right', 'Adjacent Right','Orientation','horizontal','Location',
grid
p = snapnow;

close(f)
```

Define a Multi-Lane Tracker and Track Lanes

Define a multi-lane tracker using the `trackerGNN` object. To delete undetected lane boundaries quickly, set the tracker to delete tracked lanes after three misses in three updates. Also set the maximum number of tracks to 10.

Use the `singer` acceleration model to model the way lane boundaries change over time. The Singer acceleration model enables you to model accelerations that decay with time, and you can set the decay rate using the decay constant `tau`. You use the `initSingerLane` on page 5-0   function modified from the `initsingerekf` function by setting the decay constant `tau` to 1, because the lane change maneuver time is relatively short. The function is attached at the end of the script. Note that the three dimensions defined for the Singer acceleration state are the offset, the heading, and the curvature of the lane boundary, which are the same as those reported in the lane detection.

```
laneTracker = trackerGNN('FilterInitializationFcn', @initSingerLane, 'DeletionThreshold', [3 3],
```

Rerun the scenario to track the lane boundaries.

```
laneTrackOffsetHistory = NaN(5,length(timeHistory));
timeStep = 1;
restart(scenario)
restart(egoVehicle);
reset(laneDetector);
running = true;
while running
    % Simulation time
    simTime  = scenario.SimulationTime;

    % Get the ground truth lane boundaries in the ego vehicle frame
    groundTruthLanes = laneBoundaries(egoVehicle,'XDistance',0:5:70,'AllBoundaries',true);
```

```
        [egoBoundaries,egoBoundaryExist] = findEgoBoundaries(groundTruthLanes); %% ego lane and adjac
        laneBoundaryDetections = laneDetector(egoBoundaries(egoBoundaryExist),simTime); %% ego lane a
        laneObjectDetections = packLanesAsObjectDetections(laneBoundaryDetections); %% convert lane l

        % Track the lanes
        laneTracks = laneTracker(laneObjectDetections,simTime);

        % Log data
        timeHistory(timeStep) = simTime;
        for laneIDX = 1:length(laneTracks)
            laneTrackOffsetHistory(laneTracks(laneIDX).TrackID,timeStep) = laneTracks(laneIDX).State
        end

        % Visualization road and lane ground truth in ego frame
        updateDisplay(egoCarBEP,egoVehicle,egoBoundaries);

        % Advance to the next step
        timeStep = timeStep + 1;
        running = advance(scenario);
    end
```

Plot the lateral offset of the tracked lane boundaries. Observe that the tracked lane boundaries are continuous and do not break when the ego vehicle performs the lane change maneuver.

```
figure
plot(timeHistory,laneTrackOffsetHistory(:,:),'LineWidth',2)
xlabel('Time (s)')
```
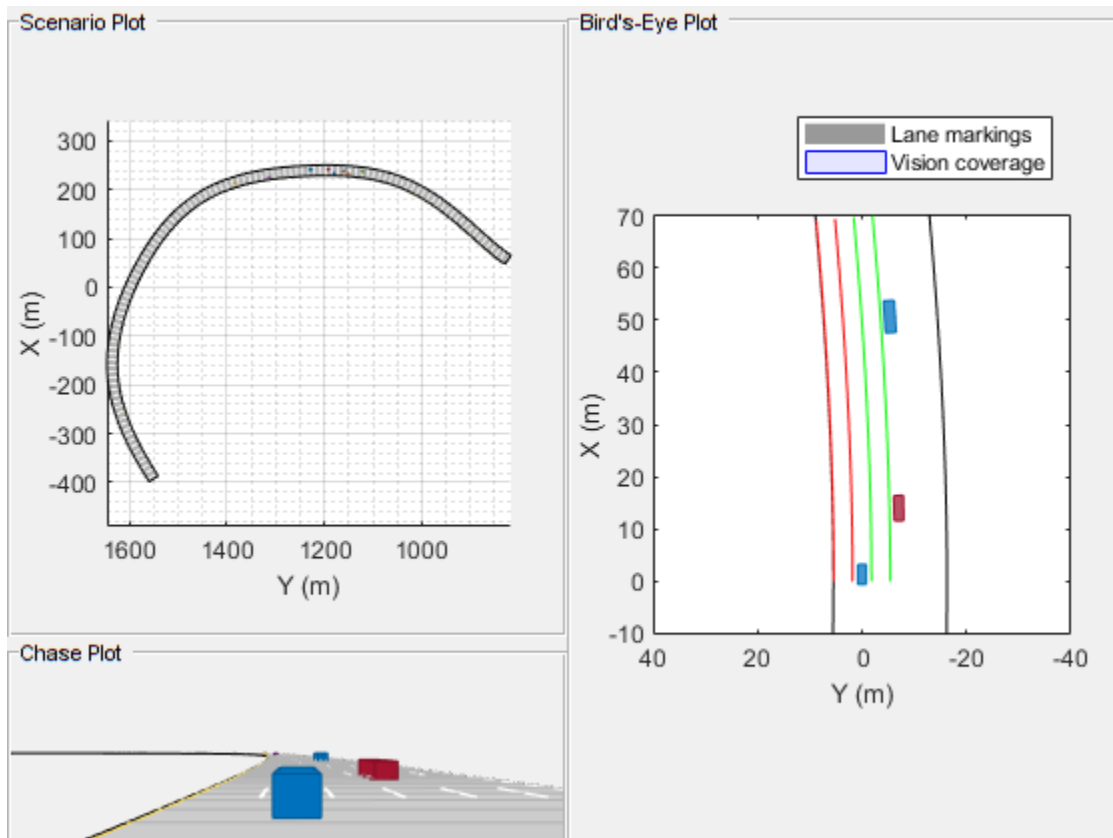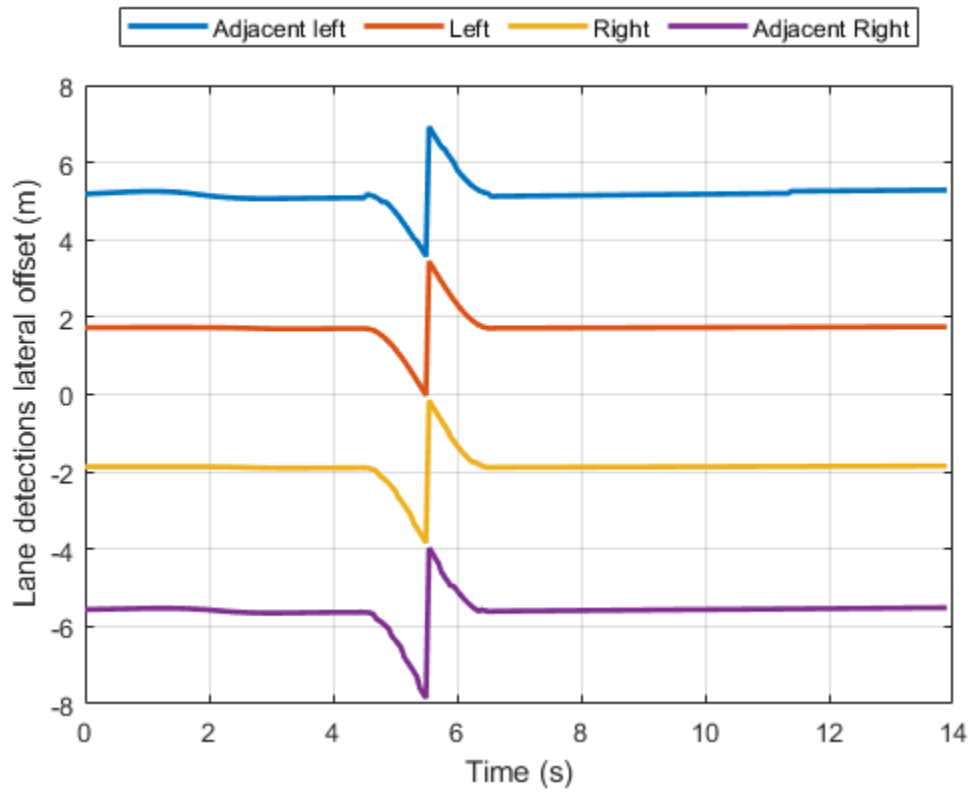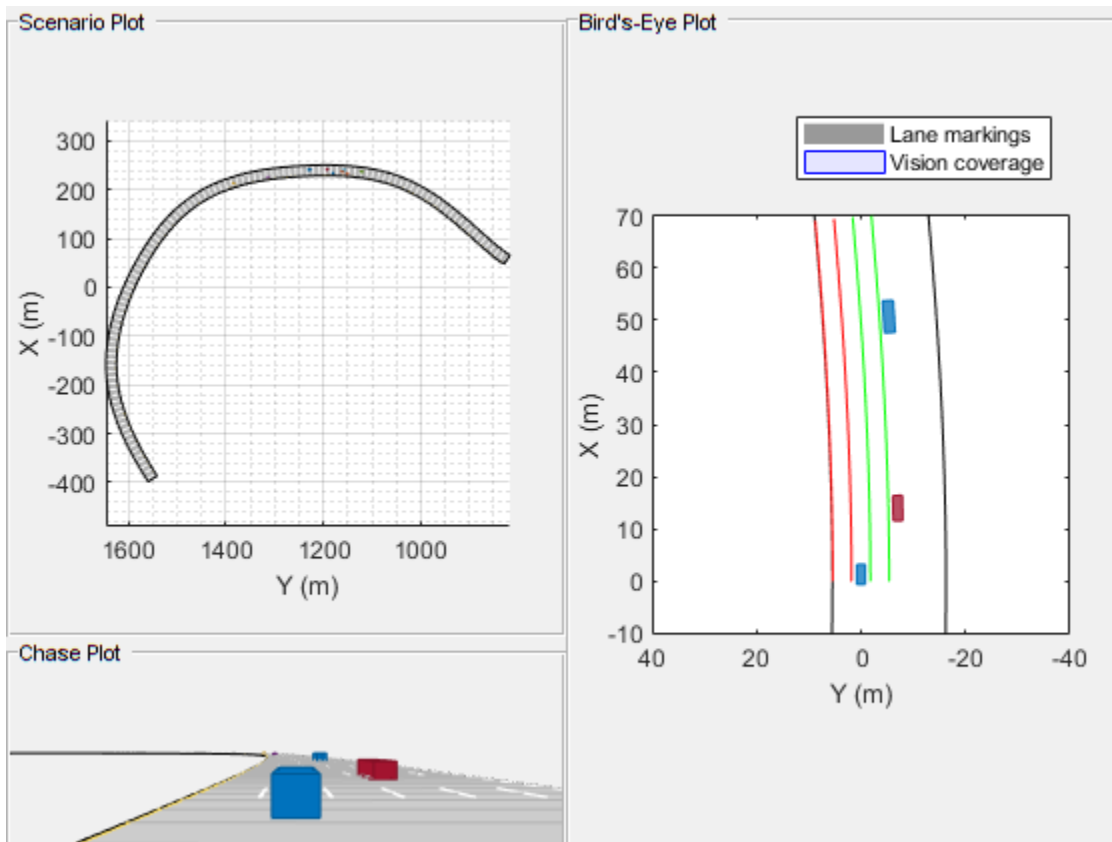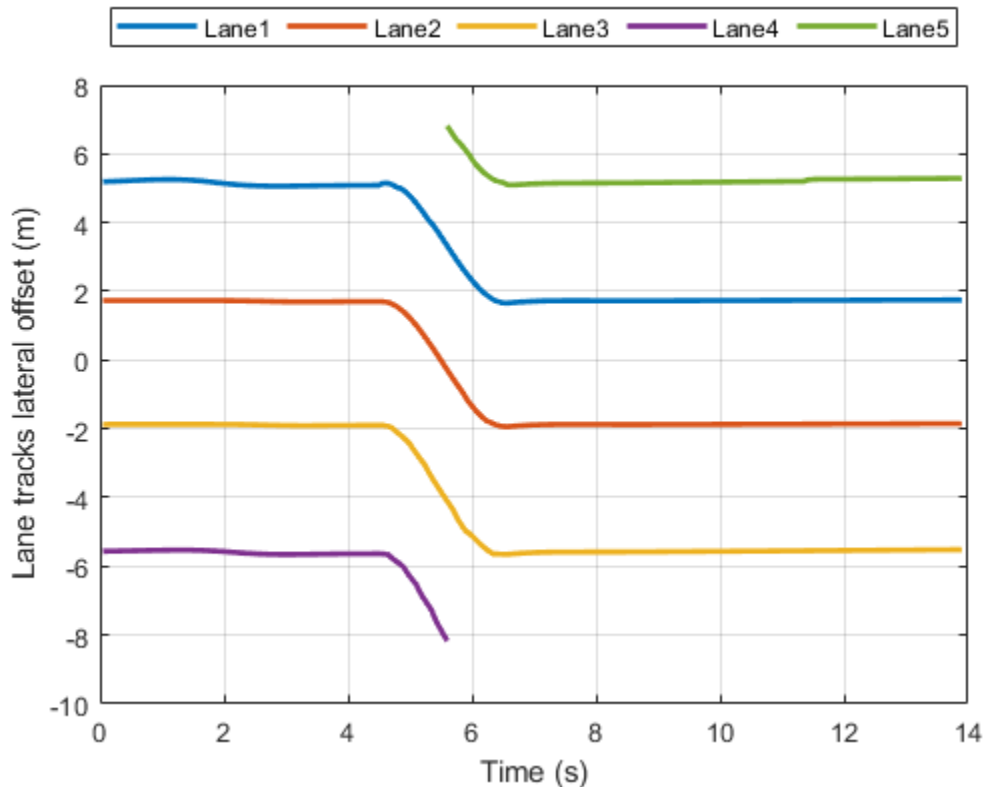
```
ylabel('Lane tracks lateral offset (m)')
legend('Lane1', 'Lane2', 'Lane3', 'Lane4', 'Lane5', 'Orientation','horizontal','Location','northo
grid
```

Summary

In this example, you learned how to track multiple lanes. Without tracking the lanes, the lane detector reports discontinuous lane offsets relative to the ego vehicle when the ego vehicle changes lanes. The discontinuity in lane offsets can cause significant performance degradation of a closed-loop automated lane change system. You used a tracker to track the lanes and observed that the lane boundary offsets are continuous and can provide a stable input to the lane change system.

Supporting functions

createDisplay Create the display for this example

```
function egoCarBEP = createDisplay(scenario,egoVehicle)
hFigure = figure;
hPanel1 = uipanel(hFigure,'Units','Normalized','Position',[0 1/4 1/2 3/4],'Title','Scenario Plot
hPanel2 = uipanel(hFigure,'Units','Normalized','Position',[0 0 1/2 1/4],'Title','Chase Plot');
hPanel3 = uipanel(hFigure,'Units','Normalized','Position',[1/2 0 1/2 1],'Title','Bird''s-Eye Plot
hAxes1 = axes('Parent',hPanel1);
hAxes2 = axes('Parent',hPanel2);
hAxes3 = axes('Parent',hPanel3);
legend(hAxes3,'AutoUpdate','off')
scenario.plot('Parent',hAxes1)      % plot is a method of drivingScenario Class
chasePlot(egoVehicle,'Parent',hAxes2); % chase plot following the egoVehicle
```

5-655

```
egoCarBEP = birdsEyePlot('Parent',hAxes3,'XLimits',[-10 70],'YLimits',[-40 40]);
% Set up plotting type
outlinePlotter(egoCarBEP,'Tag','Platforms');
laneBoundaryPlotter(egoCarBEP,'Tag','Roads');
laneBoundaryPlotter(egoCarBEP,'Color','r','LineStyle','-','Tag','Left1');
laneBoundaryPlotter(egoCarBEP,'Color','g','LineStyle','-','Tag','Right1');
laneBoundaryPlotter(egoCarBEP,'Color','r','LineStyle','-','Tag','Left2');
laneBoundaryPlotter(egoCarBEP,'Color','g','LineStyle','-','Tag','Right2');
laneMarkingPlotter(egoCarBEP,'DisplayName','Lane markings','Tag','LaneMarkings');
coverageAreaPlotter(egoCarBEP,'DisplayName','Vision coverage','FaceAlpha',0.1,'FaceColor','b','Ed
end
```

updateDisplay Update the display for this example

```
function updateDisplay(egoCarBEP,egoVehicle,LaneBdryIn)
[position,yaw,leng,width,originOffset,color] = targetOutlines(egoVehicle);
outlineplotter = findPlotter(egoCarBEP,'Tag','Platforms');
plotOutline(outlineplotter, position, yaw, leng, width, ...
    'OriginOffset',originOffset,'Color',color)
rbdry = egoVehicle.roadBoundaries;
roadPlotter = findPlotter(egoCarBEP,'Tag','Roads');
roadPlotter.plotLaneBoundary(rbdry)
lbllPlotter = findPlotter(egoCarBEP,'Tag','Left2');
plotLaneBoundary(lbllPlotter,{LaneBdryIn(1).Coordinates})
lblPlotter = findPlotter(egoCarBEP,'Tag','Left1');
plotLaneBoundary(lblPlotter,{LaneBdryIn(2).Coordinates})
lbrPlotter = findPlotter(egoCarBEP,'Tag','Right1');
plotLaneBoundary(lbrPlotter,{LaneBdryIn(3).Coordinates})
lbrrPlotter = findPlotter(egoCarBEP,'Tag','Right2');
plotLaneBoundary(lbrrPlotter,{LaneBdryIn(4).Coordinates})
end
```

findEgoBoundaries Return the two nearest lane boundaries on each side of the ego vehicle

```
function [egoBoundaries,egoBoundaryExist] = findEgoBoundaries(groundTruthLanes)
%findEgoBoundaries  Find the two adjacent lane boundaries on each side of the ego
%  [egoBoundaries,egoBoundaryExist] = findEgoBoundaries(groundTruthLanes)
%  egoBoundaries - A 4x1 struct of lane boundaries ordered as: adjacent
%  left, left, right, and adjacent right
egoBoundaries = groundTruthLanes(1:4);
lateralOffsets = [groundTruthLanes.LateralOffset];
[sortedOffsets, inds] = sort(lateralOffsets);
egoBoundaryExist = [true;true;true;true];

% Left lane and left adjacent lane
idxLeft = find(sortedOffsets>0,2,'first');
numLeft = length(idxLeft);
egoBoundaries(2) = groundTruthLanes(inds(idxLeft(1)));
if numLeft>1
    egoBoundaries(1) = groundTruthLanes(inds(idxLeft(2)));
else % if left adjacent lane does not exist
    egoBoundaries(1) = egoBoundaries(2);
    egoBoundaryExist(1) = false;
end

% Right lane and right adjacent lane
idxRight = find(sortedOffsets<0,2,'last');
numRight = length(idxRight);
```

```
egoBoundaries(3) = groundTruthLanes(inds(idxRight(end)));
if numRight>1
    egoBoundaries(4) = groundTruthLanes(inds(idxRight(1)));
else % if right adjacent lane does not exist
    egoBoundaries(4) = egoBoundaries(3);
    egoBoundaryExist(4) = false;
end
end
```

packLanesAsObjectDetections Return lane boundary detections as a cell array of objectDetection objects

```
function laneObjectDetections = packLanesAsObjectDetections(laneBoundaryDetections)
%packLanesAsObjectDetections  Packs lane detections as a cell array of objectDetection
laneStrengths = [laneBoundaryDetections.LaneBoundaries.Strength];
IdxValid = find(laneStrengths>0);
numLaneDetections = length(IdxValid);
meas = zeros(3,1);
measurementParameters = struct(...
    'Frame', 'rectangular', ...
    'OriginPosition', [0 0 0]', ...
    'Orientation', eye(3,3), ...
    'HasVelocity', false, ...
    'HasElevation', false);

detection = objectDetection(laneBoundaryDetections.Time,meas, ...
    'MeasurementNoise', eye(3,3)/10, ...
    'SensorIndex', laneBoundaryDetections.SensorIndex, ...
    'MeasurementParameters' ,measurementParameters);
laneObjectDetections = repmat({detection},numLaneDetections,1);
for i = 1:numLaneDetections
    meas = [laneBoundaryDetections.LaneBoundaries(IdxValid(i)).LateralOffset ...
        laneBoundaryDetections.LaneBoundaries(IdxValid(i)).HeadingAngle/180*pi ...
        laneBoundaryDetections.LaneBoundaries(IdxValid(i)).Curvature/180*pi];
    laneObjectDetections{i}.Measurement = meas;
end
end
```

initSingerLane Define the Singer motion model for the lane boundary filter

```
function filter = initSingerLane(detection)
filter = initsingerekf(detection);
tau = 1;
filter.StateTransitionFcn = @(state,dt)singer(state,dt,tau);
filter.StateTransitionJacobianFcn = @(state,dt)singerjac(state,dt,tau);
filter.ProcessNoise = singerProcessNoise(zeros(9,1),1,tau,1);
end
```

Grid-based Tracking in Urban Environments Using Multiple Lidars

This example shows how to track moving objects with multiple lidars using a grid-based tracker. A grid-based tracker enables early fusion of data from high-resolution sensors such as radars and lidars to create a global object list.

Introduction

Most multi-object tracking approaches represent the environment as a set of discrete and unknown number of objects. The job of the tracker is to estimate the number of objects and their corresponding states, such as position, velocity, and dimensions, using the sensor measurements. With high-resolution sensors such as radar or lidar, the tracking algorithm can be configured using point-object trackers or extended object trackers.

Point-Object Trackers

Point-object trackers assume that each object may give rise to at most one detection per sensor. Therefore, when using point-target trackers for tracking extended objects, features like bounding box detections are first extracted from the sensor measurements at the object-level. These object-level features then get fused with object-level hypothesis from the tracker. A poor object-level extraction algorithm at the sensor level (such as imperfect clustering) thus greatly impacts the performance of the tracker. For an example of this workflow, refer to "Track Vehicles Using Lidar: From Point Cloud to Track List" (Automated Driving Toolbox) example.

Extended Object Trackers

On the other hand, extended object trackers process the detections without extracting object-level hypothesis at the sensor level. Extended object trackers associate sensor measurements directly with the object-level hypothesis maintained by tracker. To do this, a class of algorithms typically requires complex measurement models of the object extents specific to each sensor modality. For example, refer to "Extended Object Tracking with Lidar for Airport Ground Surveillance" on page 5-637 and "Extended Object Tracking of Highway Vehicles with Radar and Camera" (Automated Driving Toolbox) to learn how to configure a multi-object PHD tracker for lidar and radar respectively.

A grid-based tracker can be considered as a type of extended object tracking algorithm which uses a dynamic occupancy grid map as an intermediate representation of the environment. In a dynamic occupancy grid map, the environment is discretized using a set of 2-D grid cells. The dynamic map represents the occupancy as well as kinematics of the space represented by a grid cell. Using the dynamic map estimate and further classification of cells as static and dynamic serves as a preprocessing step to filter out measurements from static objects and to reduce the computational complexity.

In this example, you use the `trackerGridRFS` System object™ to configure the grid-based tracker. This tracker uses the Random Finite Set (RFS) formulation with Dempster-Shafer approximation [1] to estimate the dynamic map. Further, it uses a nearest neighbor cell-to-track association [2] scheme to track dynamic objects in the scene. To initialize new tracks, the tracker uses the DBSCAN algorithm to cluster unassigned dynamic grid cells.

Set Up Scenario and Lidar Sensor Models

The scenario used in this example was created using the Driving Scenario Designer (Automated Driving Toolbox) app and was exported to a MATLAB® function. This MATLAB function was wrapped

as a helper function `helperCreateMultiLidarDrivingScenario`. The scenario represents an urban intersection scene and contains a variety of objects that include like pedestrians, bicyclists, cars, and trucks.

The ego vehicle is equipped with 6 homogeneous lidars, each with a horizontal field of view of 90 degrees and a vertical field of view of 40 degrees. The lidars are simulated using the `lidarPointCloudGenerator` (Automated Driving Toolbox) System object. Each lidar has 32 elevation channels and has a resolution of 0.16 degrees in azimuth. Under this configuration, each lidar sensor outputs approximately 18,000 points per scan. The configuration of each sensor is shown here.
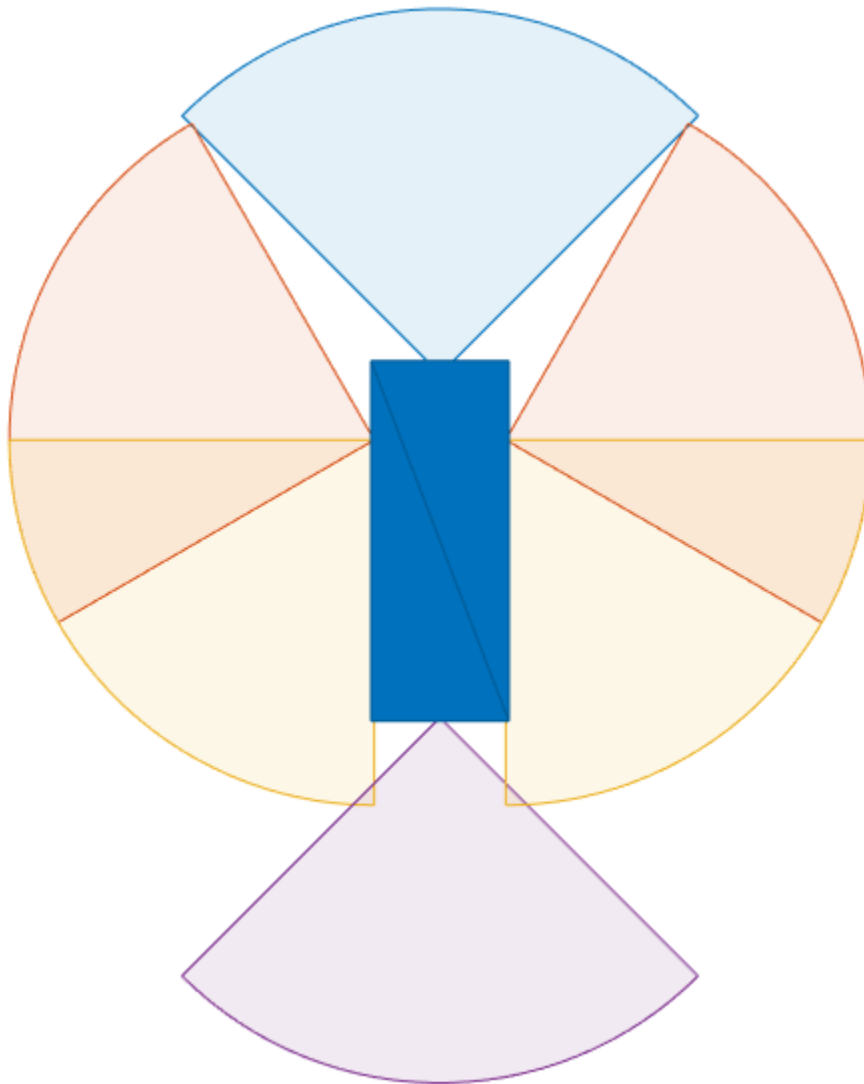

```
% For reproducible results
rng(2020);

% Create scenario
[scenario, egoVehicle, lidars] = helperCreateMultiLidarDrivingScenario;
```

The scenario and the data from the different lidars can be visualized in the animation below. For brevity and to make the example easier to visualize, the lidar is configured to not return point cloud from the ground by specifying the `HasRoadsInputPort` property as `false`. When using real data or if using simulated data from roads, the returns from ground and other environment must be removed using point cloud preprocessing. For more information, refer to the "Ground Plane and Obstacle Detection Using Lidar" (Automated Driving Toolbox) example.

Set Up Grid-Based Tracker

You define a grid-based tracker using `trackerGridRFS` to track dynamic objects in the scene. The first step of defining the tracker is setting up sensor configurations as `trackingSensorConfiguration` objects. The sensor configurations allow you to specify the mounting of each sensor with respect to the tracking coordinate frame. The sensor configurations also allow you to specify the detection limits - field of view and maximum range - of each sensor. In this example, you use the properties of the simulated lidar sensors to define these properties.

The utility function helperGetLidarConfig on page 5-0 uses the simulated lidar sensor model and returns its respective configuration. In this example, the targets are tracked in the global or world coordinate system by using the simulated pose of the vehicle. This information is typically obtained via an inertial navigation system. As the sensors move in the scenario system, their configuration must be updated each time by specifying the configurations as an input to the tracker.

```
% Store configurations of all sensor
sensorConfigs = cell(numel(lidars),1);
```

```
% Fill in sensor configurations
for i = 1:numel(sensorConfigs)
    sensorConfigs{i} = helperGetLidarConfig(lidars{i},egoVehicle);
end

% Create tracker. You can define the properties before using the tracker.
tracker = trackerGridRFS('SensorConfigurations',sensorConfigs,...
    'HasSensorConfigurationsInput',true);
```

The tracker uses a two-dimensional grid for the intermediate representation of the environment. The grid is defined by 3 attributes: its length, its width, and the resolution. The length and width describe the span of the grid in local X and local Y direction of the ego vehicle respectively. The resolution defines the number of cells per meter of the grid. In this example, you use a 120 m by 120 m grid with 2 cells per meter.

```
tracker.GridLength = 120; % meters
tracker.GridWidth = 120; % meters
tracker.GridResolution = 2; % 1/meters
```

In addition to defining the grid, you also define the relative position of the ego vehicle by specifying the origin of the grid (left corner) with respect to the origin of the ego vehicle. In this example, the ego vehicle is located at the center of the grid.

```
tracker.GridOriginInLocal = [-tracker.GridLength/2 -tracker.GridWidth/2];
```

The tracker uses particle-based methods to estimate the state of each grid cell and further classify them as dynamic or static. It uses a fixed number of persistent particles on the grid which defines the distribution of existing targets. It also uses a fixed number of particles to sample the distribution for newborn targets. These birth particles get sampled in different grid cells based on the probability of birth. Further, the velocity and other unknown states like turn-rate and acceleration (applicable when MotionModel of the tracker is not constant-velocity) of the particles is sampled uniformly using prior information supplied using prior limits. A resampling step assures that the number of particles on the grid remain constant.

```
tracker.NumParticles = 1e5; % Number of persistent particles
tracker.NumBirthParticles = 2e4; % Number of birth particles
tracker.VelocityLimits = [-15 15;-15 15]; % To sample velocity of birth particles (m/s)
tracker.BirthProbability = 0.025; % Probability of birth in each grid cell
tracker.ProcessNoise = 5*eye(2); % Process noise of particles for prediction as variance of [ax;a
```

The tracker uses the Dempster-Shafer approach to define the occupancy of each cell. The dynamic grid estimates the belief mass for occupancy and free state of the grid. During prediction, the occupancy belief mass of the grid cell updates due to prediction of the particle distribution. The DeathRate controls the probability of survival (P_s) of particles and results in a decay of occupancy belief mass during prediction. As the free belief mass is not linked to the particles, the free belief mass decays using a pre-specified, constant discount factor. This discount factor specifies the probability that free regions remain free during prediction.

```
tracker.DeathRate = 1e-3; % Per unit time. Translates to Ps = 0.9999 for 10 Hz
tracker.FreeSpaceDiscountFactor = 1e-2; % Per unit time. Translates to a discount factor of 0.63
```

After estimation of state of each grid cell, the tracker classifies each grid cell as static or dynamic by using its estimated velocity and associated uncertainty. Further, the tracker uses dynamic cells to extract object-level hypothesis using the following technique:

Each dynamic grid cell is considered for assignment with existing tracks. A dynamic grid cell is assigned to its nearest track if the negative log-likelihood between a grid cell and a track falls below

an assignment threshold. A dynamic grid cell outside the assignment threshold is considered unassigned. The tracker uses unassigned grid cells at each step to initiate new tracks. Because multiple unassigned grid cells can belong to the same object track, a DBSCAN clustering algorithm is used to assist in this step. Because there are false positives while classifying the cells as static or dynamic, the tracker filters those false alarms in two ways. First, only unassigned cells which form clusters with more than a specified number of points (`MinNumPointsPerCluster`) can create new tracks. Second, each track is initialized as a tentative track first and is only confirmed if its detected M out of N times.

```
tracker.AssignmentThreshold = 8; % Maximum distance or negative log-likelihood between cell and t
tracker.MinNumCellsPerCluster = 6; % Minimum number of grid cells per cluster for creating new tr
tracker.ClusteringThreshold = 1; % Minimum Euclidean distance between two cells for clustering
tracker.ConfirmationThreshold = [3 4]; % Threshold to confirm tracks
tracker.DeletionThreshold = [4 4]; % Threshold to delete confirmed tracks
```

You can also accelerate simulation by performing the dynamic map estimation on GPU by specifying the `UseGPU` property of the tracker.

```
tracker.UseGPU = false;
```

Visualization

The visualization used for this example is defined using a helper class, `helperGridTrackingDisplay`, attached with this example. The visualization contains three parts.

- Ground truth - Front View: This panel shows the front-view of the ground truth using a chase plot from the ego vehicle. To emphasize dynamic actors in the scene, the static objects are shown in gray.
- Lidar Views: These panels show the point cloud returns from each sensor.
- Grid-based tracker**:** This panel shows the grid-based tracker outputs. The tracks are shown as boxes, each annotated by their identity. The tracks are overlaid on the dynamic grid map. The colors of the dynamic grid cells are defined according to the color wheel, which represents the direction of motion of in the scenario frame. The static grid cells are represented using a grayscale according to their occupancy. The degree of grayness denotes the probability of the space occupied by the grid cell as free. The positions of the tracks are shown in the ego vehicle coordinate system, while the velocity vector corresponds to the velocity of the track in the scenario frame.

```
display = helperGridTrackingDisplay;
```

Run Scenario and Track Dynamic Objects

Next, run the scenario, simulate lidar sensor data from each lidar sensor, and process the data using the grid-based tracker.

```
% Initialize pointCloud ouputs from each sensor
ptClouds = cell(numel(lidars),1);
sensorConfigs = cell(numel(lidars),1);

while advance(scenario)
    % Current simulation time
    time = scenario.SimulationTime;

    % Poses of objects with respect to ego vehicle
    tgtPoses = targetPoses(egoVehicle);
```

```
    % Simulate point cloud from each sensor
    for i = 1:numel(lidars)
        [ptClouds{i}, isValidTime] = step(lidars{i},tgtPoses,time);
        sensorConfigs{i} = helperGetLidarConfig(lidars{i},egoVehicle);
    end

    % Pack point clouds as sensor data format required by the tracker
    sensorData = packAsSensorData(ptClouds,sensorConfigs,time);

    % Call the tracker
    tracks = tracker(sensorData,sensorConfigs,time);

    % Update the display
    display(scenario, egoVehicle, lidars, ptClouds, tracker, tracks);
    drawnow;
end
```

Results

Next, analyze the performance of the tracker using the visualization used in this example.

The grid-based tracker uses the dynamic cells from the estimated grid map to extract object tracks. The animation below shows the results of the tracker in this scenario. The "Grid-based tracker" panel shows the estimated dynamic map as well as the estimated tracks of the objects. It also shows the configuration of the sensors mounted on the ego vehicle as blue circular sectors. Notice that the area encapsulated by these sensors is estimated as "gray" in the dynamic map, representing that this area is not observed by any of the sensors. This patch also serves as an indication of ego-vehicle's position on the dynamic grid.

Notice that the tracks are extracted only from the dynamic grid cells and hence the tracker is able to filter out static objects. Also notice that after a vehicle enters the grid region, its track establishment takes few time steps. This is due to two main reasons. First, there is an establishment delay in classification of the cell as dynamic. Second, the confirmation threshold for the object takes some steps to establish a track as a confirmed object.
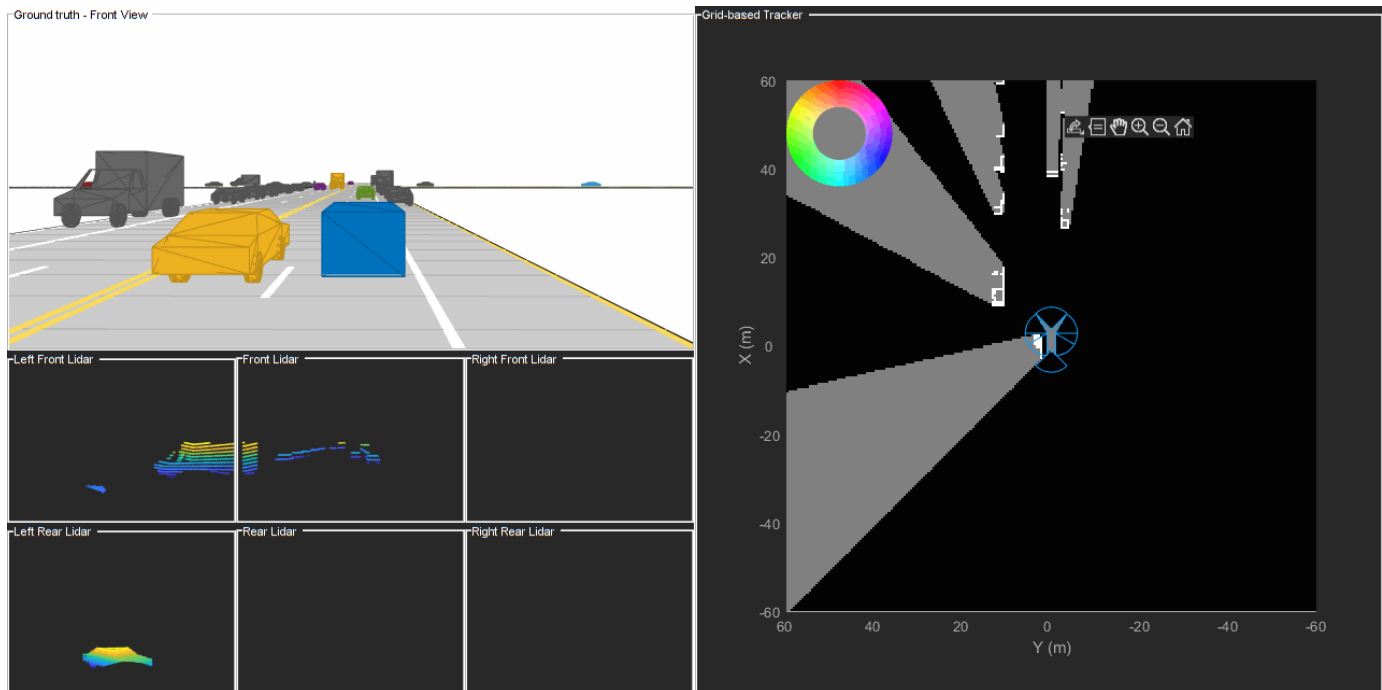
Next, you look at the history of a few tracks to understand how the state of a track gets affected by the estimation of the dynamic grid.

Longitudinally Moving Tracks

The following snapshots show the history for the track denoted by T1. The T1 track represents the yellow car that passes the ego vehicle on the left during the first few seconds of the simulation. Notice that the grid cells occupied by this track are colored in red, indicating their motion in the positive X direction. The track obtains the track's velocity and heading information using the velocity distribution of the assigned grid cells. It also obtains its length, width, and orientation using the spatial distribution of the assigned grid cells. The default `TrackUpdateFcn` of the `trackerGridRFS` extracts new length, width, and orientation information from the spatial distribution of associated grid cells at every step. This effect can be seen in the snapshots below, where the length and width of the track adjusts according to the bounding box of the associated grid cells. An additional filtering scheme can be added using the predicted length, width, and orientation of the track by using a custom `TrackUpdateFcn`.

```
% Show snapshots for TrackID = 1. Also shows close tracks like T3 and T4
% representing car and truck moving in the opposite direction.
showSnapshots(display.GridView,1);
```
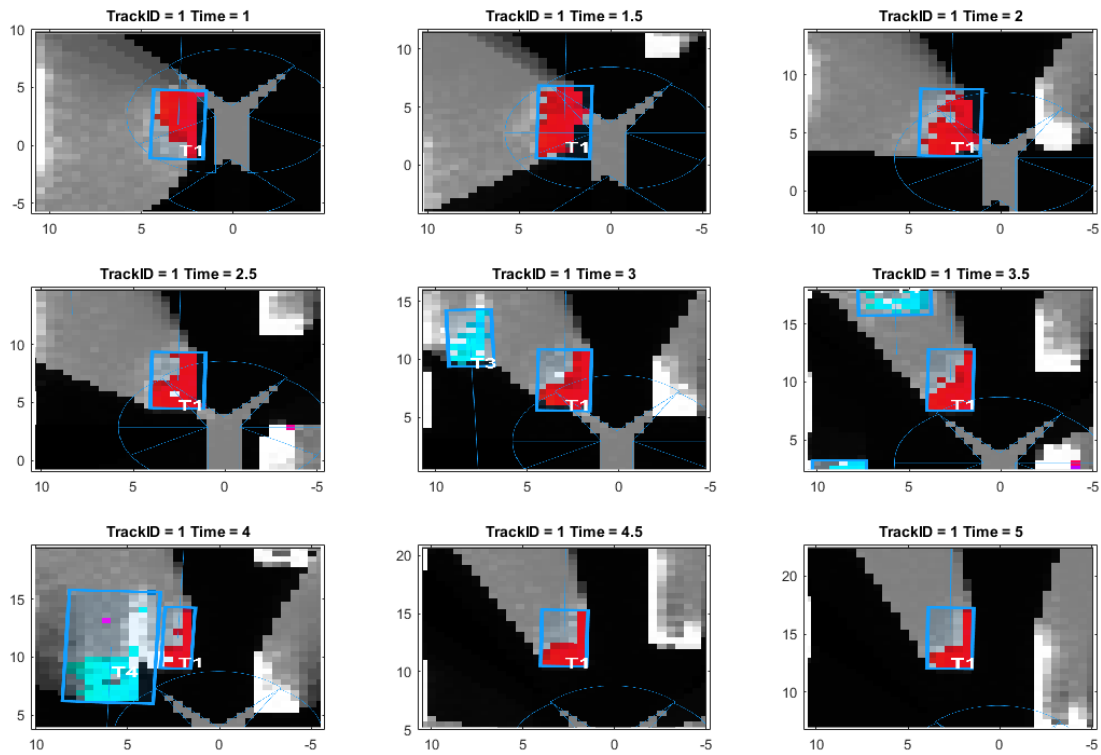
```
showSnapshots(display.GridView,4);
```
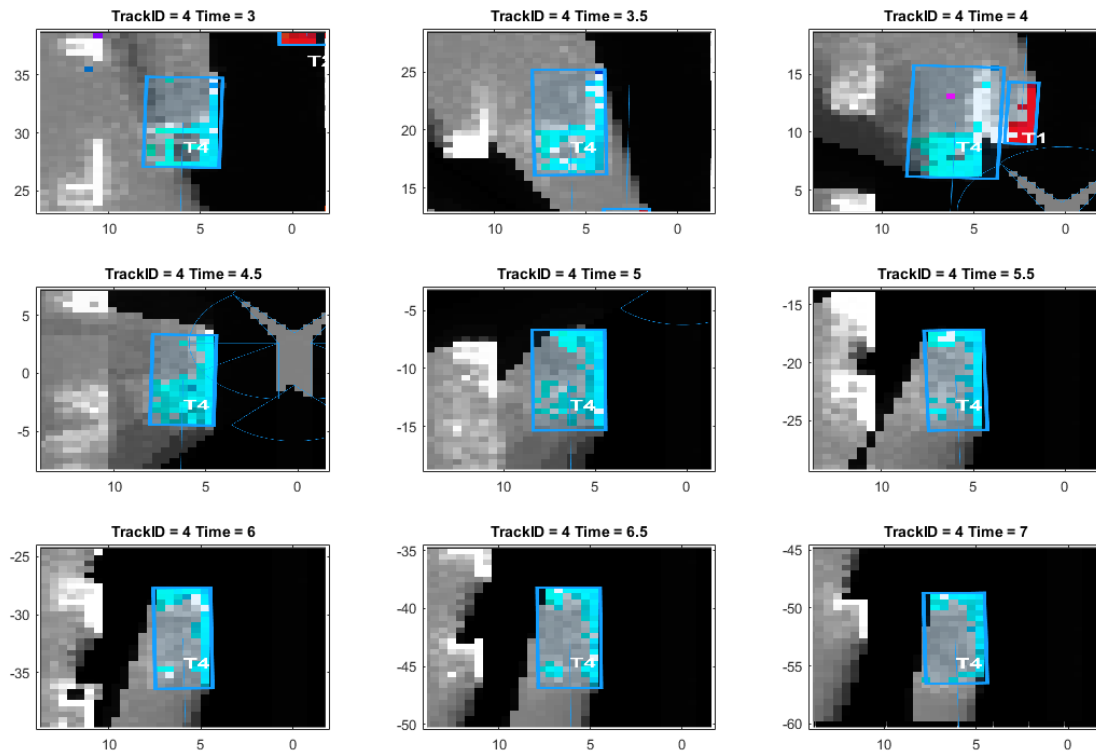
Next, take a closer look at the history of T4. The T4 track represents the truck moving in the opposite direction of the ego vehicle. Notice that the grid cells representing this track are colored in blue, representing the estimated motion direction of the grid cell. Also, notice that there are grid cells in the track that are misclassified by the tracker as static (white color). These misclassified grid cells often occur when sensors report previously occluded regions of an object, because the tracker has an establishment delay to classify these cells property.

Notice that at time = 4, when the truck and the vehicle came close to each other, the grid cells maintained their respective color, representing a stark difference between their estimated velocity directions. This also results in the correct data association between grid cells and predicted tracks of T1 and T4, which helps the tracker to resolve them as separate objects.

Laterally Moving Tracks

The following snapshots represent the track denoted by T7. This track represents the vehicle moving in the lateral direction, when the ego vehicle stops at the intersection. Notice that the grid cells of this track are colored in purple, representing the direction of motion in negative Y direction. Similar to other tracks, the track maintains its length and width using the spatial distribution of the assigned grid cells.

```
showSnapshots(display.GridView,7);
```
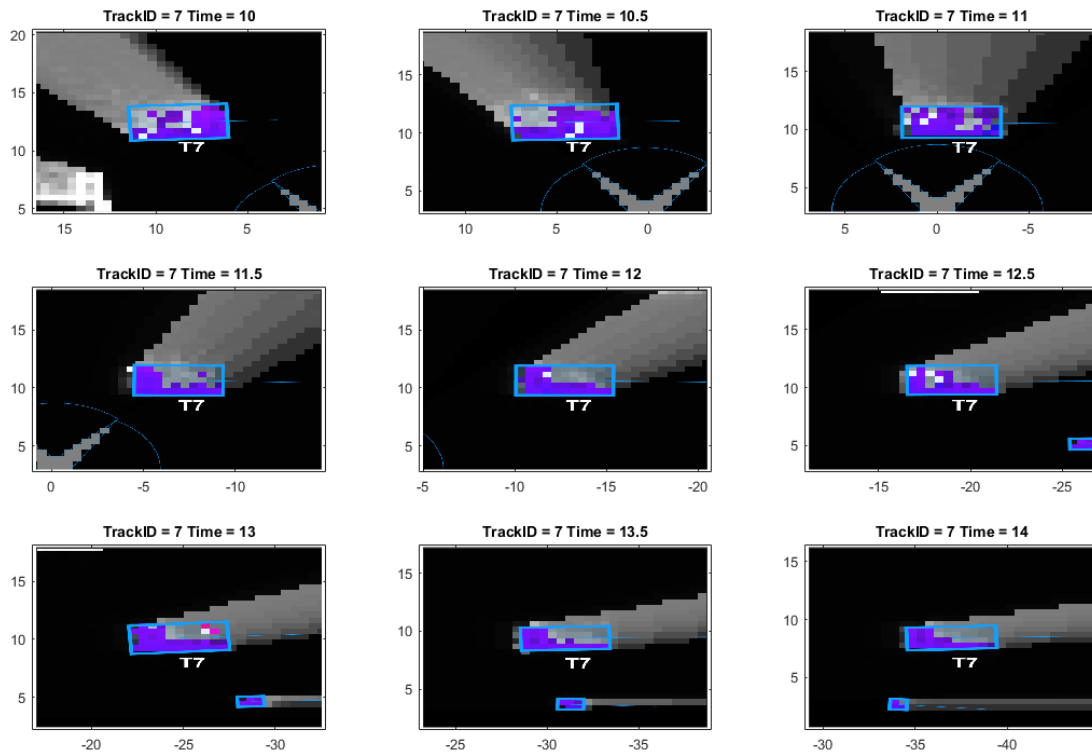
Tracks Changing Direction

In this example, you used a "constant-velocity" model with the tracker. This motion model assumes that the targets move at a constant velocity, meaning constant speed and direction. However, in urban scenes, this assumption is usually not accurate. To compensate for the unknown acceleration of the objects, a process noise is specified on the tracker. The following snapshots show the history of track T2. This track represents the vehicle directly in front of the ego vehicle. Notice in the ground truth that this vehicle turns right at the intersection.

```
showSnapshots(display.GridView, 2);
```
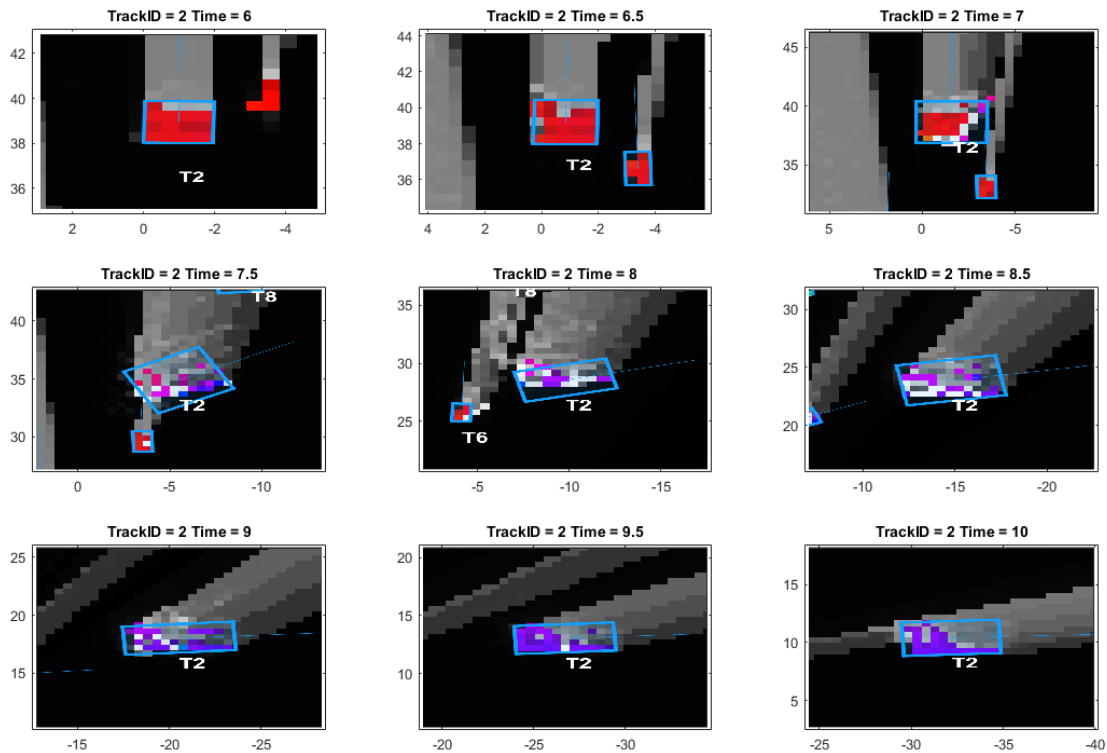
Notice that the color of the grid cells associated with this track changes from red to purple. Also, the transition of colors results in a few misclassified cells, which can result in a poor estimate of length and width of the vehicle. The ability of the tracker to maintain the track on this vehicle is due to a coupled effect of three main reasons. First, the tracker allows to specify an assignment threshold. Even if the predicted track does not align with the dynamic grid cells, it can associate with them up to a certain threshold. Second, to create a new track from grid cells that remain outside the threshold requires meeting the minimum number of cells criteria. Third, the tracker has a deletion threshold, which allows a track to be coasted for a few steps before deleting it. If the classification of grid cells is very poor during the turn, the track can survive a few steps and can get re-associated with the grid cells. Note that misclassified grid cells are far more observable with Track T8, as shown below in its history. The T8 track represents the light blue car traveling in the positive Y direction before taking at right turn at the intersection. This vehicle was partially occluded before the turn and had another closely traveling vehicle while making the turn.

```
showSnapshots(display.GridView,8);
```
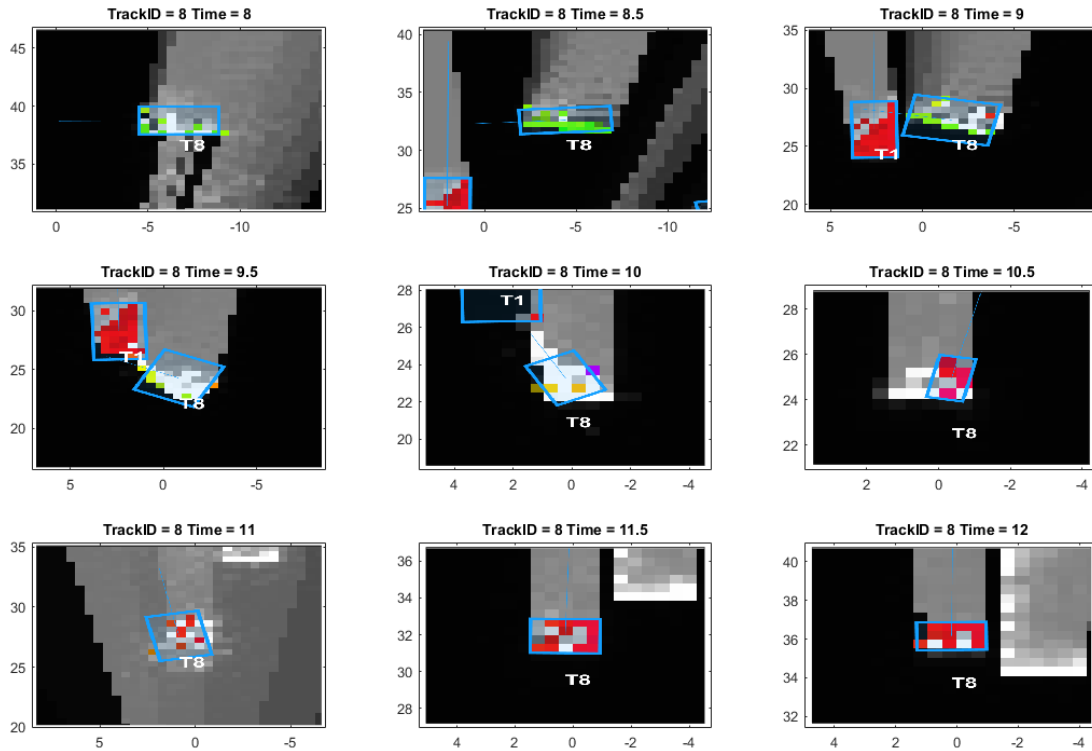
Summary

In this example, you learned the basics of a grid-based tracker and how it can be used to track dynamic objects in a complex urban driving environment. You also learned how to configure the tracker to track object using point clouds from multiple lidar sensors.

Supporting Functions

```
function sensorData = packAsSensorData(ptCloud, configs, time)
%The lidar simulation returns output as pointCloud object. The Location
%property of the point cloud is used to extract x,y and z locations of
%returns and pack them as structure with information required by a tracker.

sensorData = struct('SensorIndex',{},...
    'Time', {},...
    'Measurement', {},...
    'MeasurementParameters', {});

for i = 1:numel(ptCloud)
    % This sensor's cloud
    thisPtCloud = ptCloud{i};

    % Allows mapping between data and configurations without forcing an
    % ordered input and requiring configuration input for static sensors.
    sensorData(i).SensorIndex = configs{i}.SensorIndex;

    % Current time
```

```
        sensorData(i).Time = time;

        % Measurement as 3-by-N defininng locations of points
        sensorData(i).Measurement = reshape(thisPtCloud.Location,[],3)';

        % Data is reported in sensor coordinate frame and hence measurement
        % parameters are same as sensor transform parameters.
        sensorData(i).MeasurementParameters = configs{i}.SensorTransformParameters;
    end

    end

    function config = helperGetLidarConfig(lidar, ego)
    % Define transformation from sensor to ego
    senToEgo = struct('Frame',fusionCoordinateFrameType(1),...
        'OriginPosition',[lidar.SensorLocation(:);lidar.Height],...
        'Orientation',rotmat(quaternion([lidar.Yaw lidar.Pitch lidar.Roll],'eulerd','ZYX','frame'),'
        'IsParentToChild',true);

    % Define transformation from ego to tracking coordinates
    egoToScenario = struct('Frame',fusionCoordinateFrameType(1),...
        'OriginPosition',ego.Position(:),...
        'Orientation',rotmat(quaternion([ego.Yaw ego.Pitch ego.Roll],'eulerd','ZYX','frame'),'frame')
        'IsParentToChild',true);

    % Assemble using trackingSensorConfiguration.
    config = trackingSensorConfiguration(...
        'SensorIndex',lidar.SensorIndex,...
        'IsValidTime', true,...
        'SensorLimits',[lidar.AzimuthLimits;0 lidar.MaxRange],...
        'SensorTransformParameters',[senToEgo;egoToScenario],...
        'DetectionProbability',0.95);
    end
```

References

[1] Nuss, Dominik, et al. "A random finite set approach for dynamic occupancy grid maps with real-time application." The International Journal of Robotics Research 37.8 (2018): 841-866.

[2] Steyer, Sascha, Georg Tanzmeister, and Dirk Wollherr. "Object tracking based on evidential dynamic occupancy grids in urban environments." 2017 IEEE Intelligent Vehicles Symposium (IV). IEEE, 2017.

Automatic Tuning of the insfilterAsync Filter

The `insfilterAsync` (Navigation Toolbox) object is a complex extended Kalman filter that estimates the device pose. However, manually tuning the filter or finding the optimal values for the noise parameters can be a challenging task. This example illustrates how to use the `tune` (Navigation Toolbox) function to optimize the filter noise parameters.

Trajectory and Sensor Setup

To illustrate the tuning process of the `insfilterAsync` filter, use a simple random waypoint trajectory. The `imuSensor` and `gpsSensor` objects create inputs for the filter.

```matlab
% The IMU runs at 100 Hz and the GPS runs at 1 Hz.
imurate = 100;
gpsrate = 1;
decim = imurate/gpsrate;

% Create a random waypoint trajectory.
rng(1)
Npts = 4; % number of waypoints
wpPer = 5; % time between waypoints
tstart = 0;
tend = wpPer*(Npts -1);
wp = waypointTrajectory('Waypoints',5*rand(Npts,3), ...
        'TimeOfArrival',tstart:wpPer:tend, ...
        'Orientation',[quaternion.ones; randrot(Npts-1,1)], ...
        'SampleRate', imurate);

[Position,Orientation,Velocity,Acceleration,AngularVelocity] = lookupPose(...
    wp, tstart:(1/imurate):tend);

% Set up an IMU and process the trajectory.
imu = imuSensor('SampleRate',imurate);
loadparams(imu,fullfile(matlabroot, ...
    "toolbox\shared\positioning\positioningdata\generic.json"), ...
    "GenericLowCost9Axis");
[Accelerometer, Gyroscope, Magnetometer] = imu(Acceleration, ...
    AngularVelocity, Orientation);
imuData = timetable(Accelerometer,Gyroscope,Magnetometer,'SampleRate',imurate);

% Set up a GPS sensor and process the trajectory.
gps = gpsSensor('SampleRate', gpsrate,'DecayFactor',0.5, ...
    'HorizontalPositionAccuracy',1.6,'VerticalPositionAccuracy',1.6, ...
    'VelocityAccuracy',0.1);
[GPSPosition,GPSVelocity] = gps(Position(1:decim:end,:), Velocity(1:decim:end,:));
gpsData = timetable(GPSPosition,GPSVelocity,'SampleRate',gpsrate);

% Create a timetable for the tune function.
sensorData = synchronize(imuData,gpsData);

% Create a timetable capturing the ground truth pose.
groundTruth = timetable(Position,Orientation,'SampleRate',imurate);
```

Construct the Filter

The `insfilterAsync` filter fuses data from multiple sensors operating asynchronously

```matlab
filtUntuned = insfilterAsync;
```

5-671

Determine Filter Initial Conditions

Set the initial values for the `State` and `StateCovariance` properties based on the ground truth. Normally to obtain the initial values, you would use the first several samples of `sensorData` along with calibration routines. However, in this example the `groundTruth` is used to set the initial state for fast convergence of the filter.

```
idx = stateinfo(filtUntuned);
filtUntuned.State(idx.Orientation) = compact(Orientation(1));
filtUntuned.State(idx.AngularVelocity) = AngularVelocity(1,:);
filtUntuned.State(idx.Position) = Position(1,:);
filtUntuned.State(idx.Velocity) = Velocity(1,:);
filtUntuned.State(idx.Acceleration) = Acceleration(1,:);
filtUntuned.State(idx.AccelerometerBias) = imu.Accelerometer.ConstantBias;
filtUntuned.State(idx.GyroscopeBias) = imu.Gyroscope.ConstantBias;
filtUntuned.State(idx.GeomagneticFieldVector) = imu.MagneticField;
filtUntuned.State(idx.MagnetometerBias) = imu.Magnetometer.ConstantBias;
filtUntuned.StateCovariance = 1e-5*eye(numel(filtUntuned.State));

% Create a copy of the filtUntuned object for tuning later.
filtTuned = copy(filtUntuned);
```

Process `sensorData` with an Untuned Filter

Use the `tunernoise` function to create measurement noises which also need to be tuned. To illustrate the necessity for tuning, first use the filter with its default parameters.

```
mn = tunernoise('insfilterAsync');
[posUntunedEst, orientUntunedEst] = fuse(filtUntuned, sensorData, mn);
```

Tune the Filter and Process `sensorData`

Use the `tune` function to minimize the root mean squared (RMS) error between the `groundTruth` and state estimates.

```
cfg = tunerconfig(class(filtTuned),'MaxIterations',15,'StepForward',1.1);
tunedmn = tune(filtTuned,mn,sensorData,groundTruth,cfg);
```

Iteration	Parameter	Metric
1	AccelerometerNoise	4.7634
1	GyroscopeNoise	4.7439
1	MagnetometerNoise	4.7260
1	GPSPositionNoise	4.6562
1	GPSVelocityNoise	4.4895
1	QuaternionNoise	4.4895
1	AngularVelocityNoise	4.1764
1	PositionNoise	4.1764
1	VelocityNoise	4.1764
1	AccelerationNoise	4.1657
1	GyroscopeBiasNoise	4.1657
1	AccelerometerBiasNoise	4.1615
1	GeomagneticVectorNoise	4.1615
1	MagnetometerBiasNoise	4.1466
2	AccelerometerNoise	4.1466
2	GyroscopeNoise	4.1466
2	MagnetometerNoise	4.1218
2	GPSPositionNoise	4.1218

```
2          GPSVelocityNoise          3.9411
2          QuaternionNoise           3.9404
2          AngularVelocityNoise      3.2970
2          PositionNoise             3.2970
2          VelocityNoise             3.2970
2          AccelerationNoise         3.2735
2          GyroscopeBiasNoise        3.2735
2          AccelerometerBiasNoise    3.2711
2          GeomagneticVectorNoise    3.2711
2          MagnetometerBiasNoise     3.2709
3          AccelerometerNoise        3.2704
3          GyroscopeNoise            3.2475
3          MagnetometerNoise         3.2338
3          GPSPositionNoise          3.2338
3          GPSVelocityNoise          3.1333
3          QuaternionNoise           3.1333
3          AngularVelocityNoise      3.0267
3          PositionNoise             3.0267
3          VelocityNoise             3.0267
3          AccelerationNoise         3.0267
3          GyroscopeBiasNoise        3.0267
3          AccelerometerBiasNoise    3.0248
3          GeomagneticVectorNoise    3.0248
3          MagnetometerBiasNoise     3.0184
4          AccelerometerNoise        3.0156
4          GyroscopeNoise            3.0028
4          MagnetometerNoise         2.9903
4          GPSPositionNoise          2.9872
4          GPSVelocityNoise          2.9043
4          QuaternionNoise           2.9043
4          AngularVelocityNoise      2.7873
4          PositionNoise             2.7873
4          VelocityNoise             2.7873
4          AccelerationNoise         2.7833
4          GyroscopeBiasNoise        2.7833
4          AccelerometerBiasNoise    2.7821
4          GeomagneticVectorNoise    2.7821
4          MagnetometerBiasNoise     2.7727
5          AccelerometerNoise        2.7693
5          GyroscopeNoise            2.7591
5          MagnetometerNoise         2.7591
5          GPSPositionNoise          2.7580
5          GPSVelocityNoise          2.6806
5          QuaternionNoise           2.6804
5          AngularVelocityNoise      2.5956
5          PositionNoise             2.5956
5          VelocityNoise             2.5956
5          AccelerationNoise         2.5880
5          GyroscopeBiasNoise        2.5880
5          AccelerometerBiasNoise    2.5874
5          GeomagneticVectorNoise    2.5874
5          MagnetometerBiasNoise     2.5693
6          AccelerometerNoise        2.5651
6          GyroscopeNoise            2.5529
6          MagnetometerNoise         2.5472
6          GPSPositionNoise          2.5344
6          GPSVelocityNoise          2.4514
6          QuaternionNoise           2.4514
```

```
6          AngularVelocityNoise      2.3829
6          PositionNoise             2.3829
6          VelocityNoise             2.3829
6          AccelerationNoise         2.3826
6          GyroscopeBiasNoise        2.3826
6          AccelerometerBiasNoise    2.3822
6          GeomagneticVectorNoise    2.3822
6          MagnetometerBiasNoise     2.3675
7          AccelerometerNoise        2.3640
7          GyroscopeNoise            2.3547
7          MagnetometerNoise         2.3540
7          GPSPositionNoise          2.3428
7          GPSVelocityNoise          2.2634
7          QuaternionNoise           2.2633
7          AngularVelocityNoise      2.2179
7          PositionNoise             2.2179
7          VelocityNoise             2.2179
7          AccelerationNoise         2.2159
7          GyroscopeBiasNoise        2.2159
7          AccelerometerBiasNoise    2.2150
7          GeomagneticVectorNoise    2.2150
7          MagnetometerBiasNoise     2.2020
8          AccelerometerNoise        2.1967
8          GyroscopeNoise            2.1951
8          MagnetometerNoise         2.1948
8          GPSPositionNoise          2.1804
8          GPSVelocityNoise          2.1153
8          QuaternionNoise           2.1153
8          AngularVelocityNoise      2.0978
8          PositionNoise             2.0978
8          VelocityNoise             2.0978
8          AccelerationNoise         2.0944
8          GyroscopeBiasNoise        2.0944
8          AccelerometerBiasNoise    2.0944
8          GeomagneticVectorNoise    2.0944
8          MagnetometerBiasNoise     2.0822
9          AccelerometerNoise        2.0771
9          GyroscopeNoise            2.0758
9          MagnetometerNoise         2.0754
9          GPSPositionNoise          2.0645
9          GPSVelocityNoise          2.0028
9          QuaternionNoise           2.0027
9          AngularVelocityNoise      1.9943
9          PositionNoise             1.9943
9          VelocityNoise             1.9943
9          AccelerationNoise         1.9883
9          GyroscopeBiasNoise        1.9883
9          AccelerometerBiasNoise    1.9876
9          GeomagneticVectorNoise    1.9876
9          MagnetometerBiasNoise     1.9770
10         AccelerometerNoise        1.9707
10         GyroscopeNoise            1.9707
10         MagnetometerNoise         1.9706
10         GPSPositionNoise          1.9564
10         GPSVelocityNoise          1.9005
10         QuaternionNoise           1.9003
10         AngularVelocityNoise      1.9003
10         PositionNoise             1.9003
```

```
10          VelocityNoise              1.9003
10          AccelerationNoise          1.8999
10          GyroscopeBiasNoise         1.8999
10          AccelerometerBiasNoise     1.8992
10          GeomagneticVectorNoise     1.8992
10          MagnetometerBiasNoise      1.8885
11          AccelerometerNoise         1.8815
11          GyroscopeNoise             1.8807
11          MagnetometerNoise          1.8806
11          GPSPositionNoise           1.8674
11          GPSVelocityNoise           1.8111
11          QuaternionNoise            1.8110
11          AngularVelocityNoise       1.8110
11          PositionNoise              1.8110
11          VelocityNoise              1.8110
11          AccelerationNoise          1.8094
11          GyroscopeBiasNoise         1.8094
11          AccelerometerBiasNoise     1.8085
11          GeomagneticVectorNoise     1.8085
11          MagnetometerBiasNoise      1.7984
12          AccelerometerNoise         1.7921
12          GyroscopeNoise             1.7919
12          MagnetometerNoise          1.7902
12          GPSPositionNoise           1.7774
12          GPSVelocityNoise           1.7315
12          QuaternionNoise            1.7313
12          AngularVelocityNoise       1.7282
12          PositionNoise              1.7282
12          VelocityNoise              1.7282
12          AccelerationNoise          1.7278
12          GyroscopeBiasNoise         1.7278
12          AccelerometerBiasNoise     1.7268
12          GeomagneticVectorNoise     1.7268
12          MagnetometerBiasNoise      1.7160
13          AccelerometerNoise         1.7103
13          GyroscopeNoise             1.7094
13          MagnetometerNoise          1.7094
13          GPSPositionNoise           1.6980
13          GPSVelocityNoise           1.6589
13          QuaternionNoise            1.6587
13          AngularVelocityNoise       1.6587
13          PositionNoise              1.6587
13          VelocityNoise              1.6587
13          AccelerationNoise          1.6575
13          GyroscopeBiasNoise         1.6575
13          AccelerometerBiasNoise     1.6566
13          GeomagneticVectorNoise     1.6566
13          MagnetometerBiasNoise      1.6480
14          AccelerometerNoise         1.6432
14          GyroscopeNoise             1.6422
14          MagnetometerNoise          1.6415
14          GPSPositionNoise           1.6330
14          GPSVelocityNoise           1.6097
14          QuaternionNoise            1.6095
14          AngularVelocityNoise       1.6093
14          PositionNoise              1.6093
14          VelocityNoise              1.6093
14          AccelerationNoise          1.6083
```

```
14          GyroscopeBiasNoise          1.6083
14          AccelerometerBiasNoise      1.6077
14          GeomagneticVectorNoise      1.6077
14          MagnetometerBiasNoise       1.6012
15          AccelerometerNoise          1.5972
15          GyroscopeNoise              1.5949
15          MagnetometerNoise           1.5942
15          GPSPositionNoise            1.5886
15          GPSVelocityNoise            1.5802
15          QuaternionNoise             1.5801
15          AngularVelocityNoise        1.5786
15          PositionNoise               1.5786
15          VelocityNoise               1.5786
15          AccelerationNoise           1.5786
15          GyroscopeBiasNoise          1.5786
15          AccelerometerBiasNoise      1.5783
15          GeomagneticVectorNoise      1.5783
15          MagnetometerBiasNoise       1.5737
```

```
[posTunedEst, orientTunedEst] = fuse(filtTuned,sensorData,tunedmn);
```

Compare Tuned vs Untuned Filter

Plot the position estimates from the tuned and untuned filters along with the ground truth positions. Then, plot the orientation error (quaternion distance) in degrees for both tuned and untuned filters. The tuned filter estimates the position and orientation better than the untuned filter.

```
% Position error
figure;
t = sensorData.Time;
subplot(3,1,1);
plot(t, [posTunedEst(:,1) posUntunedEst(:,1) Position(:,1)]);
title('Position');
ylabel('X-axis');
legend('Tuned', 'Untuned', 'Truth');
subplot(3,1,2);
plot(t, [posTunedEst(:,2) posUntunedEst(:,2) Position(:,2)]);
ylabel('Y-axis');
subplot(3,1,3);
plot(t, [posTunedEst(:,3) posUntunedEst(:,3) Position(:,3)]);
ylabel('Z-axis');
```
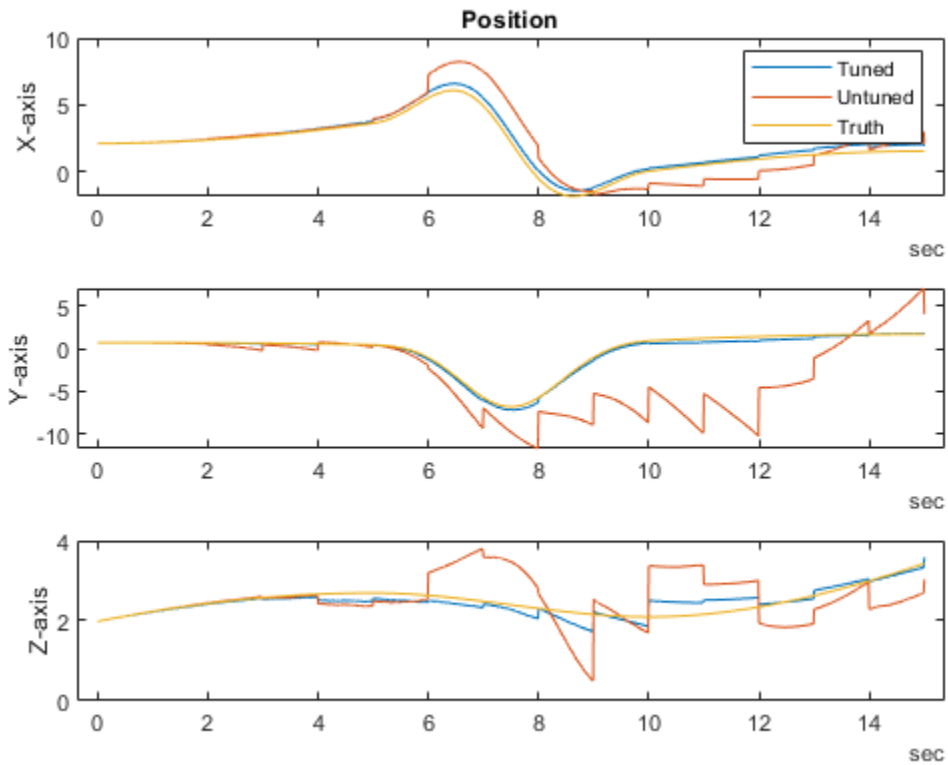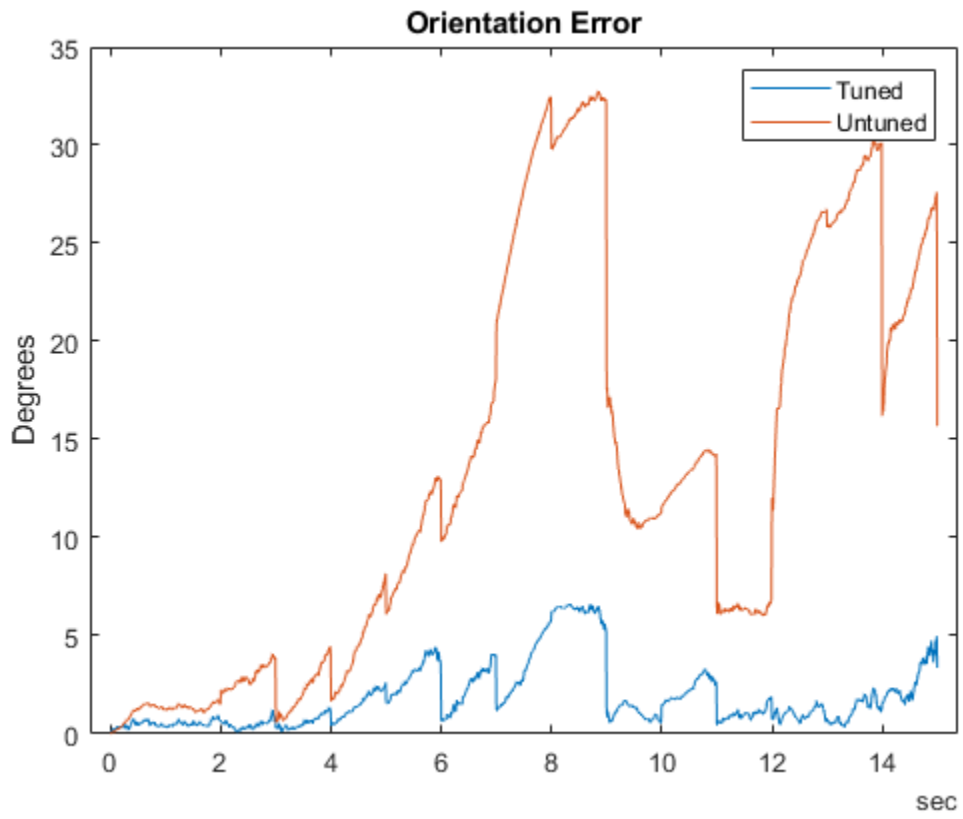
```
% Orientation Error
figure;
plot(t, rad2deg(dist(Orientation, orientTunedEst)), ...
    t, rad2deg(dist(Orientation, orientUntunedEst)));
title('Orientation Error');
ylabel('Degrees');
legend('Tuned', 'Untuned');
```

Detect, Classify, and Track Vehicles Using Lidar

This example shows how to detect, classify, and track vehicles by using lidar point cloud data captured by a lidar sensor mounted on an ego vehicle. The lidar data used in this example is recorded from a highway-driving scenario. In this example, the point cloud data is segmented to determine the class of objects using the `PointSeg` network. A joint probabilistic data association (JPDA) tracker with an interactive multiple model filter is used to track the detected vehicles.

Overview

The perception module plays an important role in achieving full autonomy for vehicles with an ADAS system. Lidar and camera are essential sensors in the perception workflow. Lidar is good at extracting accurate depth information of objects, while camera produces rich and detailed information of the environment which is useful for object classification.

This example mainly includes these parts:

• Ground plane segmentation
• Semantic segmentation
• Oriented bounding box fitting
• Tracking oriented bounding boxes

The flowchart gives an overview of the whole system.

5 Channel Input

Semantic Segmentation

Oriented Bounding Box
Fitting

Bounding Box
Tracking
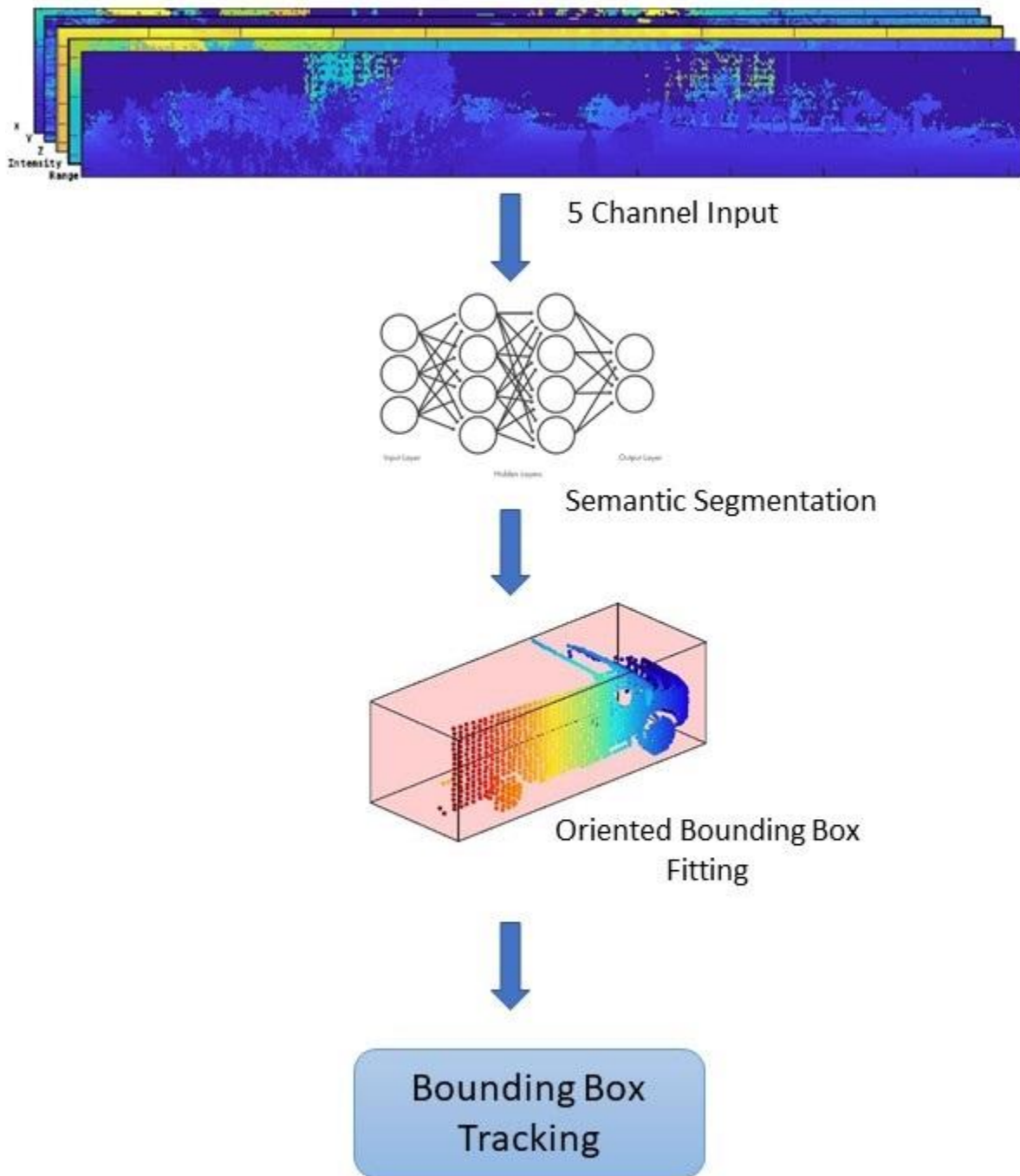
Load Data

The lidar sensor generates point cloud data either in an organized format or an unorganized format. The data used in this example is collected using an Ouster OS1 lidar sensor. This lidar produces an organized point cloud with 64 horizontal scan lines. The point cloud data is comprised of three channels, representing the *x*-, *y*-, and *z*-coordinates of the points. Each channel is of the size 64-by-1024.

```
dataFile = 'highwayData.mat';
data = load(dataFile);
```

Ground Plane Segmentation

This example employs a hybrid approach that uses the segmentGroundFromLidarData (Computer Vision Toolbox) and pcfitplane (Computer Vision Toolbox) functions. First, estimate the ground plane parameters using the segmentGroundFromLidarData function. The estimated ground plane is divided into strips along the direction of the vehicle in order to fit the plane, using the pcfitplane function on each strip. This hybrid approach robustly fits the ground plane in a piecewise manner and handles variations in the point cloud.

```
% Load point cloud
ptClouds = data.ptCloudData;
ptCloud = ptClouds{1};
% Define ROI for cropping point cloud
xLimit = [-30, 30];
yLimit = [-12, 12];
zLimit = [-3, 15];

roi = [xLimit, yLimit, zLimit];
% Extract ground plane
[nonGround, ground] = helperExtractGround(ptCloud, roi);
figure;
pcshowpair(nonGround, ground);
legend({'\color{white} Nonground','\color{white} Ground'},'Location','northeastoutside');
```
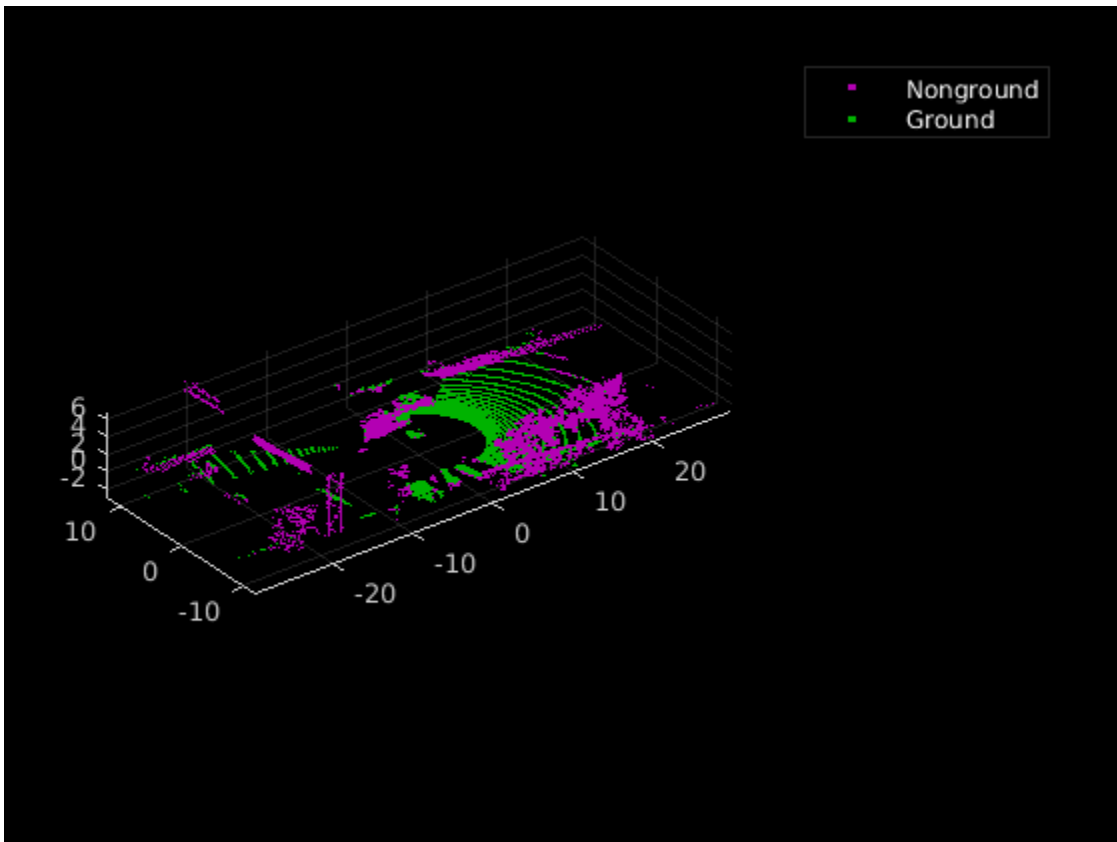
Semantic Segmentation

This example uses a pretrained `PointSeg` network model. `PointSeg` is an end-to-end real-time semantic segmentation network trained for object classes like cars, trucks, and background. The output from the network is a masked image with each pixel labeled per its class. This mask is used to filter different types of objects in the point cloud. The input to the network is five-channel image, that is *x, y, z, intensity, and range.* For more information on the network or how to train the network, refer to the Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network example

Prepare Input Data

The `helperPrepareData` function generates five-channel data from the loaded point cloud data.

```
% Load and visualize a sample frame
frame = helperPrepareData(ptCloud);
figure;
subplot(5, 1, 1);
imagesc(frame(:, :, 1));
title('X channel');

subplot(5, 1, 2);
imagesc(frame(:, :, 2));
title('Y channel');

subplot(5, 1, 3);
imagesc(frame(:, :, 3));
title('Z channel');

subplot(5, 1, 4);
imagesc(frame(:, :, 4));
title('Intensity channel');

subplot(5, 1, 5);
imagesc(frame(:, :, 5));
title('Range channel');
```
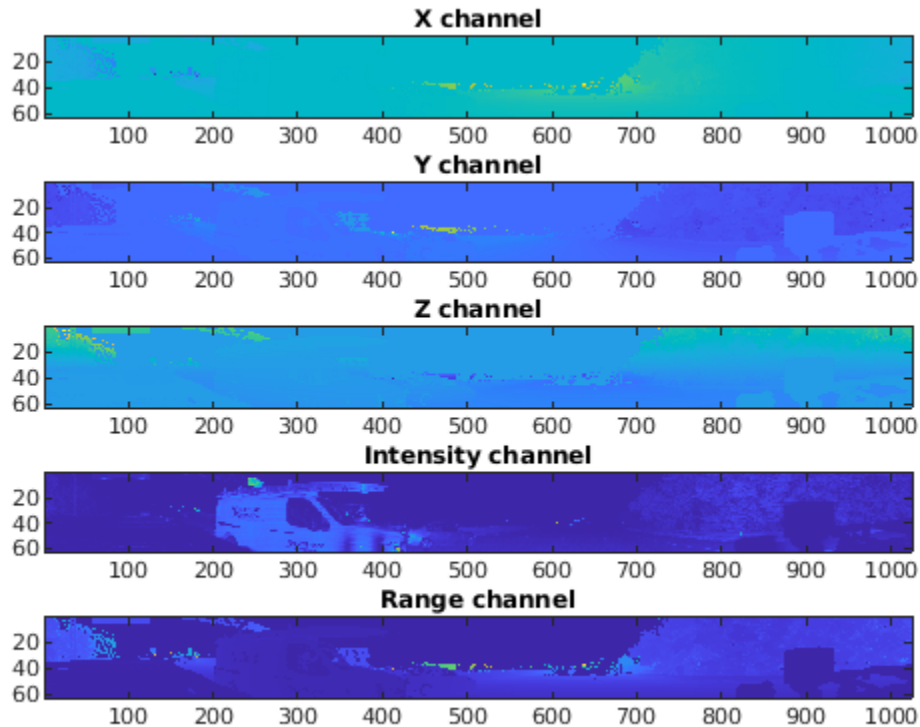
Load the pre-trained network and run forward inference on one frame.

```matlab
% Pretrained PointSeg model file
modelfile = 'pretrainedPointSegModel.mat';

if ~exist('net', 'var')
    load(modelfile);
end

% Define classes
classes = ["background", "car", "truck"];

% Define color map
lidarColorMap = [
            0.98  0.98   0.00  % unknown
            0.01  0.98   0.01  % green color for car
            0.01  0.01   0.98  % blue color for motorcycle
            ];

% Run forward pass
pxdsResults = semanticseg(frame, net);

% Overlay intensity image with segmented output
segmentedImage = labeloverlay(uint8(frame(:, :, 4)), pxdsResults, 'Colormap', lidarColorMap, 'Tra

% Display results
figure;
```

```
imshow(segmentedImage);
helperPixelLabelColorbar(lidarColorMap,classes);
```


Use the generated semantic mask to filter point clouds containing trucks. Similarly, filter point clouds for other classes.

```
truckIndices = pxdsResults == 'truck';
truckPointCloud = select(nonGround, truckIndices, 'OutputSize', 'full');

% Crop point cloud for better display
croppedPtCloud = select(ptCloud, findPointsInROI(ptCloud, roi));
croppedTruckPtCloud = select(truckPointCloud, findPointsInROI(truckPointCloud, roi));

% Display ground and nonground points
figure;
pcshowpair(croppedPtCloud,  croppedTruckPtCloud);
legend({'\color{white} Nonvehicle','\color{white} Vehicle'},'Location','northeastoutside');
```
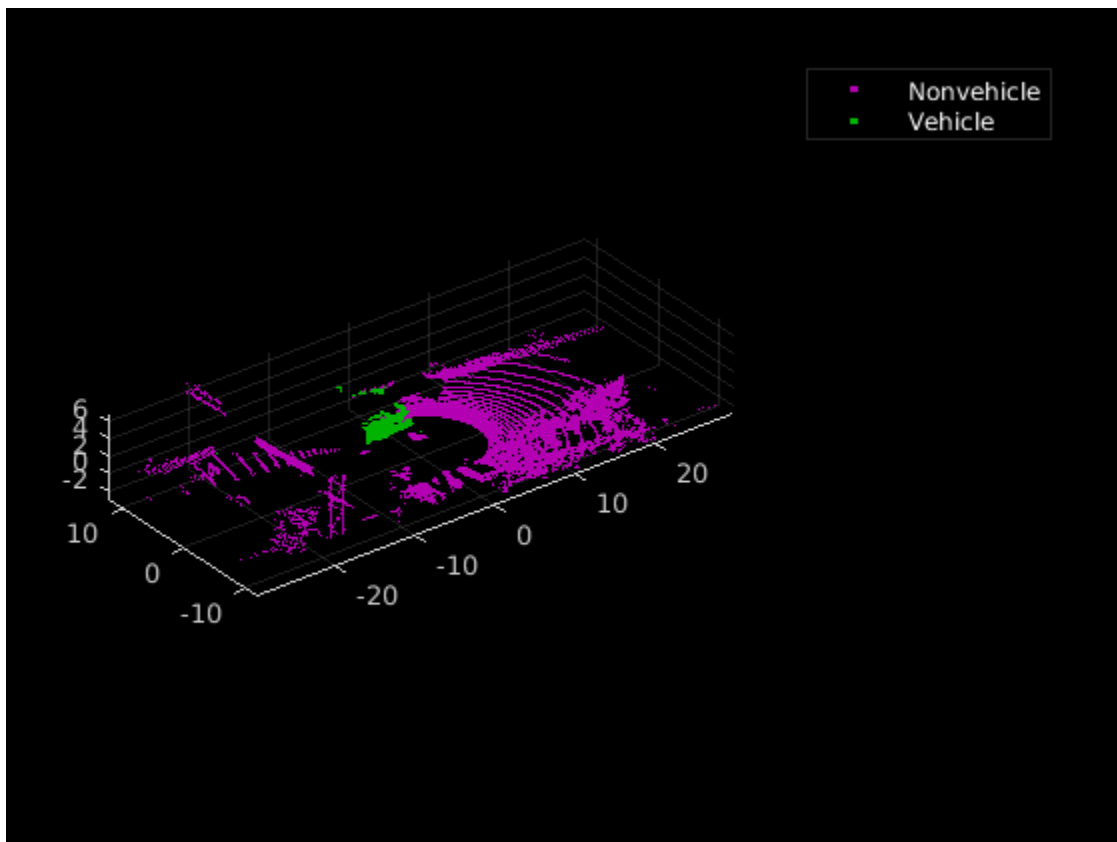
Clustering and Bounding Box Fitting

After extracting point clouds of different object classes, the objects are clustered by applying Euclidean clustering using the `pcsegdist` (Computer Vision Toolbox) function. To group all the points belonging to one single cluster, the point cloud obtained as a cluster is used as seed points for growing region in nonground points. Use the `findNearestNeighbors` (Computer Vision Toolbox) function to loop over all the points to grow the region. The extracted cluster is fitted in an L-shape bounding box using the `pcfitcuboid` (Lidar Toolbox) function. These clusters of vehicles resemble the shape of the letter L when seen from a top-down view. This feature helps in estimating the orientation of the vehicle. The oriented bounding box fitting helps in estimating the heading angle of the objects, which is useful in applications such as path planning, and traffic maneuvering traffic.

The cuboid boundaries of the clusters can also be calculated by finding the minimum and maximum spatial extents in each direction. However, this method fails in estimating the orientation of the detected vehicles. The difference between the two methods is shown in the figure.
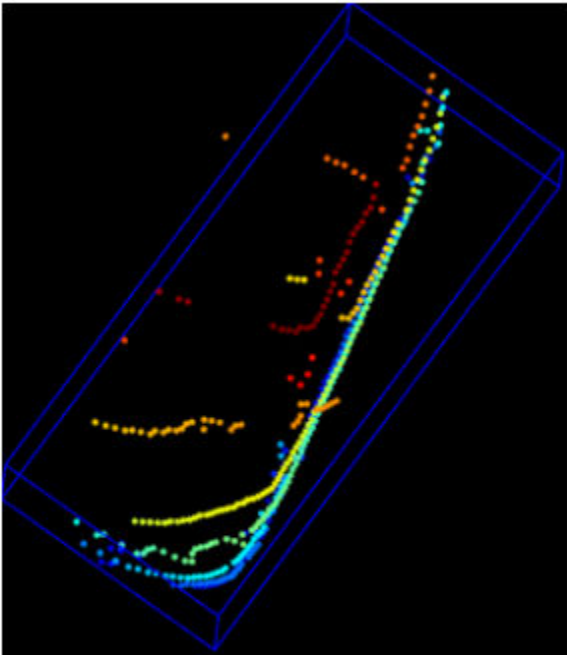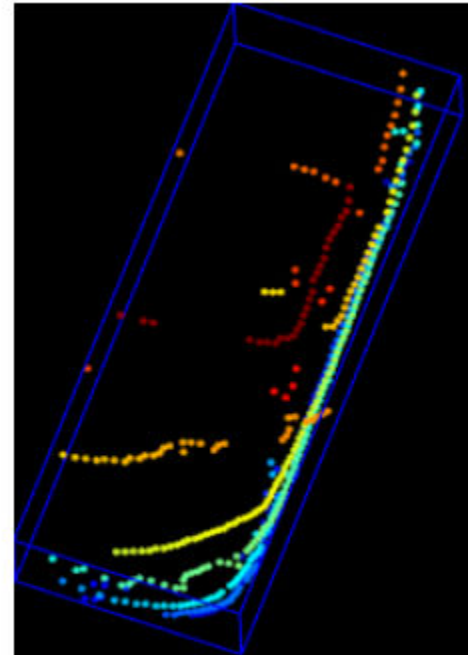
Min. Area Rectangle   L-Shape Fitting

```
[labels, numClusters] = pcsegdist(croppedTruckPtCloud, 1);

% Define cuboid parameters
params = zeros(0, 9);

for clusterIndex = 1:numClusters
    ptsInCluster = labels == clusterIndex;

    pc = select(croppedTruckPtCloud, ptsInCluster);
    location = pc.Location;

    xl = (max(location(:, 1)) - min(location(:, 1)));
    yl = (max(location(:, 2)) - min(location(:, 2)));
```

```
    zl = (max(location(:, 3)) - min(location(:, 3)));

    % Filter small bounding boxes
    if size(location, 1)*size(location, 2) > 20 && any(any(pc.Location)) && xl > 1 && yl > 1
        indices = zeros(0, 1);
        objectPtCloud = pointCloud(location);
        for i = 1:size(location, 1)
            seedPoint = location(i, :);
            indices(end+1) = findNearestNeighbors(nonGround, seedPoint, 1);
        end

        % Remove overlapping indices
        indices = unique(indices);

        % Fit oriented bounding box
        model = pcfitcuboid(select(nonGround, indices));
        params(end+1, :) = model.Parameters;
    end
end

% Display point cloud and detected bounding box
figure;
pcshow(croppedPtCloud.Location, croppedPtCloud.Location(:, 3));
showShape('cuboid', params, "Color", "red", "Label", "Truck");
```
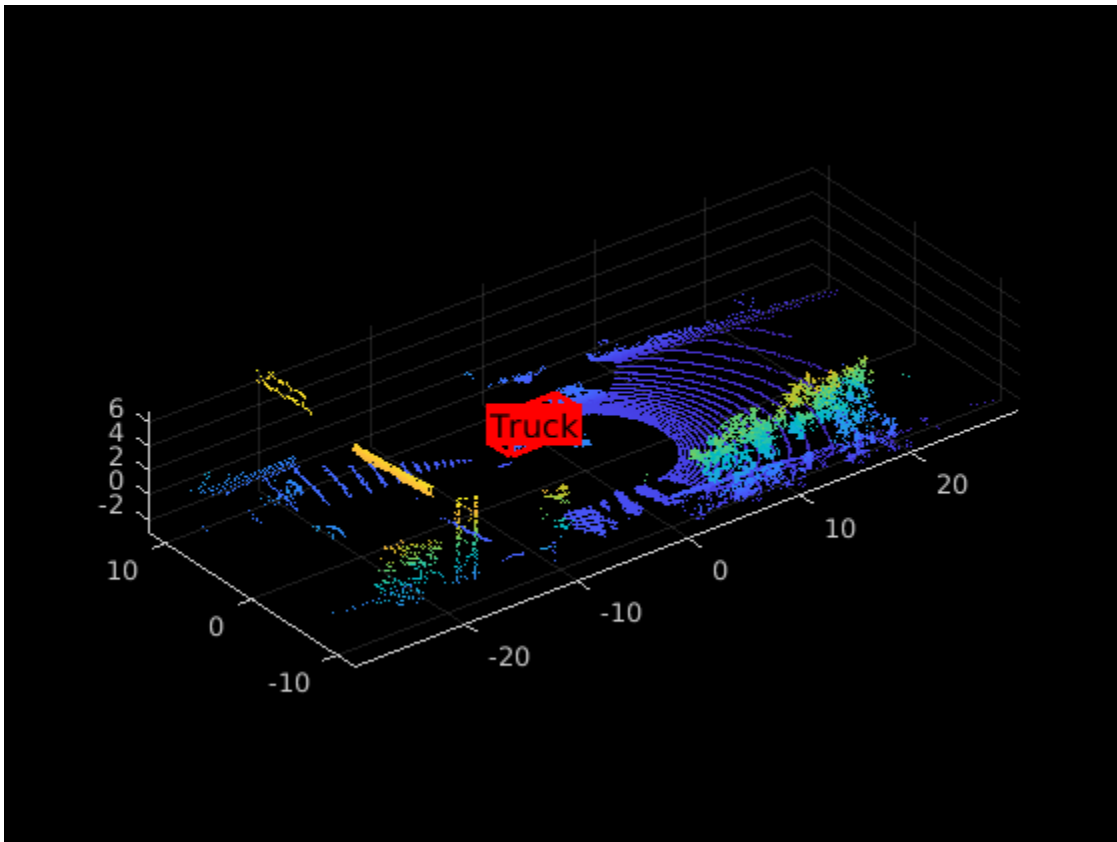
Visualization Setup

Use the helperLidarObjectDetectionDisplay class to visualize the complete workflow in one window. The layout of the visualization window is divided into the following sections:

1 Lidar Range Image: point cloud image in 2-D as a range image
2 Segmented Image: Detected labels generated from the semantic segmentation network overlaid with the intensity image or the fourth channel of the data
3 Oriented Bounding Box Detection: 3-D point cloud with oriented bounding boxes
4 Top View: Top view of the point cloud with oriented bounding boxes

```
display = helperLidarObjectDetectionDisplay;
```

Loop Through Data

The helperLidarObjectDetection class is a wrapper encapsulating all the segmentation, clustering, and bounding box fitting steps mentioned in the above sections. Use the findDetections function to extract the detected objects.

```
% Initialize lidar object detector
lidarDetector = helperLidarObjecDetector('ModelFile',modelfile, 'XLimits', xLimit,...
    'YLimit', yLimit, 'ZLimit', zLimit);

% Prepare 5-D lidar data
inputData = helperPrepareData(data.ptCloudData);

% Set random number generator for reproducible results.
S = rng(2018);

% Initialize the display
initializeDisplay(display);

numFrames = numel(inputData);
for count = 1:numFrames

    % Get current data
    input = inputData{count};

    rangeImage = input(:, :, 5);

    % Extact bounding boxes from lidar data
    [boundingBox, coloredPtCloud, pointLabels] = detectBbox(lidarDetector, input);

    % Update display with colored point cloud
    updatePointCloud(display, coloredPtCloud);

    % Update bounding boxes
    updateBoundingBox(display, boundingBox);

    % Update segmented image
    updateSegmentedImage(display, pointLabels, rangeImage);

    drawnow('limitrate');
end
```
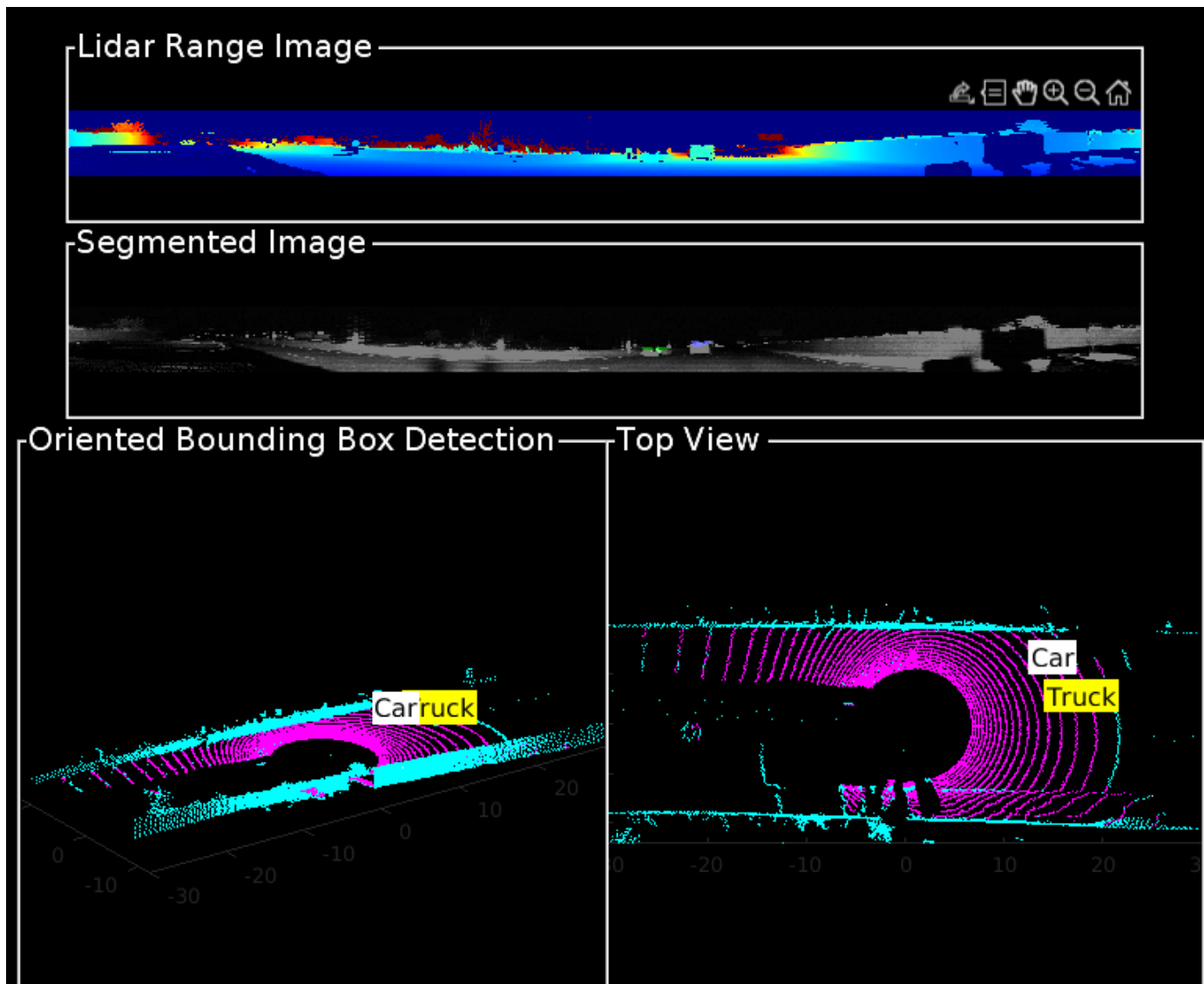
Tracking Oriented Bounding Boxes

In this example, you use a joint probabilistic data association (JPDA) tracker. The time step `dt` is set to 0.1 seconds since the dataset is captured at 10 Hz. The state-space model used in the tracker is based on a cuboid model with parameters, $[x, y, z, \phi, l, w, h]$. For more details on how to track bounding boxes in lidar data, see Track Vehicles Using Lidar: From Point Cloud to Track List. In this example, the class information is provided using the `ObjectAttributes` property of the `objectDetection object`. When creating new tracks, the filter initialization function, defined using the helper function `helperMultiClassInitIMMFilter` uses the class of the detection to set up initial dimensions of the object. This helps the tracker to adjust bounding box measurement model with the appropriate dimensions of the track.

Set up a JPDA tracker object with these parameters.

```
assignmentGate = [10 100]; % Assignment threshold;
confThreshold = [7 10];    % Confirmation threshold for history logi
delThreshold = [2 3];      % Deletion threshold for history logic
```

```
Kc = 1e-5;                      % False-alarm rate per unit volume

% IMM filter initialization function
filterInitFcn = @helperMultiClassInitIMMFilter;

% A joint probabilistic data association tracker with IMM filter
tracker = trackerJPDA('FilterInitializationFcn',filterInitFcn,...
    'TrackLogic','History',...
    'AssignmentThreshold',assignmentGate,...
    'ClutterDensity',Kc,...
    'ConfirmationThreshold',confThreshold,...
    'DeletionThreshold',delThreshold,'InitializationThreshold',0);

allTracks = struct([]);
time = 0;
dt = 0.1;

% Define Measurement Noise
measNoise = blkdiag(0.25*eye(3),25,eye(3));

numTracks = zeros(numFrames, 2);
```

The detected objects are assembled as a cell array of `objectDetection` (Automated Driving Toolbox) objects using the `helperAssembleDetections` function.

```
display = helperLidarObjectDetectionDisplay;
initializeDisplay(display);

for count = 1:numFrames
    time = time + dt;
    % Get current data
    input = inputData{count};

    rangeImage = input(:, :, 5);

    % Extact bounding boxes from lidar data
    [boundingBox, coloredPtCloud, pointLabels] = detectBbox(lidarDetector, input);

    % Assemble bounding boxes into objectDetections
    detections = helperAssembleDetections(boundingBox, measNoise, time);

    % Pass detections to tracker
    if ~isempty(detections)
        % Update the tracker
        [confirmedTracks, tentativeTracks, allTracks, info] = tracker(detections, time);
        numTracks(count, 1) = numel(confirmedTracks);
    end

    % Update display with colored point cloud
    updatePointCloud(display, coloredPtCloud);

    % Update segmented image
    updateSegmentedImage(display, pointLabels, rangeImage);

    % Update the display if the tracks are not empty
    if ~isempty(confirmedTracks)
        updateTracks(display, confirmedTracks);
    end
```
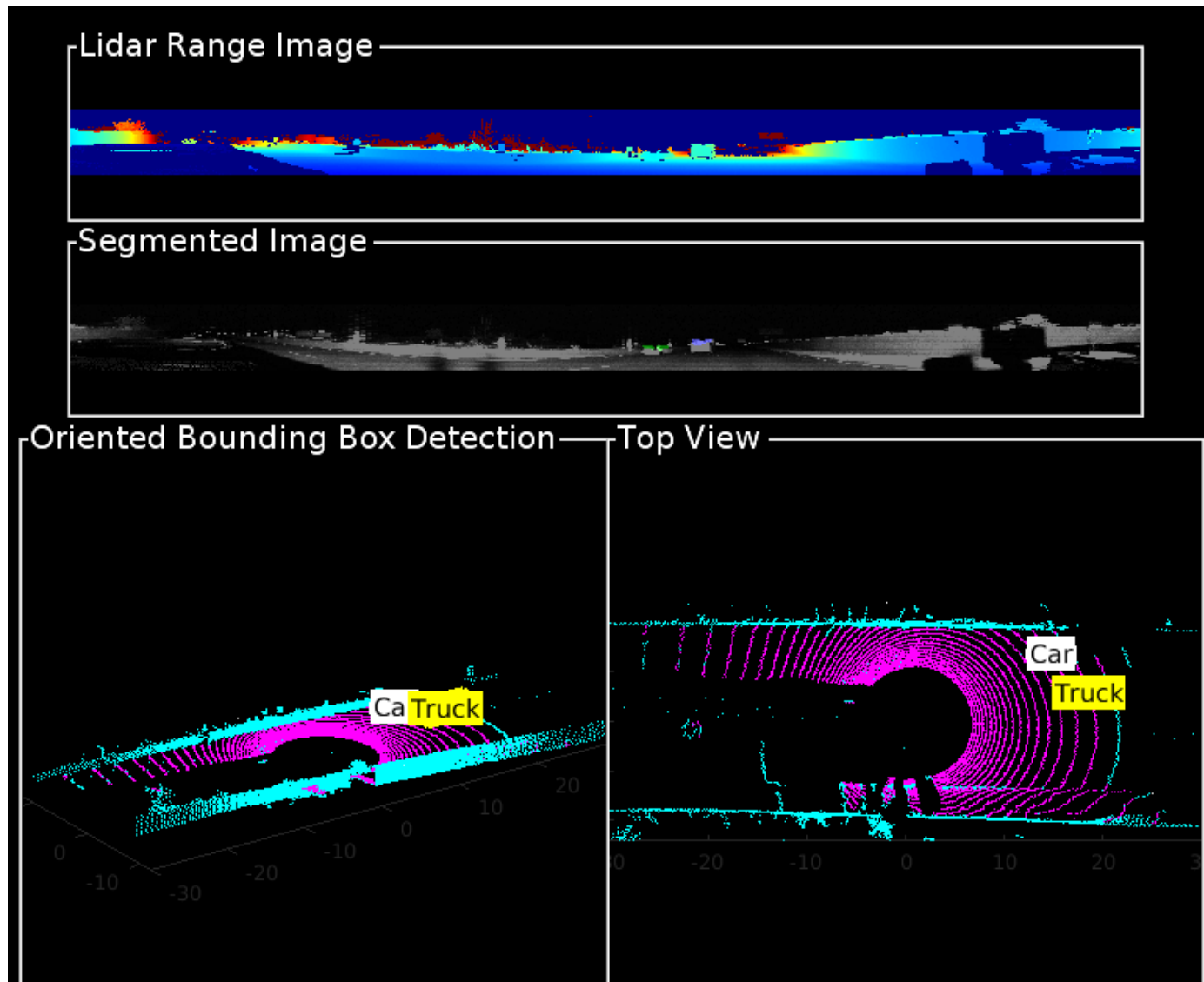
5-689

```
        drawnow('limitrate');
end
```


Summary

This example showed how to detect and classify vehicles fitted with oriented bounding box on lidar data. You also learned how to use IMM filter to track objects with multiple class information. The semantic segmentation results can be improved further by adding more training data.

Supporting Functions

helperPrepareData

```
function multiChannelData = helperPrepareData(input)
% Create 5-channel data as x, y, z, intensity and range
% of size 64-by-1024-by-5 from pointCloud.
```

```matlab
if isa(input, 'cell')
    numFrames = numel(input);
    multiChannelData = cell(1, numFrames);
    for i = 1:numFrames
        inputData = input{i};

        x = inputData.Location(:,:,1);
        y = inputData.Location(:,:,2);
        z = inputData.Location(:,:,3);

        intensity = inputData.Intensity;
        range = sqrt(x.^2 + y.^2 + z.^2);

        multiChannelData{i} = cat(3, x, y, z, intensity, range);
    end
else
    x = input.Location(:,:,1);
    y = input.Location(:,:,2);
    z = input.Location(:,:,3);

    intensity = input.Intensity;
    range = sqrt(x.^2 + y.^2 + z.^2);

    multiChannelData = cat(3, x, y, z, intensity, range);
end
end
```

pixelLabelColorbar

```matlab
function helperPixelLabelColorbar(cmap, classNames)
% Add a colorbar to the current axis. The colorbar is formatted
% to display the class names with the color.

colormap(gca,cmap)

% Add colorbar to current figure.
c = colorbar('peer', gca);

% Use class names for tick marks.
c.TickLabels = classNames;
numClasses = size(cmap,1);

% Center tick labels.
c.Ticks = 1/(numClasses*2):1/numClasses:1;

% Remove tick mark.
c.TickLength = 0;
end
```

helperExtractGround

```matlab
function [ptCloudNonGround, ptCloudGround] = helperExtractGround(ptCloudIn, roi)
% Crop the point cloud

idx = findPointsInROI(ptCloudIn, roi);
pc = select(ptCloudIn, idx, 'OutputSize', 'full');

% Get the ground plane the indices using piecewise plane fitting
```

```matlab
    [ptCloudGround, idx] = piecewisePlaneFitting(pc, roi);

    nonGroundIdx = true(size(pc.Location, [1,2]));
    nonGroundIdx(idx) = false;
    ptCloudNonGround = select(pc, nonGroundIdx, 'OutputSize', 'full');
end


function [groundPlane, idx] = piecewisePlaneFitting(ptCloudIn, roi)
groundPtsIdx = ...
    segmentGroundFromLidarData(ptCloudIn, ...
    'ElevationAngleDelta', 5, 'InitialElevationAngle', 15);
groundPC = select(ptCloudIn,groundPtsIdx, 'OutputSize', 'full');

% Divide x-axis in 3 regions
segmentLength = (roi(2) - roi(1))/3;

x1 = [roi(1), roi(1) + segmentLength];
x2 = [x1(2), x1(2) + segmentLength];
x3 = [x2(2), x2(2) + segmentLength];

roi1 = [x1, roi(3: end)];
roi2 = [x2, roi(3: end)];
roi3 = [x3, roi(3: end)];

idxBack = findPointsInROI(groundPC, roi1);
idxCenter = findPointsInROI(groundPC, roi2);
idxForward = findPointsInROI(groundPC, roi3);

% Break the point clouds in front and back
ptBack = select(groundPC, idxBack, 'OutputSize', 'full');

ptForward = select(groundPC, idxForward, 'OutputSize', 'full');

[~, inliersForward] = planeFit(ptForward);
[~, inliersBack] = planeFit(ptBack);
idx = [inliersForward; idxCenter; inliersBack];
groundPlane = select(ptCloudIn, idx, 'OutputSize', 'full');
end

function [plane, inlinersIdx] = planeFit(ptCloudIn)
[~, inlinersIdx, ~] = pcfitplane(ptCloudIn, 1, [0, 0, 1]);
plane = select(ptCloudIn, inlinersIdx, 'OutputSize', 'full');
end
```

helperAssembleDetections

```matlab
function mydetections = helperAssembleDetections(bboxes, measNoise, timestamp)
% Assemble bounding boxes as cell array of objectDetection

mydetections = cell(size(bboxes, 1), 1);
for i = 1:size(bboxes, 1)
    classid = bboxes(i, end);
    lidarModel = [bboxes(i, 1:3), bboxes(i, end-1), bboxes(i, 4:6)];
    % To avoid direct confirmation by the tracker, the ClassID is passed as
    % ObjectAttributes.
    mydetections{i} = objectDetection(timestamp, ...
        lidarModel', 'MeasurementNoise',...
```

```
        measNoise, 'ObjectAttributes', struct('ClassID',classid));
end
end
```

References

[1] Xiao Zhang, Wenda Xu, Chiyu Dong and John M. Dolan, "Efficient L-Shape Fitting for Vehicle Detection Using Laser Scanners", IEEE Intelligent Vehicles Symposium, June 2017

[2] Y. Wang, T. Shi, P. Yun, L. Tai, and M. Liu, "Pointseg: Real-time semantic segmentation based on 3d lidar point cloud," arXiv preprint arXiv:1807.06288, 2018.